pytest-django

Oct 16, 2018

Contents

pytest-django is a plugin for pytest that provides a set of useful tools for testing Django applications and projects.

CHAPTER 1

Quick Start

```
$ pip install pytest-django
```

Make sure `DJANGO_SETTINGS_MODULE` is defined (see *Configuring Django settings*) and make your tests discoverable (see *My tests are not being found. Why?*):

```
# -- FILE: pytest.ini (or tox.ini)
[pytest]
DJANGO_SETTINGS_MODULE = test_settings
# -- recommended but optional:
python_files = tests.py test_*.py *_tests.py
```

Run your tests with `pytest`:

```
$ pytest
```

CHAPTER 2

Why would I use this instead of Django's manage.py test command?

Running the test suite with pytest offers some features that are not present in Django's standard test mechanism:

- Less boilerplate: no need to import unittest, create a subclass with methods. Just write tests as regular functions.
- Manage test dependencies with fixtures.
- Run tests in multiple processes for increased speed.
- There are a lot of other nice plugins available for pytest.
- Easy switching: Existing unittest-style tests will still work without any modifications.

See the pytest documentation for more information on pytest.

CHAPTER 3

Bugs? Feature Suggestions?

Report issues and feature requests at the GitHub issue tracker.

CHAPTER 4

Table of Contents

4.1 Getting started with pytest and pytest-django

4.1.1 Introduction

pytest and pytest-django are compatible with standard Django test suites and Nose test suites. They should be able to pick up and run existing tests without any or little configuration. This section describes how to get started quickly.

4.1.2 Talks, articles and blog posts

- Talk from DjangoCon Europe 2014: pytest: helps you write better Django apps, by Andreas Pelme
- Talk from EuroPython 2013: Testing Django application with pytest, by Andreas Pelme
- Three part blog post tutorial (part 3 mentions Django integration): pytest: no-boilerplate testing, by Daniel Greenfeld
- Blog post: Django Projects to Django Apps: Converting the Unit Tests, by John Costa.

For general information and tutorials on pytest, see the pytest tutorial page.

4.1.3 Step 1: Installation

pytest-django can be obtained directly from PyPI, and can be installed with `pip`:

```
pip install pytest-django
```

Installing pytest-django will also automatically install the latest version of pytest. `pytest-django` uses `pytest`'s plugin system and can be used right away after installation, there is nothing more to configure.

4.1.4 Step 2: Point pytest to your Django settings

You need to tell pytest which Django settings that should be used for test runs. The easiest way to achieve this is to create a pytest configuration file with this information.

Create a file called `pytest.ini` in your project root directory that contains:

```
[pytest]
DJANGO_SETTINGS_MODULE = yourproject.settings
```

You can also specify your Django settings by setting the `DJANGO_SETTINGS_MODULE` environment variable or specifying the `--ds=yourproject.settings` command line flag when running the tests. See the full documentation on *Configuring Django settings*.

Optionally, also add the following line to the `[pytest]` section to instruct pytest to collect tests in Django's default app layouts, too. See the FAQ at *My tests are not being found. Why?* for more infos.

```
python_files = tests.py test_*.py *_tests.py
```

4.1.5 Step 3: Run your test suite

Tests are invoked directly with the `pytest` command, instead of `manage.py test`, that you might be used to:

```
pytest
```

Do you have problems with pytest not finding your code? See the FAQ *I see an error saying "could not import myproject.settings"*.

4.1.6 Next steps

The *Usage and invocations* section describes more ways to interact with your test suites.

pytest-django also provides some *Django helpers* to make it easier to write Django tests.

Consult the pytest documentation for more information on pytest itself.

4.1.7 Stuck? Need help?

No problem, see the FAQ on *How can I use manage.py test with pytest-django?* for information on how to get help.

4.2 Configuring Django settings

There are a couple of different ways Django settings can be provided for the tests.

4.2.1 The environment variable `DJANGO_SETTINGS_MODULE`

Running the tests with DJANGO_SETTINGS_MODULE defined will find the Django settings the same way Django does by default.

Example:

```
$ export DJANGO_SETTINGS_MODULE=test_settings
$ pytest
```

or:

```
$ DJANGO_SETTINGS_MODULE=test_settings pytest
```

4.2.2 Command line option `--ds=SETTINGS`

Example:

```
$ pytest --ds=test_settings
```

4.2.3 `pytest.ini` settings

Example contents of pytest.ini:

```
[pytest]
DJANGO_SETTINGS_MODULE = test_settings
```

4.2.4 Order of choosing settings

The order of precedence is, from highest to lowest:

- The command line option `--ds`
- The environment variable `DJANGO_SETTINGS_MODULE`
- The `DJANGO_SETTINGS_MODULE` option in the configuration file - `pytest.ini`, or other file that Pytest finds such as `tox.ini`

If you want to use the highest precedence in the configuration file, you can use `addopts = --ds=yourtestsettings`.

4.2.5 Using django-configurations

There is support for using django-configurations.

To do so configure the settings class using an environment variable, the `--dc` flag, or `pytest.ini` option `DJANGO_CONFIGURATION`.

Environment Variable:

```
$ export DJANGO_CONFIGURATION=MySettings
$ pytest
```

Command Line Option:

```
$ pytest --dc=MySettings
```

INI File Contents:

```
[pytest]
DJANGO_CONFIGURATION=MySettings
```

4.2.6 Using `django.conf.settings.configure()`

Django settings can be set up by calling `django.conf.settings.configure()`.

This can be done from your project's `conftest.py` file:

```
from django.conf import settings

def pytest_configure():
    settings.configure(DATABASES=...)
```

4.2.7 Changing your app before Django gets set up

pytest-django calls `django.setup()` automatically. If you want to do anything before this, you have to create a pytest plugin and use the `pytest_load_initial_conftests()` hook, with `tryfirst=True`, so that it gets run before the hook in pytest-django itself:

```
@pytest.hookimpl(tryfirst=True)
def pytest_load_initial_conftests(early_config, parser, args):
    import project.app.signals

    def noop(*args, **kwargs):
        pass

    project.app.signals.something = noop
```

This plugin can then be used e.g. via `-p` in `addopts`.

4.3 Managing the Python path

pytest needs to be able to import the code in your project. Normally, when interacting with Django code, the interaction happens via `manage.py`, which will implicilty add that directory to the Python path.

However, when Python is started via the `pytest` command, some extra care is needed to have the Python path setup properly. There are two ways to handle this problem, described below.

4.3.1 Automatic looking for of Django projects

By default, pytest-django tries to find Django projects by automatically looking for the project's `manage.py` file and adding its directory to the Python path.

Looking for the `manage.py` file uses the same algorithm as pytest uses to find `pytest.ini`, `tox.ini` and `setup.cfg`: Each test root directories parents will be searched for `manage.py` files, and it will stop when the first file is found.

If you have a custom project setup, have none or multiple `manage.py` files in your project, the automatic detection may not be correct. See *Managing the Python path explicitly* for more details on how to configure your environment in that case.

4.3.2 Managing the Python path explicitly

First, disable the automatic Django project finder. Add this to `pytest.ini`, `setup.cfg` or `tox.ini`:

```
[pytest]
django_find_project = false
```

Next, you need to make sure that your project code is available on the Python path. There are multiple ways to achieve this:

Managing your project with virtualenv, pip and editable mode

The easiest way to have your code available on the Python path when using virtualenv and pip is to have a setup.py file and install your project in editable mode when developing.

If you don't already have a setup.py file, creating a setup.py file with this content will get you started:

```
import setuptools
setuptools.setup(name='myproj', version='1.0')
```

This `setup.py` file is not sufficient to distribute your package to PyPI or more general packaging, but it should help you get started. Please refer to the Python Packaging User Guide for more information on packaging Python applications.'

To install the project afterwards:

```
pip install --editable .
```

Your code should then be importable from any Python application. You can also add this directly to your project's requirements.txt file like this:

```
# requirements.txt
-e .
django>=1.11
pytest-django
```

Using pytest-pythonpath

You can also use the pytest-pythonpath plugin to explicitly add paths to the Python path.

4.4 Usage and invocations

4.4.1 Basic usage

When using pytest-django, django-admin.py or manage.py is not used to run tests. This makes it possible to invoke pytest and other plugins with all its different options directly.

Running a test suite is done by invoking the pytest command directly:

```
pytest
```

Specific test files or directories can be selected by specifying the test file names directly on the command line:

```
pytest test_something.py a_directory
```

See the pytest documentation on Usage and invocations for more help on available parameters.

4.4.2 Additional command line options

`--fail-on-template-vars` - fail for invalid variables in templates

Fail tests that render templates which make use of invalid template variables.

4.4.3 Running tests in parallel with pytest-xdist

pytest-django supports running tests on multiple processes to speed up test suite run time. This can lead to significant speed improvements on multi core/multi CPU machines.

This requires the pytest-xdist plugin to be available, it can usually be installed with:

```
pip install pytest-xdist
```

You can then run the tests by running:

```
pytest -n <number of processes>
```

When tests are invoked with xdist, pytest-django will create a separate test database for each process. Each test database will be given a suffix (something like "gw0", "gw1") to map to a xdist process. If your database name is set to "foo", the test database with xdist will be "test_foo_gw0", "test_foo_gw1" etc.

See the full documentation on pytest-xdist for more information. Among other features, pytest-xdist can distribute/coordinate test execution on remote machines.

4.5 Database creation/re-use

`pytest-django` takes a conservative approach to enabling database access. By default your tests will fail if they try to access the database. Only if you explicitly request database access will this be allowed. This encourages you to keep database-needing tests to a minimum which is a best practice since next-to-no business logic should be requiring the database. Moreover it makes it very clear what code uses the database and catches any mistakes.

4.5.1 Enabling database access in tests

You can use pytest marks to tell `pytest-django` your test needs database access:

```
import pytest

@pytest.mark.django_db
def test_my_user():
    me = User.objects.get(username='me')
    assert me.is_superuser
```

It is also possible to mark all tests in a class or module at once. This demonstrates all the ways of marking, even though they overlap. Just one of these marks would have been sufficient. See the pytest documentation for detail:

```
import pytest

pytestmark = pytest.mark.django_db

@pytest.mark.django_db
class TestUsers:
    pytestmark = pytest.mark.django_db
    def test_my_user(self):
        me = User.objects.get(username='me')
        assert me.is_superuser
```

By default `pytest-django` will set up the Django databases the first time a test needs them. Once setup the database is cached for used for all subsequent tests and rolls back transactions to isolate tests from each other. This is the same way the standard Django TestCase uses the database. However `pytest-django` also caters for transaction test cases and allows you to keep the test databases configured across different test runs.

4.5.2 Testing transactions

Django itself has the `TransactionTestCase` which allows you to test transactions and will flush the database between tests to isolate them. The downside of this is that these tests are much slower to set up due to the required flushing of the database. `pytest-django` also supports this style of tests, which you can select using an argument to the `django_db` mark:

```
@pytest.mark.django_db(transaction=True)
def test_spam():
    pass  # test relying on transactions
```

4.5.3 Tests requiring multiple databases

Currently `pytest-django` does not specifically support Django's multi-database support.

You can however use normal `TestCase` instances to use its Tests and multiple databases support. In particular, if your database is configured for replication, be sure to read about Testing primary/replica configurations.

If you have any ideas about the best API to support multiple databases directly in `pytest-django` please get in touch, we are interested in eventually supporting this but unsure about simply following Django's approach.

See *https://github.com/pytest-dev/pytest-django/pull/431* for an idea / discussion to approach this.

4.5.4 `--reuse-db` - reuse the testing database between test runs

Using `--reuse-db` will create the test database in the same way as `manage.py test` usually does.

However, after the test run, the test database will not be removed.

The next time a test run is started with `--reuse-db`, the database will instantly be re used. This will allow much faster startup time for tests.

This can be especially useful when running a few tests, when there are a lot of database tables to set up.

`--reuse-db` will not pick up schema changes between test runs. You must run the tests with `--reuse-db --create-db` to re-create the database according to the new schema. Running without `--reuse-db` is also possible, since the database will automatically be re-created.

4.5.5 `--create-db` - force re creation of the test database

When used with `--reuse-db`, this option will re-create the database, regardless of whether it exists or not.

4.5.6 Example work flow with `--reuse-db` and `--create-db`.

A good way to use `--reuse-db` and `--create-db` can be:

- Put `--reuse-db` in your default options (in your project's `pytest.ini` file):

```
[pytest]
addopts = --reuse-db
```

- Just run tests with `pytest`, on the first run the test database will be created. The next test run it will be reused.
- When you alter your database schema, run `pytest --create-db`, to force re-creation of the test database.

4.5.7 `--nomigrations` - Disable Django 1.7+ migrations

Using `--nomigrations` will disable Django migrations and create the database by inspecting all models. It may be faster when there are several migrations to run in the database setup. You can use `--migrations` to force running migrations in case `--nomigrations` is used, e.g. in `setup.cfg`.

4.5.8 Advanced database configuration

pytest-django provides options to customize the way database is configured. The default database construction mostly follows Django's own test runner. You can however influence all parts of the database setup process to make it fit in projects with special requirements.

This section assumes some familiary with the Django test runner, Django database creation and pytest fixtures.

Fixtures

There are some fixtures which will let you change the way the database is configured in your own project. These fixtures can be overridden in your own project by specifying a fixture with the same name and scope in `conftest.py`.

Use the pytest-django source code

The default implementation of these fixtures can be found in fixtures.py.

The code is relatively short and straightforward and can provide a starting point when you need to customize database setup in your own project.

---

django_db_setup

This is the top-level fixture that ensures that the test databases are created and available. This fixture is session scoped (it will be run once per test session) and is responsible for making sure the test database is available for tests that need it.

The default implementation creates the test database by applying migrations and removes databases after the test run.

You can override this fixture in your own `conftest.py` to customize how test databases are constructed.

django_db_modify_db_settings

This fixture allows modifying *django.conf.settings.DATABASES* just before the databases are configured.

If you need to customize the location of your test database, this is the fixture you want to override.

The default implementation of this fixture requests the *django_db_modify_db_settings_xdist_suffix* to provide compatibility with pytest-xdist.

This fixture is by default requested from *django_db_setup*.

django_db_modify_db_settings_xdist_suffix

Requesting this fixture will add a suffix to the database name when the tests are run via pytest-xdist.

This fixture is by default requested from *django_db_modify_db_settings*.

django_db_use_migrations

Returns whether or not to use migrations to create the test databases.

The default implementation returns the value of the `--migrations`/`--nomigrations` command line options.

This fixture is by default requested from *django_db_setup*.

django_db_keepdb

Returns whether or not to re-use an existing database and to keep it after the test run.

The default implementation handles the `--reuse-db` and `--create-db` command line options.

This fixture is by default requested from *django_db_setup*.

django_db_createdb

Returns whether or not the database is to be re-created before running any tests.

This fixture is by default requested from *django_db_setup*.

django_db_blocker

Warning: It does not manage transactions and changes made to the database will not be automatically restored. Using the `pytest.mark.django_db` marker or *db* fixture, which wraps database changes in a transaction and restores the state is generally the thing you want in tests. This marker can be used when you are trying to influence the way the database is configured.

Database access is by default not allowed. `django_db_blocker` is the object which can allow specific code paths to have access to the database. This fixture is used internally to implement the `db` fixture.

django_db_blocker can be used as a context manager to enable database access for the specified block:

```
@pytest.fixture
def myfixture(django_db_blocker):
    with django_db_blocker.unblock():
        ...  # modify something in the database
```

You can also manage the access manually via these methods:

`django_db_blocker.`**`unblock`**`()`
: Enable database access. Should be followed by a call to *restore()*.

`django_db_blocker.`**`block`**`()`
: Disable database access. Should be followed by a call to *restore()*.

`django_db_blocker.`**`restore`**`()`
: Restore the previous state of the database blocking.

Examples

Using a template database for tests

This example shows how a pre-created PostgreSQL source database can be copied and used for tests.

Put this into `conftest.py`:

```
import pytest
from django.db import connections

import psycopg2
from psycopg2.extensions import ISOLATION_LEVEL_AUTOCOMMIT


def run_sql(sql):
    conn = psycopg2.connect(database='postgres')
    conn.set_isolation_level(ISOLATION_LEVEL_AUTOCOMMIT)
    cur = conn.cursor()
    cur.execute(sql)
    conn.close()


@pytest.yield_fixture(scope='session')
def django_db_setup():
    from django.conf import settings

    settings.DATABASES['default']['NAME'] = 'the_copied_db'

    run_sql('DROP DATABASE IF EXISTS the_copied_db')
    run_sql('CREATE DATABASE the_copied_db TEMPLATE the_source_db')

    yield

    for connection in connections.all():
        connection.close()

    run_sql('DROP DATABASE the_copied_db')
```

Using an existing, external database for tests

This example shows how you can connect to an existing database and use it for your tests. This example is trivial, you just need to disable all of pytest-django and Django's test database creation and point to the existing database. This is achieved by simply implementing a no-op *django_db_setup* fixture.

Put this into `conftest.py`:

```
import pytest


@pytest.fixture(scope='session')
def django_db_setup():
    settings.DATABASES['default'] = {
        'ENGINE': 'django.db.backends.mysql',
        'HOST': 'db.example.com',
        'NAME': 'external_db',
    }
```

Populate the database with initial test data

This example shows how you can populate the test database with test data. The test data will be saved in the database, i.e. it will not just be part of a transactions. This example uses Django's fixture loading mechanism, but it can be replaced with any way of loading data into the database.

Notice that *django_db_setup* is in the argument list. This may look odd at first, but it will make sure that the original pytest-django fixture is used to create the test database. When `call_command` is invoked, the test database is already prepared and configured.

Put this in `conftest.py`:

```
import pytest

from django.core.management import call_command

@pytest.fixture(scope='session')
def django_db_setup(django_db_setup, django_db_blocker):
    with django_db_blocker.unblock():
        call_command('loaddata', 'your_data_fixture.json')
```

Use the same database for all xdist processes

By default, each xdist process gets its own database to run tests on. This is needed to have transactional tests that does not interfere with eachother.

If you instead want your tests to use the same database, override the *django_db_modify_db_settings* to not do anything. Put this in `conftest.py`:

```
import pytest


@pytest.fixture(scope='session')
def django_db_modify_db_settings():
    pass
```

Randomize database sequences

You can customize the test database after it has been created by extending the *django_db_setup* fixture. This example shows how to give a PostgreSQL sequence a random starting value. This can be used to detect and prevent primary key id's from being hard-coded in tests.

Put this in `conftest.py`:

```
import random
import pytest
from django.db import connection


@pytest.fixture(scope='session')
def django_db_setup(django_db_setup, django_db_blocker):
    with django_db_blocker.unblock():
        cur = connection.cursor()
        cur.execute('ALTER SEQUENCE app_model_id_seq RESTART WITH %s;',
                    [random.randint(10000, 20000)])
```

Create the test database from a custom SQL script

You can replace the *django_db_setup* fixture and run any code in its place. This includes creating your database by hand by running a SQL script directly. This example shows how sqlite3's executescript method. In more a more general use cases you probably want to load the SQL statements from a file or invoke the `psql` or the `mysql` command line tool.

Put this in `conftest.py`:

```
import pytest
from django.db import connection


@pytest.fixture(scope='session')
def django_db_setup(django_db_blocker):
    with django_db_blocker.unblock():
        with connection.cursor() as c:
            c.executescript('''
            DROP TABLE IF EXISTS theapp_item;
            CREATE TABLE theapp_item (id, name);
            INSERT INTO theapp_item (name) VALUES ('created from a sql script');
            ''')
```

Use a read only database

You can replace the ordinary *django_db_setup* to completely avoid database creation/migrations. If you have no need for rollbacks or truncating tables, you can simply avoid blocking the database and use it directly. When using this method you must ensure that your tests do not change the database state.

Put this in `conftest.py`:

```
import pytest
```

(continues on next page)

(continued from previous page)

```
@pytest.fixture(scope='session')
def django_db_setup():
    """Avoid creating/setting up the test database"""
    pass


@pytest.fixture
def db_access_without_rollback_and_truncate(request, django_db_setup, django_db_
↪blocker):
    django_db_blocker.unblock()
    request.addfinalizer(django_db_blocker.restore)
```

4.6 Django helpers

4.6.1 Markers

`pytest-django` registers and uses markers. See the pytest documentation on what marks are and for notes on using them.

`pytest.mark.django_db` - request database access

This is used to mark a test function as requiring the database. It will ensure the database is set up correctly for the test. Each test will run in its own transaction which will be rolled back at the end of the test. This behavior is the same as Django's standard django.test.TestCase class.

In order for a test to have access to the database it must either be marked using the `django_db` mark or request one of the `db`, `transactional_db` or `django_db_reset_sequences` fixtures. Otherwise the test will fail when trying to access the database.

type transaction bool

param transaction The `transaction` argument will allow the test to use real transactions. With `transaction=False` (the default when not specified), transaction operations are noops during the test. This is the same behavior that django.test.TestCase uses. When `transaction=True`, the behavior will be the same as django.test.TransactionTestCase

type reset_sequences bool

param reset_sequences The `reset_sequences` argument will ask to reset auto increment sequence values (e.g. primary keys) before running the test. Defaults to `False`. Must be used together with `transaction=True` to have an effect. Please be aware that not all databases support this feature. For details see `django.test.TransactionTestCase.reset_sequences`.

Note: If you want access to the Django database *inside a fixture* this marker will not help even if the function requesting your fixture has this marker applied. To access the database in a fixture, the fixture itself will have to request the `db`, `transactional_db` or `django_db_reset_sequences` fixture. See below for a description of them.

Note: Automatic usage with `django.test.TestCase`.

Test classes that subclass django.test.TestCase will have access to the database always to make them compatible with existing Django tests. Test classes that subclass Python's `unittest.TestCase` need to have the marker applied in order to access the database.

`pytest.mark.urls` - override the urlconf

`pytest.mark.`**`urls`**(*urls*)

Specify a different `settings.ROOT_URLCONF` module for the marked tests.

Parameters **`urls`** (*str*) – The urlconf module to use for the test, e.g. `myapp.test_urls`. This is similar to Django's `TestCase.urls` attribute.

Example usage:

```
@pytest.mark.urls('myapp.test_urls')
def test_something(client):
    assert 'Success!' in client.get('/some_url_defined_in_test_urls/').content
```

`pytest.mark.ignore_template_errors` - ignore invalid template variables

`pytest.mark.`**`ignore_template_errors`**()

Ignore errors when using the `--fail-on-template-vars` option, i.e. do not cause tests to fail if your templates contain invalid variables.

This marker sets the `string_if_invalid` template option, or the older `settings.TEMPLATE_STRING_IF_INVALID=None` (Django up to 1.10). See How invalid variables are handled.

Example usage:

```
@pytest.mark.ignore_template_errors
def test_something(client):
    client('some-url-with-invalid-template-vars')
```

4.6.2 Fixtures

pytest-django provides some pytest fixtures to provide dependencies for tests. More information on fixtures is available in the pytest documentation.

`rf` - `RequestFactory`

An instance of a django.test.RequestFactory

Example

```
from myapp.views import my_view

def test_details(rf):
    request = rf.get('/customer/details')
    response = my_view(request)
    assert response.status_code == 200
```

`client` - `django.test.Client`

An instance of a django.test.Client

Example

```
def test_with_client(client):
    response = client.get('/')
    assert response.content == 'Foobar'
```

To use *client* as an authenticated standard user, call its *login()* method before accessing a URL:

```
def test_with_authenticated_client(client, django_user_model):
    username = "user1"
    password = "bar"
    django_user_model.objects.create_user(username=username, password=password)
    client.login(username=username, password=password)
    response = client.get('/private')
    assert response.content == 'Protected Area'
```

`admin_client` - `django.test.Client` logged in as admin

An instance of a django.test.Client, logged in as an admin user.

Example

```
def test_an_admin_view(admin_client):
    response = admin_client.get('/admin/')
    assert response.status_code == 200
```

Using the *admin_client* fixture will cause the test to automatically be marked for database use (no need to specify the `django_db` mark).

`admin_user` - an admin user (superuser)

An instance of a superuser, with username "admin" and password "password" (in case there is no "admin" user yet).

Using the *admin_user* fixture will cause the test to automatically be marked for database use (no need to specify the `django_db` mark).

`django_user_model`

A shortcut to the User model configured for use by the current Django project (aka the model referenced by *settings.AUTH_USER_MODEL*). Use this fixture to make pluggable apps testable regardless what User model is configured in the containing Django project.

Example

```
def test_new_user(django_user_model):
    django_user_model.objects.create(username="someone", password="something")
```

django_username_field

This fixture extracts the field name used for the username on the user model, i.e. resolves to the current `settings.USERNAME_FIELD`. Use this fixture to make pluggable apps testable regardless what the username field is configured to be in the containing Django project.

db

This fixture will ensure the Django database is set up. Only required for fixtures that want to use the database themselves. A test function should normally use the `pytest.mark.django_db` mark to signal it needs the database.

transactional_db

This fixture can be used to request access to the database including transaction support. This is only required for fixtures which need database access themselves. A test function should normally use the `pytest.mark.django_db` mark with `transaction=True`.

django_db_reset_sequences

This fixture provides the same transactional database access as `transactional_db`, with additional support for reset of auto increment sequences (if your database supports it). This is only required for fixtures which need database access themselves. A test function should normally use the `pytest.mark.django_db` mark with `transaction=True` and `reset_sequences=True`.

live_server

This fixture runs a live Django server in a background thread. The server's URL can be retrieved using the `live_server.url` attribute or by requesting it's string value: `unicode(live_server)`. You can also directly concatenate a string to form a URL: `live_server + '/foo`.

Note: Combining database access fixtures.

When using multiple database fixtures together, only one of them is used. Their order of precedence is as follows (the last one wins):

- `db`
- `transactional_db`
- `django_db_reset_sequences`

In addition, using `live_server` will also trigger transactional database access, if not specified.

---

settings

This fixture will provide a handle on the Django settings module, and automatically revert any changes made to the settings (modifications, additions and deletions).

Example

```
def test_with_specific_settings(settings):
    settings.USE_TZ = True
    assert settings.USE_TZ
```

django_assert_num_queries

This fixture allows to check for an expected number of DB queries.

It wraps *django.test.utils.CaptureQueriesContext*. A non-default DB connection can be passed in using the *connection* keyword argument, and it will yield the wrapped CaptureQueriesContext instance.

Example

```
def test_queries(django_assert_num_queries):
    with django_assert_num_queries(3) as captured:
        Item.objects.create('foo')
        Item.objects.create('bar')
        Item.objects.create('baz')

    assert 'foo' in captured.captured_queries[0]['sql']
```

django_assert_max_num_queries

This fixture allows to check for an expected maximum number of DB queries.

It is a specialized version of *django_assert_num_queries*.

Example

```
def test_max_queries(django_assert_max_num_queries):
    with django_assert_max_num_queries(3):
        Item.objects.create('foo')
        Item.objects.create('bar')
```

mailoutbox

A clean email outbox to which Django-generated emails are sent.

Example

```
from django.core import mail

def test_mail(mailoutbox):
    mail.send_mail('subject', 'body', 'from@example.com', ['to@example.com'])
    assert len(mailoutbox) == 1
    m = mailoutbox[0]
```

(continues on next page)

(continued from previous page)

```
    assert m.subject == 'subject'
    assert m.body == 'body'
    assert m.from_email == 'from@example.com'
    assert list(m.to) == ['to@example.com']
```

This uses the `django_mail_patch_dns` fixture, which patches `DNS_NAME` used by `django.core.mail` with the value from the `django_mail_dnsname` fixture, which defaults to "fake-tests.example.com".

4.6.3 Automatic cleanup

pytest-django provides some functionality to assure a clean and consistent environment during tests.

Clearing of site cache

If `django.contrib.sites` is in your INSTALLED_APPS, Site cache will be cleared for each test to avoid hitting the cache and causing the wrong Site object to be returned by `Site.objects.get_current()`.

Clearing of mail.outbox

`mail.outbox` will be cleared for each pytest, to give each new test an empty mailbox to work with. However, it's more "pytestic" to use the `mailoutbox` fixture described above than to access `mail.outbox`.

4.7 FAQ

4.7.1 I see an error saying "could not import myproject.settings"

pytest-django tries to automatically add your project to the Python path by looking for a `manage.py` file and adding its path to the Python path.

If this for some reason fails for you, you have to manage your Python paths explicitly. See the documentation on *Managing the Python path explicitly* for more information.

4.7.2 How can I make sure that all my tests run with a specific locale?

Create a pytest fixture that is automatically run before each test case. To run all tests with the english locale, put the following code in your project's conftest.py file:

```
from django.utils.translation import activate

@pytest.fixture(autouse=True)
def set_default_language():
    activate('en')
```

4.7.3 My tests are not being found. Why?

By default, pytest looks for tests in files named `test_*.py` (note that this is not the same as `test*.py`) and `*_test.py`. If you have your tests in files with other names, they will not be collected. Note that Django's

`startapp` manage command creates an `app_dir/tests.py` file. Also, it is common to put tests under `app_dir/tests/views.py`, etc.

To find those tests, create a `pytest.ini` file in your project root and add an appropriate `python_files` line to it:

```
[pytest]
python_files = tests.py test_*.py *_tests.py
```

See the related pytest docs for more details.

When debugging test collection problems, the `--collectonly` flag and `-rs` (report skipped tests) can be helpful.

4.7.4 Does pytest-django work with the pytest-xdist plugin?

Yes. pytest-django supports running tests in parallel with pytest-xdist. Each process created by xdist gets its own separate database that is used for the tests. This ensures that each test can run independently, regardless of whether transactions are tested or not.

4.7.5 How can I use `manage.py test` with pytest-django?

pytest-django is designed to work with the `pytest` command, but if you really need integration with `manage.py test`, you can create a simple test runner like this:

```
class PytestTestRunner(object):
    """Runs pytest to discover and run tests."""

    def __init__(self, verbosity=1, failfast=False, keepdb=False, **kwargs):
        self.verbosity = verbosity
        self.failfast = failfast
        self.keepdb = keepdb

    def run_tests(self, test_labels):
        """Run pytest and return the exitcode.

        It translates some of Django's test command option to pytest's.
        """
        import pytest

        argv = []
        if self.verbosity == 0:
            argv.append('--quiet')
        if self.verbosity == 2:
            argv.append('--verbose')
        if self.verbosity == 3:
            argv.append('-vv')
        if self.failfast:
            argv.append('--exitfirst')
        if self.keepdb:
            argv.append('--reuse-db')

        argv.extend(test_labels)
        return pytest.main(argv)
```

Add the path to this class in your Django settings:

```
TEST_RUNNER = 'my_project.runner.PytestTestRunner'
```

Usage:

```
./manage.py test <django args> -- <pytest args>
```

Note: the pytest-django command line options `--ds` and `--dc` are not compatible with this approach, you need to use the standard Django methods of setting the `DJANGO_SETTINGS_MODULE`/`DJANGO_CONFIGURATION` environmental variables or the `--settings` command line option.

4.7.6 How can I give database access to all my tests without the *django_db* marker?

Create an autouse fixture and put it in `conftest.py` in your project root:

```
@pytest.fixture(autouse=True)
def enable_db_access_for_all_tests(db):
    pass
```

4.7.7 How/where can I get help with pytest/pytest-django?

Usage questions can be asked on StackOverflow with the pytest tag.

If you think you've found a bug or something that is wrong in the documentation, feel free to open an issue on the GitHub project for pytest-django.

Direct help can be found in the #pylib IRC channel on irc.freenode.org.

4.8 Contributing to pytest-django

Like every open-source project, pytest-django is always looking for motivated individuals to contribute to its source code. However, to ensure the highest code quality and keep the repository nice and tidy, everybody has to follow a few rules (nothing major, I promise :))

4.8.1 Community

The fastest way to get feedback on contributions/bugs is usually to open an issue in the issue tracker.

Discussions also happen via IRC in #pylib on irc.freenode.org. You may also be interested in following @andreas-pelme on Twitter.

4.8.2 In a nutshell

Here's what the contribution process looks like, in a bullet-points fashion:

1. pytest-django is hosted on GitHub, at https://github.com/pytest-dev/pytest-django
2. The best method to contribute back is to create an account there and fork the project. You can use this fork as if it was your own project, and should push your changes to it.
3. When you feel your code is good enough for inclusion, "send us a pull request", by using the nice GitHub web interface.

4.8.3 Contributing Code

Getting the source code

- Code will be reviewed and tested by at least one core developer, preferably by several. Other community members are welcome to give feedback.
- Code *must* be tested. Your pull request should include unit-tests (that cover the piece of code you're submitting, obviously).
- Documentation should reflect your changes if relevant. There is nothing worse than invalid documentation.
- Usually, if unit tests are written, pass, and your change is relevant, then your pull request will be merged.

Since we're hosted on GitHub, pytest-django uses git as a version control system.

The GitHub help is very well written and will get you started on using git and GitHub in a jiffy. It is an invaluable resource for newbies and oldtimers alike.

Syntax and conventions

We try to conform to PEP8 as much as possible. A few highlights:

- Indentation should be exactly 4 spaces. Not 2, not 6, not 8. **4**. Also, tabs are evil.
- We try (loosely) to keep the line length at 79 characters. Generally the rule is "it should look good in a terminal-based editor" (eg vim), but we try not be [Godwin's law] about it.

Process

This is how you fix a bug or add a feature:

1. fork the repository on GitHub.
2. Checkout your fork.
3. Hack hack hack, test test test, commit commit commit, test again.
4. Push to your fork.
5. Open a pull request.

Tests

Having a wide and comprehensive library of unit-tests and integration tests is of exceeding importance. Contributing tests is widely regarded as a very prestigious contribution (you're making everybody's future work much easier by doing so). Good karma for you. Cookie points. Maybe even a beer if we meet in person :)

Generally tests should be:

- Unitary (as much as possible). I.E. should test as much as possible only on one function/method/class. That's the very definition of unit tests. Integration tests are also interesting obviously, but require more time to maintain since they have a higher probability of breaking.
- Short running. No hard numbers here, but if your one test doubles the time it takes for everybody to run them, it's probably an indication that you're doing it wrong.

In a similar way to code, pull requests will be reviewed before pulling (obviously), and we encourage discussion via code review (everybody learns something this way) or in the IRC channel.

Running the tests

There is a Makefile in the repository which aids in setting up a virtualenv and running the tests:

```
$ make test
```

You can manually create the virtualenv using:

```
$ make testenv
```

This will install a virtualenv with pytest and the latest stable version of Django. The virtualenv can then be activated with:

```
$ source bin/activate
```

Then, simply invoke pytest to run the test suite:

```
$ pytest --ds=pytest_django_test.settings_sqlite
```

tox can be used to run the test suite under different configurations by invoking:

```
$ tox
```

There is a huge number of unique test configurations (98 at the time of writing), running them all will take a long time. All valid configurations can be found in *tox.ini*. To test against a few of them, invoke tox with the *-e* flag:

```
$ tox -e py36-dj111-postgres,py27-dj111-mysql_innodb
```

This will run the tests on Python 3.6/Django 1.11/PostgeSQL and Python 2.7/Django 1.11/MySQL.

Measuring test coverage

Some of the tests are executed in subprocesses. Because of that regular coverage measurements (using pytest-cov plugin) are not reliable.

If you want to measure coverage you'll need to create .pth file as described in subprocess section of coverage documentation. If you're using `setup.py develop` you should uninstall pytest_django (using pip) for the time of measuring coverage.

You'll also need mysql and postgres databases. There are predefined settings for each database in the tests directory. You may want to modify these files but please don't include them in your pull requests.

After this short initial setup you're ready to run tests:

```
$ COVERAGE_PROCESS_START=`pwd`/.coveragerc COVERAGE_FILE=`pwd`/.coverage␣
↪PYTHONPATH=`pwd` pytest --ds=pytest_django_test.settings_postgres
```

You should repeat the above step for sqlite and mysql before the next step. This step will create a lot of `.coverage` files with additional suffixes for every process.

The final step is to combine all the files created by different processes and generate the html coverage report:

```
$ coverage combine
$ coverage html
```

Your coverage report is now ready in the `htmlcov` directory.

Continuous integration

Travis is used to automatically run all tests against all supported versions of Python, Django and different database backends.

The pytest-django Travis page shows the latest test run. Travis will automatically pick up pull requests, test them and report the result directly in the pull request.

4.8.4 Contributing Documentation

Perhaps considered "boring" by hard-core coders, documentation is sometimes even more important than code! This is what brings fresh blood to a project, and serves as a reference for oldtimers. On top of this, documentation is the one area where less technical people can help most - you just need to write a semi-decent English. People need to understand you. We don't care about style or correctness.

Documentation should be:

- We use Sphinx/restructuredText. So obviously this is the format you should use :) File extensions should be .rst.
- Written in English. We can discuss how it would bring more people to the project to have a Klingon translation or anything, but that's a problem we will ask ourselves when we already have a good documentation in English.
- Accessible. You should assume the reader to be moderately familiar with Python and Django, but not anything else. Link to documentation of libraries you use, for example, even if they are "obvious" to you (South is the first example that comes to mind - it's obvious to any Django programmer, but not to any newbie at all). A brief description of what it does is also welcome.

Pulling of documentation is pretty fast and painless. Usually somebody goes over your text and merges it, since there are no "breaks" and that GitHub parses rst files automagically it's really convenient to work with.

Also, contributing to the documentation will earn you great respect from the core developers. You get good karma just like a test contributor, but you get double cookie points. Seriously. You rock.

Note: This very document is based on the contributing docs of the django CMS project. Many thanks for allowing us to steal it!

4.9 Changelog

4.9.1 3.4.3 (2018-09-16)

Bugfixes

- Fix OSError with arguments containing `::` on Windows (#641).

4.9.2 3.4.2 (2018-08-20)

Bugfixes

- Changed dependency for pathlib to pathlib2 (#636).
- Fixed code for inserting the project to sys.path with pathlib to use an absolute path, regression in 3.4.0 (#637, #638).

4.9.3 3.4.0 (2018-08-16)

Features

- Added new fixture *django_assert_max_num_queries* (#547).
- Added support for `connection` and returning the wrapped context manager with *django_assert_num_queries* (#547).
- Added support for resetting sequences via *django_db_reset_sequences* (#619).

Bugfixes

- Made sure to not call django.setup() multiple times (#629, #531).

Compatibility

- Removed py dependency, use pathlib instead (#631).

4.9.4 3.3.3 (2018-07-26)

Bug fixes

- Fixed registration of *ignore_template_errors()* marker, which is required with `pytest --strict` (#609).
- Fixed another regression with unittest (#624, #625).

Docs

- Use sphinx_rtf_theme (#621).
- Minor fixes.

4.9.5 3.3.2 (2018-06-21)

Bug fixes

- Fixed test for classmethod with Django TestCases again (#618, introduced in #598 (3.3.0)).

Compatibility

- Support Django 2.1 (no changes necessary) (#614).

4.9.6 3.3.0 (2018-06-15)

Features

- Added new fixtures `django_mail_dnsname` and `django_mail_patch_dns`, used by `mailoutbox` to monkeypatch the `DNS_NAME` used in `django.core.mail` to improve performance and reproducibility.

Bug fixes

- Fixed test for classmethod with Django TestCases (#597, #598).
- Fixed RemovedInPytest4Warning: MarkInfo objects are deprecated (#596, #603)
- Fixed scope of overridden settings with live_server fixture: previously they were visible to following tests (#612).

Compatibility

- The required *pytest* version changed from >=2.9 to >=3.6.

4.9.7 3.2.1

- Fixed automatic deployment to PyPI.

4.9.8 3.2.0

Features

- Added new fixture *django_assert_num_queries* for testing the number of database queries (#387).
- *–fail-on-template-vars* has been improved and should now return full/absolute path (#470).
- Support for setting the live server port (#500).
- unittest: help with setUpClass not being a classmethod (#544).

Bug fixes

- Fix –reuse-db and –create-db not working together (#411).
- Numerous fixes in the documentation. These should not go unnoticed

Compatibility

- Support for Django 2.0 has been added.
- Support for Django before 1.8 has been dropped.

4.9.9 3.1.2

Bug fixes

- Auto clearing of `mail.outbox` has been re-introduced to not break functionality in 3.x.x release. This means that Compatibility issues mentioned in the 3.1.0 release are no longer present. Related issue: pytest-django issue

4.9.10 3.1.1

Bug fixes

- Workaround *–pdb* interaction with Django TestCase. The issue is caused by Django TestCase not implementing TestCase.debug() properly but was brought to attention with recent changes in pytest 3.0.2. Related issues: pytest issue, Django issue

4.9.11 3.1.0

Features

- Added new function scoped fixture `mailoutbox` that gives access to djangos `mail.outbox`. The will clean/empty the `mail.outbox` to assure that no old mails are still in the outbox.
- If `django.contrib.sites` is in your INSTALLED_APPS, Site cache will be cleared for each test to avoid hitting the cache and cause wrong Site object to be returned by `Site.objects.get_current()`.

Compatibility

- IMPORTANT: the internal autouse fixture _django_clear_outbox has been removed. If you have relied on this to get an empty outbox for your test, you should change tests to use the `mailoutbox` fixture instead. See documentation of `mailoutbox` fixture for usage. If you try to access mail.outbox directly, AssertionError will be raised. If you previously relied on the old behaviour and do not want to change your tests, put this in your project conftest.py:

```
@pytest.fixture(autouse=True)
def clear_outbox():
    from django.core import mail
    mail.outbox = []
```

4.9.12 3.0.0

Bug fixes

- Fix error when Django happens to be imported before pytest-django runs. Thanks to Will Harris for the bug report.

Features

- Added a new option `--migrations` to negate a default usage of `--nomigrations`.
- The previously internal pytest-django fixture that handles database creation and setup has been refactored, refined and made a public API.

  This opens up more flexibility and advanced use cases to configure the test database in new ways.

  See *Advanced database configuration* for more information on the new fixtures and example use cases.

Compatibility

- Official for the pytest 3.0.0 (2.9.2 release should work too, though). The documentation is updated to mention `pytest` instead of `py.test`.
- Django versions 1.4, 1.5 and 1.6 is no longer supported. The supported versions are now 1.7 and forward. Django master is supported as of 2016-08-21.
- pytest-django no longer supports Python 2.6.
- Specifying the `DJANGO_TEST_LIVE_SERVER_ADDRESS` environment variable is no longer supported. Use `DJANGO_LIVE_TEST_SERVER_ADDRESS` instead.
- Ensuring accidental database access is now stricter than before. Previously database access was prevented on the cursor level. To be safer and prevent more cases, it is now prevented at the connection level. If you previously had tests which interacted with the databases without a database cursor, you will need to mark them with the `pytest.mark.django_db` marker or request the `db` fixture.
- The previously undocumented internal fixtures `_django_db_setup`, `_django_cursor_wrapper` have been removed in favour of the new public fixtures. If you previously relied on these internal fixtures, you must update your code. See *Advanced database configuration* for more information on the new fixtures and example use cases.

4.9.13 2.9.1

Bug fixes

- Fix regression introduced in 2.9.0 that caused TestCase subclasses with mixins to cause errors. Thanks MikeVL for the bug report.

4.9.14 2.9.0

2.9.0 focus on compatibility with Django 1.9 and master as well as pytest 2.8.1 and Python 3.5

Features

- `--fail-on-template-vars` - fail tests for invalid variables in templates. Thanks to Johannes Hoppe for idea and implementation. Thanks Daniel Hahler for review and feedback.

Bug fixes

- Ensure urlconf is properly reset when using @pytest.mark.urls. Thanks to Sarah Bird, David Szotten, Daniel Hahler and Yannick PÉROUX for patch and discussions. Fixes issue #183.
- Call `setUpClass()` in Django `TestCase` properly when test class is inherited multiple places. Thanks to Benedikt Forchhammer for report and initial test case. Fixes issue #265.

Compatibility

- Settings defined in `pytest.ini`/`tox.ini`/`setup.cfg` used to override `DJANGO_SETTINGS_MODULE` defined in the environment. Previously the order was undocumented. Now, instead the settings from the environment will be used instead. If you previously relied on overriding the environment variable, you can instead

specify `addopts = --ds=yourtestsettings` in the ini-file which will use the test settings. See PR #199.

- Support for Django 1.9.
- Support for Django master (to be 1.10) as of 2015-10-06.
- Drop support for Django 1.3. While pytest-django supports a wide range of Django versions, extended for Django 1.3 was dropped in february 2013.

4.9.15 2.8.0

Features

- pytest's verbosity is being used for Django's code to setup/teardown the test database (#172).
- Added a new option *–nomigrations* to avoid running Django 1.7+ migrations when constructing the test database. Huge thanks to Renan Ivo for complete patch, tests and documentation.

Bug fixes

- Fixed compatibility issues related to Django 1.8's *setUpClass/setUpTestData*. Django 1.8 is now a fully supported version. Django master as of 2014-01-18 (the Django 1.9 branch) is also supported.

4.9.16 2.7.0

Features

- New fixtures: `admin_user`, `django_user_model` and `django_username_field` (#109).
- Automatic discovery of Django projects to make it easier for new users. This change is slightly backward incompatible, if you encounter problems with it, the old behaviour can be restored by adding this to `pytest.ini`, `setup.cfg` or `tox.ini`:

```
[pytest]
django_find_project = false
```

Please see the *Managing the Python path* section for more information.

Bugfixes

- Fix interaction between `db` and `transaction_db` fixtures (#126).
- Fix admin client with custom user models (#124). Big thanks to Benjamin Hedrich and Dmitry Dygalo for patch and tests.
- Fix usage of South migrations, which were unconditionally disabled previously (#22).
- Fixed #119, #134: Call `django.setup()` in Django >=1.7 directly after settings is loaded to ensure proper loading of Django applications. Thanks to Ionel Cristian Mărieș, Daniel Hahler, Tymur Maryokhin, Kirill SIbirev, Paul Collins, Aymeric Augustin, Jannis Leidel, Baptiste Mispelon and Anatoly Bubenkoff for report, discussion and feedback.
- *The 'live_server'* fixture can now serve static files also for Django>=1.7 if the `django.contrib.staticfiles` app is installed. (#140).

- DJANGO_LIVE_TEST_SERVER_ADDRESS environment variable is read instead of DJANGO_TEST_LIVE_SERVER_ADDRESS. (#140)

4.9.17 2.6.2

- Fixed a bug that caused doctests to runs. Thanks to @jjmurre for the patch
- Fixed issue #88 - make sure to use SQLite in memory database when running with pytest-xdist.

4.9.18 2.6.1

This is a bugfix/support release with no new features:

- Added support for Django 1.7 beta and Django master as of 2014-04-16. pytest-django is now automatically tested against the latest git master version of Django.
- Support for MySQL with MyISAM tables. Thanks to Zach Kanzler and Julen Ruiz Aizpuru for fixing this. This fixes issue #8 #64.

4.9.19 2.6.0

- Experimental support for Django 1.7 / Django master as of 2014-01-19.

  pytest-django is now automatically tested against the latest git version of Django. The support is experimental since Django 1.7 is not yet released, but the goal is to always be up to date with the latest Django master

4.9.20 2.5.1

Invalid release accidentally pushed to PyPI (identical to 2.6.1). Should not be used - use 2.6.1 or newer to avoid confusion.

4.9.21 2.5.0

- Python 2.5 compatibility dropped. py.test 2.5 dropped support for Python 2.5, therefore it will be hard to properly support in pytest-django. The same strategy as for pytest itself is used: No code will be changed to prevent Python 2.5 from working, but it will not be actively tested.
- pytest-xdist support: it is now possible to run tests in parallel. Just use pytest-xdist as normal (pass -n to py.test). One database will be created for each subprocess so that tests run independent from each other.

4.9.22 2.4.0

- Support for py.test 2.4 pytest_load_initial_conftests. This makes it possible to import Django models in project conftest.py files, since pytest-django will be initialized before the conftest.py is loaded.

4.9.23 2.3.1

- Support for Django 1.5 custom user models, thanks to Leonardo Santagada.

4.9.24 2.3.0

- Support for configuring settings via django-configurations. Big thanks to Donald Stufft for this feature!

4.9.25 2.2.1

- Fixed an issue with the settings fixture when used in combination with django-appconf. It now uses pytest's monkeypatch internally and should be more robust.

4.9.26 2.2.0

- Python 3 support. pytest-django now supports Python 3.2 and 3.3 in addition to 2.5-2.7. Big thanks to Rafal Stozek for making this happen!

4.9.27 2.1.0

- Django 1.5 support. pytest-django is now tested against 1.5 for Python 2.6-2.7. This is the first step towards Python 3 support.

4.9.28 2.0.1

- Fixed #24/#25: Make it possible to configure Django via `django.conf.settings.configure()`.
- Fixed #26: Don't set DEBUG_PROPAGATE_EXCEPTIONS = True for test runs. Django does not change this setting in the default test runner, so pytest-django should not do it either.

4.9.29 2.0.0

This release is *backward incompatible*. The biggest change is the need to add the `pytest.mark.django_db` to tests which require database access.

Finding such tests is generally very easy: just run your test suite, the tests which need database access will fail. Add `pytestmark = pytest.mark.django_db` to the module/class or decorate them with `@pytest.mark.django_db`.

Most of the internals have been rewritten, exploiting py.test's new fixtures API. This release would not be possible without Floris Bruynooghe who did the port to the new fixture API and fixed a number of bugs.

The tests for pytest-django itself has been greatly improved, paving the way for easier additions of new and exciting features in the future!

- Semantic version numbers will now be used for releases, see http://semver.org/.
- Do not allow database access in tests by default. Introduce `pytest.mark.django_db` to enable database access.
- Large parts re-written using py.test's 2.3 fixtures API (issue #9).
  - Fixes issue #17: Database changes made in fixtures or funcargs will now be reverted as well.
  - Fixes issue 21: Database teardown errors are no longer hidden.
  - Fixes issue 16: Database setup and teardown for non-TestCase classes works correctly.
- `pytest.urls()` is replaced by the standard marking API and is now used as `pytest.mark.urls()`

- Make the plugin behave gracefully without DJANGO_SETTINGS_MODULE specified. `py.test` will still work and tests needing django features will skip (issue #3).
- Allow specifying of `DJANGO_SETTINGS_MODULE` on the command line (`--ds=settings`) and py.test ini configuration file as well as the environment variable (issue #3).
- Deprecate the `transaction_test_case` decorator, this is now integrated with the `django_db` mark.

4.9.30 1.4

- Removed undocumented pytest.load_fixture: If you need this feature, just use `django.management.call_command('loaddata', 'foo.json')` instead.
- Fixed issue with RequestFactory in Django 1.3.
- Fixed issue with RequestFactory in Django 1.3.

4.9.31 1.3

- Added `--reuse-db` and `--create-db` to allow database re-use. Many thanks to django-nose for code and inspiration for this feature.

4.9.32 1.2.2

- Fixed Django 1.3 compatibility.

4.9.33 1.2.1

- Disable database access and raise errors when using –no-db and accessing the database by accident.

4.9.34 1.2

- Added the `--no-db` command line option.

4.9.35 1.1.1

- Flush tables after each test run with transaction_test_case instead of before.

4.9.36 1.1

- The initial release of this fork from Ben Firshman original project
- Added documentation
- Uploaded to PyPI for easy installation
- Added the `transaction_test_case` decorator for tests that needs real transactions
- Added initial implementation for live server support via a funcarg (no docs yet, it might change!)

CHAPTER 5

Indices and Tables

- genindex
- modindex

Index