Advanced testing topics

The request factory

`class RequestFactory` [source]

The `RequestFactory` shares the same API as the test client. However, instead of behaving like a browser, the RequestFactory provides a way to generate a request instance that can be used as the first argument to any view. This means you can test a view function the same way as you would test any other function – as a black box, with exactly known inputs, testing for specific outputs.

The API for the `RequestFactory` is a slightly restricted subset of the test client API:

- It only has access to the HTTP methods `get()`, `post()`, `put()`, `delete()`, `head()`, `options()`, and `trace()`.
- These methods accept all the same arguments *except* for `follow`. Since this is just a factory for producing requests, it's up to you to handle the response.
- It does not support middleware. Session and authentication attributes must be supplied by the test itself if required for the view to function properly.

Changed in Django 5.1:
The `query_params` parameter was added.

Example

The following is a unit test using the request factory:

```
from django.contrib.auth.models import AnonymousUser, User
from django.test import RequestFactory, TestCase

from .views import MyView, my_view


class SimpleTest(TestCase):
    def setUp(self):
        # Every test needs access to the request factory.
        self.factory = RequestFactory()
        self.user = User.objects.create_user(
            username="jacob", email="jacob@…", password="top_secret"
        )

    def test_details(self):
        # Create an instance of a GET request.
        request = self.factory.get("/customer/details")

        # Recall that middleware are not supported. You can simulate a
        # logged-in user by setting request.user manually.
        request.user = self.user

        # Or you can simulate an anonymous user by setting request.user to
        # an AnonymousUser instance.
        request.user = AnonymousUser()

        # Test my_view() as if it were deployed at /customer/details
        response = my_view(request)
        # Use this syntax for class-based views.
        response = MyView.as_view()(request)
        self.assertEqual(response.status_code, 200)
```

AsyncRequestFactory

`class AsyncRequestFactory` [source]

`RequestFactory` creates WSGI-like requests. If you want to create ASGI-like requests, including having a correct ASGI `scope`, you can instead use `django.test.AsyncRequestFactory`.

This class is directly API-compatible with `RequestFactory`, with the only difference being that it returns `ASGIRequest` instances rather than `WSGIRequest` instances. All of its methods are still synchronous callables.

Arbitrary keyword arguments in `defaults` are added directly into the ASGI scope.

Changed in Django 5.1:

The `query_params` parameter was added.

Testing class-based views

In order to test class-based views outside of the request/response cycle you must ensure that they are configured correctly, by calling `setup()` after instantiation.

For example, assuming the following class-based view:

`views.py`

```
from django.views.generic import TemplateView


class HomeView(TemplateView):
    template_name = "myapp/home.html"

    def get_context_data(self, **kwargs):
        kwargs["environment"] = "Production"
        return super().get_context_data(**kwargs)
```

You may directly test the `get_context_data()` method by first instantiating the view, then passing a `request` to `setup()`, before proceeding with your test's code:

`tests.py`

```
from django.test import RequestFactory, TestCase
from .views import HomeView


class HomePageTest(TestCase):
    def test_environment_set_in_context(self):
        request = RequestFactory().get("/")
        view = HomeView()
        view.setup(request)

        context = view.get_context_data()
        self.assertIn("environment", context)
```

Tests and multiple host names

The `ALLOWED_HOSTS` setting is validated when running tests. This allows the test client to differentiate between internal and external URLs.

Projects that support multitenancy or otherwise alter business logic based on the request's host and use custom host names in tests must include those hosts in `ALLOWED_HOSTS`.

The first option to do so is to add the hosts to your settings file. For example, the test suite for docs.djangoproject.com includes the following:

```
from django.test import TestCase


class SearchFormTestCase(TestCase):
    def test_empty_get(self):
        response = self.client.get(
            "/en/dev/search/",
            headers={"host": "docs.djangoproject.dev:8000"},
        )
        self.assertEqual(response.status_code, 200)
```

and the settings file includes a list of the domains supported by the project:

```
ALLOWED_HOSTS = ["www.djangoproject.dev", "docs.djangoproject.dev", ...]
```

Another option is to add the required hosts to `ALLOWED_HOSTS` using `override_settings()` or `modify_settings()`. This option may be preferable in standalone apps that can't package their own settings file or for projects where the list of domains is not static (e.g., subdomains for multitenancy). For example, you could write a test for the domain `http://otherserver/` as follows:

```
from django.test import TestCase, override_settings


class MultiDomainTestCase(TestCase):
    @override_settings(ALLOWED_HOSTS=["otherserver"])
    def test_other_domain(self):
        response = self.client.get("http://otherserver/foo/bar/")
```

Disabling `ALLOWED_HOSTS` checking (`ALLOWED_HOSTS = ['*']`) when running tests prevents the test client from raising a helpful error message if you follow a redirect to an external URL.

Tests and multiple databases

Testing primary/replica configurations

If you're testing a multiple database configuration with primary/replica (referred to as master/slave by some databases) replication, this strategy of creating test databases poses a problem. When the test databases are created, there won't be any replication, and as a result, data created on the primary won't be seen on the replica.

To compensate for this, Django allows you to define that a database is a *test mirror*. Consider the following (simplified) example database configuration:

```
DATABASES = {
    "default": {
        "ENGINE": "django.db.backends.mysql",
        "NAME": "myproject",
        "HOST": "dbprimary",
        # ... plus some other settings
    },
    "replica": {
        "ENGINE": "django.db.backends.mysql",
        "NAME": "myproject",
        "HOST": "dbreplica",
        "TEST": {
            "MIRROR": "default",
        },
        # ... plus some other settings
    },
}
```

In this setup, we have two database servers: `dbprimary` , described by the database alias `default` , and `dbreplica` described by the alias `replica` . As you might expect, `dbreplica` has been configured by the database administrator as a read replica of `dbprimary` , so in normal activity, any write to `default` will appear on `replica` .

If Django created two independent test databases, this would break any tests that expected replication to occur. However, the `replica` database has been configured as a test mirror (using the `MIRROR` test setting), indicating that under testing, `replica` should be treated as a mirror of `default` .

When the test environment is configured, a test version of `replica` will *not* be created. Instead the connection to `replica` will be redirected to point at `default` . As a result, writes to `default` will appear on `replica` – but because they are actually the same database, not because there is data replication between the two databases. As this depends on transactions, the tests must use `TransactionTestCase` instead of `TestCase`.

Controlling creation order for test databases

By default, Django will assume all databases depend on the `default` database and therefore always create the `default` database first. However, no guarantees are made on the creation order of any other databases in your test setup.

If your database configuration requires a specific creation order, you can specify the dependencies that exist using the `DEPENDENCIES` test setting. Consider the following (simplified) example database configuration:

```
DATABASES = {
    "default": {
        # ... db settings
        "TEST": {
            "DEPENDENCIES": ["diamonds"],
        },
    },
    "diamonds": {
        # ... db settings
        "TEST": {
```

```
            "DEPENDENCIES": [],
        },
    },
    "clubs": {
        # ... db settings
        "TEST": {
            "DEPENDENCIES": ["diamonds"],
        },
    },
    "spades": {
        # ... db settings
        "TEST": {
            "DEPENDENCIES": ["diamonds", "hearts"],
        },
    },
    "hearts": {
        # ... db settings
        "TEST": {
            "DEPENDENCIES": ["diamonds", "clubs"],
        },
    },
}
```

Under this configuration, the `diamonds` database will be created first, as it is the only database alias without dependencies. The `default` and `clubs` alias will be created next (although the order of creation of this pair is not guaranteed), then `hearts`, and finally `spades`.

If there are any circular dependencies in the `DEPENDENCIES` definition, an `ImproperlyConfigured` exception will be raised.

Advanced features of `TransactionTestCase`

`TransactionTestCase.available_apps`

> **Warning:** This attribute is a private API. It may be changed or removed without a deprecation period in the future, for instance to accommodate changes in application loading.
>
> It's used to optimize Django's own test suite, which contains hundreds of models but no relations between models in different applications.

By default, `available_apps` is set to `None`. After each test, Django calls `flush` to reset the database state. This empties all tables and emits the `post_migrate` signal, which recreates one content type and four permissions for each model. This operation gets expensive proportionally to the number of models.

Setting `available_apps` to a list of applications instructs Django to behave as if only the models from these applications were available. The behavior of `TransactionTestCase` changes as follows:

- `post_migrate` is fired before each test to create the content types and permissions for each model in available apps, in case they're missing.
- After each test, Django empties only tables corresponding to models in available apps. However, at the database level, truncation may cascade to related models in unavailable apps. Furthermore `post_migrate` isn't fired; it will be fired by the next `TransactionTestCase`, after the correct set of applications is selected.

Since the database isn't fully flushed, if a test creates instances of models not included in `available_apps`, they will leak and they may cause unrelated tests to fail. Be careful with tests that use sessions; the default session engine stores them in the database.

Since `post_migrate` isn't emitted after flushing the database, its state after a `TransactionTestCase` isn't the same as after a `TestCase`: it's missing the rows created by listeners to `post_migrate`. Considering the order in which tests are executed, this isn't an issue, provided either all `TransactionTestCase` in a given test suite declare `available_apps`, or none of them.

`available_apps` is mandatory in Django's own test suite.

`TransactionTestCase.reset_sequences`

Setting `reset_sequences = True` on a `TransactionTestCase` will make sure sequences are always reset before the test run:

```
class TestsThatDependsOnPrimaryKeySequences(TransactionTestCase):
    reset_sequences = True

    def test_animal_pk(self):
        lion = Animal.objects.create(name="lion", sound="roar")
        # lion.pk is guaranteed to always be 1
        self.assertEqual(lion.pk, 1)
```

Unless you are explicitly testing primary keys sequence numbers, it is recommended that you do not hard code primary key values in tests.

Using `reset_sequences = True` will slow down the test, since the primary key reset is a relatively expensive database operation.

Enforce running test classes sequentially

If you have test classes that cannot be run in parallel (e.g. because they share a common resource), you can use `django.test.testcases.SerializeMixin` to run them sequentially. This mixin uses a filesystem `lockfile`.

For example, you can use `__file__` to determine that all test classes in the same file that inherit from `SerializeMixin` will run sequentially:

```
import os

from django.test import TestCase
from django.test.testcases import SerializeMixin


class ImageTestCaseMixin(SerializeMixin):
    lockfile = __file__

    def setUp(self):
        self.filename = os.path.join(temp_storage_dir, "my_file.png")
        self.file = create_file(self.filename)


class RemoveImageTests(ImageTestCaseMixin, TestCase):
    def test_remove_image(self):
        os.remove(self.filename)
        self.assertFalse(os.path.exists(self.filename))


class ResizeImageTests(ImageTestCaseMixin, TestCase):
    def test_resize_image(self):
        resize_image(self.file, (48, 48))
        self.assertEqual(get_image_size(self.file), (48, 48))
```

Using the Django test runner to test reusable applications

If you are writing a reusable application you may want to use the Django test runner to run your own test suite and thus benefit from the Django testing infrastructure.

A common practice is a *tests* directory next to the application code, with the following structure:

```
runtests.py
polls/
    __init__.py
    models.py
    ...
tests/
    __init__.py
    models.py
    test_settings.py
    tests.py
```

Let's take a look inside a couple of those files:

`runtests.py`

```
#!/usr/bin/env python
import os
import sys

import django
from django.conf import settings
from django.test.utils import get_runner

if __name__ == "__main__":
    os.environ["DJANGO_SETTINGS_MODULE"] = "tests.test_settings"
    django.setup()
    TestRunner = get_runner(settings)
    test_runner = TestRunner()
    failures = test_runner.run_tests(["tests"])
    sys.exit(bool(failures))
```

This is the script that you invoke to run the test suite. It sets up the Django environment, creates the test database and runs the tests.

For the sake of clarity, this example contains only the bare minimum necessary to use the Django test runner. You may want to add command-line options for controlling verbosity, passing in specific test labels to run, etc.

`tests/test_settings.py`

```
SECRET_KEY = "fake-key"
INSTALLED_APPS = [
    "tests",
]
```

This file contains the Django settings required to run your app's tests.

Again, this is a minimal example; your tests may require additional settings to run.

Since the *tests* package is included in `INSTALLED_APPS` when running your tests, you can define test-only models in its `models.py` file.

Using different testing frameworks

Clearly, `unittest` is not the only Python testing framework. While Django doesn't provide explicit support for alternative frameworks, it does provide a way to invoke tests constructed for an alternative framework as if they were normal Django tests.

When you run `./manage.py test`, Django looks at the `TEST_RUNNER` setting to determine what to do. By default, `TEST_RUNNER` points to `'django.test.runner.DiscoverRunner'`. This class defines the default Django testing behavior. This behavior involves:

1. Performing global pre-test setup.
2. Looking for tests in any file below the current directory whose name matches the pattern `test*.py`.
3. Creating the test databases.
4. Running `migrate` to install models and initial data into the test databases.
5. Running the system checks.
6. Running the tests that were found.
7. Destroying the test databases.
8. Performing global post-test teardown.

If you define your own test runner class and point `TEST_RUNNER` at that class, Django will execute your test runner whenever you run `./manage.py test`. In this way, it is possible to use any test framework that can be executed from Python code, or to modify the Django test execution process to satisfy whatever testing requirements you may have.

Defining a test runner

A test runner is a class defining a `run_tests()` method. Django ships with a `DiscoverRunner` class that defines the default Django testing behavior. This class defines the `run_tests()` entry point, plus a selection of other methods that are used by `run_tests()` to set up, execute and tear down the test suite.

```
class DiscoverRunner(pattern='test*.py', top_level=None, verbosity=1, interactive=True, failfast=False, keepdb=False, reverse=False, debug_mode=False, debug_sql=False, parallel=0, tags=None, exclude_tags=None, test_name_patterns=None, pdb=False, buffer=False, enable_faulthandler=True, timing=True, shuffle=False, logger=None, durations=None, **kwargs)  [source]
```

`DiscoverRunner` will search for tests in any file matching `pattern`.

`top_level` can be used to specify the directory containing your top-level Python modules. Usually Django can figure this out automatically, so it's not necessary to specify this option. If specified, it should generally be the directory containing your `manage.py` file.

`verbosity` determines the amount of notification and debug information that will be printed to the console; `0` is no output, `1` is normal output, and `2` is verbose output.

If `interactive` is `True`, the test suite has permission to ask the user for instructions when the test suite is executed. An example of this behavior would be asking for permission to delete an existing test database. If `interactive` is `False`, the test suite must be able to run without any manual intervention.

If `failfast` is `True`, the test suite will stop running after the first test failure is detected.

If `keepdb` is `True`, the test suite will use the existing database, or create one if necessary. If `False`, a new database will be created, prompting the user to remove the existing one, if present.

If `reverse` is `True`, test cases will be executed in the opposite order. This could be useful to debug tests that aren't properly isolated and have side effects. Grouping by test class is preserved when using this option. This option can be used in conjunction with `--shuffle` to reverse the order for a particular random seed.

`debug_mode` specifies what the `DEBUG` setting should be set to prior to running tests.

`parallel` specifies the number of processes. If `parallel` is greater than `1`, the test suite will run in `parallel` processes. If there are fewer test case classes than configured processes, Django will reduce the number of processes accordingly. Each process gets its own database. This option requires the third-party `tblib` package to display trace-

backs correctly.

`tags` can be used to specify a set of tags for filtering tests. May be combined with `exclude_tags`.

`exclude_tags` can be used to specify a set of tags for excluding tests. May be combined with `tags`.

If `debug_sql` is `True`, failing test cases will output SQL queries logged to the django.db.backends logger as well as the traceback. If `verbosity` is `2`, then queries in all tests are output.

`test_name_patterns` can be used to specify a set of patterns for filtering test methods and classes by their names.

If `pdb` is `True`, a debugger (`pdb` or `ipdb`) will be spawned at each test error or failure.

If `buffer` is `True`, outputs from passing tests will be discarded.

If `enable_faulthandler` is `True`, `faulthandler` will be enabled.

If `timing` is `True`, test timings, including database setup and total run time, will be shown.

If `shuffle` is an integer, test cases will be shuffled in a random order prior to execution, using the integer as a random seed. If `shuffle` is `None`, the seed will be generated randomly. In both cases, the seed will be logged and set to `self.shuffle_seed` prior to running tests. This option can be used to help detect tests that aren't properly isolated. Grouping by test class is preserved when using this option.

`logger` can be used to pass a Python Logger object. If provided, the logger will be used to log messages instead of printing to the console. The logger object will respect its logging level rather than the `verbosity`.

`durations` will show a list of the N slowest test cases. Setting this option to `0` will result in the duration for all tests being shown. Requires Python 3.12+.

Django may, from time to time, extend the capabilities of the test runner by adding new arguments. The `**kwargs` declaration allows for this expansion. If you subclass `DiscoverRunner` or write your own test runner, ensure it accepts `**kwargs`.

Your test runner may also define additional command-line options. Create or override an `add_arguments(cls, parser)` class method and add custom arguments by calling `parser.add_argument()` inside the method, so that the `test` command will be able to use those arguments.

> **New in Django 5.0:**
> The `durations` argument was added.

Attributes

`DiscoverRunner.test_suite`

The class used to build the test suite. By default it is set to `unittest.TestSuite`. This can be overridden if you wish to implement different logic for collecting tests.

`DiscoverRunner.test_runner`

This is the class of the low-level test runner which is used to execute the individual tests and format the results. By default it is set to `unittest.TextTestRunner`. Despite the unfortunate similarity in naming conventions, this is not the same type of class as `DiscoverRunner`, which covers a broader set of responsibilities. You can override this attribute to modify the way tests are run and reported.

`DiscoverRunner.test_loader`

This is the class that loads tests, whether from TestCases or modules or otherwise and bundles them into test suites for the runner to execute. By default it is set to `unittest.defaultTestLoader`. You can override this attribute if your tests are going to be loaded in unusual ways.

Methods

`DiscoverRunner.run_tests(test_labels, **kwargs)` [source]

Run the test suite.

`test_labels` allows you to specify which tests to run and supports several formats (see `DiscoverRunner.build_suite()` for a list of supported formats).

This method should return the number of tests that failed.

`classmethod DiscoverRunner.add_arguments(parser)` [source]

Override this class method to add custom arguments accepted by the `test` management command. See `argparse.ArgumentParser.add_argument()` for details about adding arguments to a parser.

`DiscoverRunner.setup_test_environment(**kwargs)` [source]

Sets up the test environment by calling `setup_test_environment()` and setting `DEBUG` to `self.debug_mode` (defaults to `False`).

`DiscoverRunner.build_suite(test_labels=None, **kwargs)` [source]

Constructs a test suite that matches the test labels provided.

`test_labels` is a list of strings describing the tests to be run. A test label can take one of four forms:

- `path.to.test_module.TestCase.test_method` – Run a single test method in a test case class.
- `path.to.test_module.TestCase` – Run all the test methods in a test case.
- `path.to.module` – Search for and run all tests in the named Python package or module.
- `path/to/directory` – Search for and run all tests below the named directory.

If `test_labels` has a value of `None`, the test runner will search for tests in all files below the current directory whose names match its `pattern` (see above).

Returns a `TestSuite` instance ready to be run.

`DiscoverRunner.setup_databases(**kwargs)` [source]

Creates the test databases by calling `setup_databases()`.

`DiscoverRunner.run_checks(databases)` [source]

Runs the system checks on the test `databases`.

`DiscoverRunner.run_suite(suite, **kwargs)` [source]

Runs the test suite.

Returns the result produced by the running the test suite.

`DiscoverRunner.get_test_runner_kwargs()` [source]

Returns the keyword arguments to instantiate the `DiscoverRunner.test_runner` with.

`DiscoverRunner.teardown_databases(old_config, **kwargs)` [source]

Destroys the test databases, restoring pre-test conditions by calling `teardown_databases()`.

`DiscoverRunner.teardown_test_environment(**kwargs)` [source]

Restores the pre-test environment.

`DiscoverRunner.suite_result(suite, result, **kwargs)` [source]

Computes and returns a return code based on a test suite, and the result from that test suite.

`DiscoverRunner.log(msg, level=None)` [source]

If a `logger` is set, logs the message at the given integer logging level (e.g. `logging.DEBUG`, `logging.INFO`, or `logging.WARNING`). Otherwise, the message is printed to the console, respecting the current `verbosity`. For example, no message will be printed if the `verbosity` is 0, `INFO` and above will be printed if the `verbosity` is at least 1, and `DEBUG` will be printed if it is at least 2. The `level` defaults to `logging.INFO`.

Testing utilities

`django.test.utils`

To assist in the creation of your own test runner, Django provides a number of utility methods in the `django.test.utils` module.

`setup_test_environment(debug=None)` [source]

Performs global pre-test setup, such as installing instrumentation for the template rendering system and setting up the dummy email outbox.

If `debug` isn't `None`, the `DEBUG` setting is updated to its value.

`teardown_test_environment()` [source]

Performs global post-test teardown, such as removing instrumentation from the template system and restoring normal email services.

`setup_databases(verbosity, interactive, *, time_keeper=None, keepdb=False, debug_sql=False, parallel=0, aliases=None, serialized_aliases=None, **kwargs)` [source]

Creates the test databases.

Returns a data structure that provides enough detail to undo the changes that have been made. This data will be provided to the `teardown_databases()` function at the conclusion of testing.

The `aliases` argument determines which `DATABASES` aliases test databases should be set up for. If it's not provided, it defaults to all of `DATABASES` aliases.

The `serialized_aliases` argument determines what subset of `aliases` test databases should have their state serialized to allow usage of the serialized_rollback feature. If it's not provided, it defaults to `aliases`.

`teardown_databases(old_config, parallel=0, keepdb=False)` [source]

Destroys the test databases, restoring pre-test conditions.

`old_config` is a data structure defining the changes in the database configuration that need to be reversed. It's the return value of the `setup_databases()` method.

`django.db.connection.creation`

The creation module of the database backend also provides some utilities that can be useful during testing.

`create_test_db(verbosity=1, autoclobber=False, serialize=True, keepdb=False)`

Creates a new test database and runs `migrate` against it.

`verbosity` has the same behavior as in `run_tests()`.

`autoclobber` describes the behavior that will occur if a database with the same name as the test database is discovered:

- If `autoclobber` is `False`, the user will be asked to approve destroying the existing database. `sys.exit` is called if the user does not approve.
- If `autoclobber` is `True`, the database will be destroyed without consulting the user.

`serialize` determines if Django serializes the database into an in-memory JSON string before running tests (used to restore the database state between tests if you don't have transactions). You can set this to `False` to speed up creation time if you don't have any test classes with serialized_rollback=True.

`keepdb` determines if the test run should use an existing database, or create a new one. If `True`, the existing database will be used, or created if not present. If `False`, a new database will be created, prompting the user to remove the existing one, if present.

Returns the name of the test database that it created.

`create_test_db()` has the side effect of modifying the value of `NAME` in `DATABASES` to match the name of the test database.

`destroy_test_db(old_database_name, verbosity=1, keepdb=False)`

Destroys the database whose name is the value of `NAME` in `DATABASES`, and sets `NAME` to the value of `old_database_name`.

The `verbosity` argument has the same behavior as for `DiscoverRunner`.

If the `keepdb` argument is `True`, then the connection to the database will be closed, but the database will not be destroyed.

Integration with `coverage.py`

Code coverage describes how much source code has been tested. It shows which parts of your code are being exercised by tests and which are not. It's an important part of testing applications, so it's strongly recommended to check the coverage of your tests.

Django can be easily integrated with coverage.py, a tool for measuring code coverage of Python programs. First, install coverage. Next, run the following from your project folder containing `manage.py`:

```
coverage run --source='.' manage.py test myapp
```

This runs your tests and collects coverage data of the executed files in your project. You can see a report of this data by typing following command:

```
coverage report
```

Note that some Django code was executed while running tests, but it is not listed here because of the `source` flag passed to the previous command.

For more options like annotated HTML listings detailing missed lines, see the coverage.py docs.