Testing tools

Django provides a small set of tools that come in handy when writing tests.

The test client

The test client is a Python class that acts as a dummy web browser, allowing you to test your views and interact with your Django-powered application programmatically.

Some of the things you can do with the test client are:

- Simulate GET and POST requests on a URL and observe the response – everything from low-level HTTP (result headers and status codes) to page content.
- See the chain of redirects (if any) and check the URL and status code at each step.
- Test that a given request is rendered by a given Django template, with a template context that contains certain values.

Note that the test client is not intended to be a replacement for Selenium or other "in-browser" frameworks. Django's test client has a different focus. In short:

- Use Django's test client to establish that the correct template is being rendered and that the template is passed the correct context data.
- Use `RequestFactory` to test view functions directly, bypassing the routing and middleware layers.
- Use in-browser frameworks like Selenium to test *rendered* HTML and the *behavior* of web pages, namely JavaScript functionality. Django also provides special support for those frameworks; see the section on `LiveServerTestCase` for more details.

A comprehensive test suite should use a combination of all of these test types.

Overview and a quick example

To use the test client, instantiate `django.test.Client` and retrieve web pages:

```
>>> from django.test import Client
>>> c = Client()
>>> response = c.post("/login/", {"username": "john", "password": "smith"})
>>> response.status_code
200
>>> response = c.get("/customer/details/")
>>> response.content
b'<!DOCTYPE html...'
```

As this example suggests, you can instantiate `Client` from within a session of the Python interactive interpreter.

Note a few important things about how the test client works:

- The test client does *not* require the web server to be running. In fact, it will run just fine with no web server running at all! That's because it avoids the overhead of HTTP and deals directly with the Django framework. This helps make the unit tests run quickly.
- When retrieving pages, remember to specify the *path* of the URL, not the whole domain. For example, this is correct:

  ```
  >>> c.get("/login/")
  ```

  This is incorrect:

  ```
  >>> c.get("https://www.example.com/login/")
  ```

  The test client is not capable of retrieving web pages that are not powered by your Django project. If you need to retrieve other web pages, use a Python standard library module such as `urllib`.
- To resolve URLs, the test client uses whatever URLconf is pointed-to by your `ROOT_URLCONF` setting.
- Although the above example would work in the Python interactive interpreter, some of the test client's functionality, notably the template-related functionality, is only available *while tests are running*.

  The reason for this is that Django's test runner performs a bit of black magic in order to determine which template was loaded by a given view. This black magic (essentially a patching of Django's template system in memory) only happens during test running.
- By default, the test client will disable any CSRF checks performed by your site.

  If, for some reason, you *want* the test client to perform CSRF checks, you can create an instance of the test client that enforces CSRF checks. To do this, pass in the `enforce_csrf_checks` argument when you construct your client:

  ```
  >>> from django.test import Client
  >>> csrf_client = Client(enforce_csrf_checks=True)
  ```

Making requests

Use the `django.test.Client` class to make requests.

`class Client(enforce_csrf_checks=False, raise_request_exception=True, json_encoder=DjangoJSONEncoder, *, headers=None, query_params=None, **defaults)` **[source]**

A testing HTTP client. Takes several arguments that can customize behavior.

`headers` allows you to specify default headers that will be sent with every request. For example, to set a `User-Agent` header:

```
client = Client(headers={"user-agent": "curl/7.79.1"})
```

`query_params` allows you to specify the default query string that will be set on every request.

Arbitrary keyword arguments in `**defaults` set WSGI **environ variables**. For example, to set the script name:

```
client = Client(SCRIPT_NAME="/app/")
```

Note: Keyword arguments starting with a `HTTP_` prefix are set as headers, but the `headers` parameter should be preferred for readability.

The values from the `headers`, `query_params`, and `extra` keyword arguments passed to `get()`, `post()`, etc. have precedence over the defaults passed to the class constructor.

The `enforce_csrf_checks` argument can be used to test CSRF protection (see above).

The `raise_request_exception` argument allows controlling whether or not exceptions raised during the request should also be raised in the test. Defaults to `True`.

The `json_encoder` argument allows setting a custom JSON encoder for the JSON serialization that's described in `post()`.

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

Once you have a `Client` instance, you can call any of the following methods:

`get(path, data=None, follow=False, secure=False, *, headers=None, query_params=None, **extra)` **[source]**

Makes a GET request on the provided `path` and returns a `Response` object, which is documented below.

The key-value pairs in the `query_params` dictionary are used to set query strings. For example:

```
>>> c = Client()
>>> c.get("/customers/details/", query_params={"name": "fred", "age": 7})
```

...will result in the evaluation of a GET request equivalent to:

```
/customers/details/?name=fred&age=7
```

It is also possible to pass these parameters into the `data` parameter. However, `query_params` is preferred as it works for any HTTP method.

The `headers` parameter can be used to specify headers to be sent in the request. For example:

```
>>> c = Client()
>>> c.get(
...     "/customers/details/",
...     query_params={"name": "fred", "age": 7},
...     headers={"accept": "application/json"},
... )
```

...will send the HTTP header `HTTP_ACCEPT` to the details view, which is a good way to test code paths that use the `django.http.HttpRequest.accepts()` method.

Arbitrary keyword arguments set WSGI **environ variables**. For example, headers to set the script name:

```
>>> c = Client()
>>> c.get("/", SCRIPT_NAME="/app/")
```

If you already have the GET arguments in URL-encoded form, you can use that encoding instead of using the data argument. For example, the previous GET request could also be posed as:

```
>>> c = Client()
>>> c.get("/customers/details/?name=fred&age=7")
```

If you provide a URL with both an encoded GET data and either a query_params or data argument these arguments will take precedence.

If you set `follow` to `True` the client will follow any redirects and a `redirect_chain` attribute will be set in the response object containing tuples of the intermediate urls and status codes.

If you had a URL `/redirect_me/` that redirected to `/next/`, that redirected to `/final/`, this is what you'd see:

```
>>> response = c.get("/redirect_me/", follow=True)
>>> response.redirect_chain
[('http://testserver/next/', 302), ('http://testserver/final/', 302)]
```

If you set `secure` to `True` the client will emulate an HTTPS request.

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

`post(path, data=None, content_type=MULTIPART_CONTENT, follow=False, secure=False, *, headers=None, query_params=None, **extra)` [source]

Makes a POST request on the provided `path` and returns a `Response` object, which is documented below.

The key-value pairs in the `data` dictionary are used to submit POST data. For example:

```
>>> c = Client()
>>> c.post("/login/", {"name": "fred", "passwd": "secret"})
```

...will result in the evaluation of a POST request to this URL:

```
/login/
```

...with this POST data:

```
name=fred&passwd=secret
```

If you provide `content_type` as *application/json*, the `data` is serialized using `json.dumps()` if it's a dict, list, or tuple. Serialization is performed with `DjangoJSONEncoder` by default, and can be overridden by providing a `json_encoder` argument to `Client`. This serialization also happens for `put()`, `patch()`, and `delete()` requests.

If you provide any other `content_type` (e.g. *text/xml* for an XML payload), the contents of `data` are sent as-is in the POST request, using `content_type` in the HTTP `Content-Type` header.

If you don't provide a value for `content_type`, the values in `data` will be transmitted with a content type of *multipart/form-data*. In this case, the key-value pairs in `data` will be encoded as a multipart message and used to create the POST data payload.

To submit multiple values for a given key – for example, to specify the selections for a `<select multiple>` – provide the values as a list or tuple for the required key. For example, this value of `data` would submit three selected values for the field named `choices`:

```
{"choices": ["a", "b", "d"]}
```

Submitting files is a special case. To POST a file, you need only provide the file field name as a key, and a file handle to the file you wish to upload as a value. For example, if your form has fields `name` and `attachment`, the latter a `FileField`:

```
>>> c = Client()
>>> with open("wishlist.doc", "rb") as fp:
...     c.post("/customers/wishes/", {"name": "fred", "attachment": fp})
...
```

You may also provide any file-like object (e.g., StringIO or BytesIO) as a file handle. If you're uploading to an ImageField, the object needs a `name` attribute that passes the validate_image_file_extension validator. For example:

```
>>> from io import BytesIO
>>> img = BytesIO(
...     b"GIF89a\x01\x00\x01\x00\x00\x00\x00!\xf9\x04\x01\x00\x00\x00"
...     b"\x00,\x00\x00\x00\x00\x01\x00\x01\x00\x00\x02\x01\x00\x00"
... )
>>> img.name = "myimage.gif"
```

Note that if you wish to use the same file handle for multiple `post()` calls then you will need to manually reset the file pointer between posts. The easiest way to do this is to manually close the file after it has been provided to `post()`, as demonstrated above.

You should also ensure that the file is opened in a way that allows the data to be read. If your file contains binary data such as an image, this means you will need to open the file in `rb` (read binary) mode.

The `headers`, `query_params`, and `extra` parameters acts the same as for Client.get().

If the URL you request with a POST contains encoded parameters, these parameters will be made available in the request.GET data. For example, if you were to make the request:

```
>>> c.post(
...     "/login/", {"name": "fred", "passwd": "secret"}, query_params={"visitor": "true"}
... )
```

... the view handling this request could interrogate request.POST to retrieve the username and password, and could interrogate request.GET to determine if the user was a visitor.

If you set `follow` to `True` the client will follow any redirects and a `redirect_chain` attribute will be set in the response object containing tuples of the intermediate urls and status codes.

If you set `secure` to `True` the client will emulate an HTTPS request.

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

`head(path, data=None, follow=False, secure=False, *, headers=None, query_params=None, **extra)` [source]

Makes a HEAD request on the provided `path` and returns a `Response` object. This method works just like Client.get(), including the `follow`, `secure`, `headers`, `query_params`, and `extra` parameters, except it does not return a message body.

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

`options(path, data='', content_type='application/octet-stream', follow=False, secure=False, *, headers=None, query_params=None, **extra)` [source]

Makes an OPTIONS request on the provided `path` and returns a `Response` object. Useful for testing RESTful interfaces.

When `data` is provided, it is used as the request body, and a `Content-Type` header is set to `content_type`.

The `follow`, `secure`, `headers`, `query_params`, and `extra` parameters act the same as for Client.get().

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

`put(path, data='', content_type='application/octet-stream', follow=False, secure=False, *, headers=None, query_params=None, **extra)` [source]

Makes a PUT request on the provided `path` and returns a `Response` object. Useful for testing RESTful interfaces.

When `data` is provided, it is used as the request body, and a `Content-Type` header is set to `content_type`.

The `follow`, `secure`, `headers`, `query_params`, and `extra` parameters act the same as for Client.get().

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

`patch(path, data='', content_type='application/octet-stream', follow=False, secure=False, *, headers=None, query_params=None, **extra)` [source]

Makes a PATCH request on the provided `path` and returns a `Response` object. Useful for testing RESTful interfaces.

The `follow`, `secure`, `headers`, `query_params`, and `extra` parameters act the same as for `Client.get()`.

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

`delete(path, data='', content_type='application/octet-stream', follow=False, secure=False, *, headers=None, query_params=None, **extra)` [source]

Makes a DELETE request on the provided `path` and returns a `Response` object. Useful for testing RESTful interfaces.

When `data` is provided, it is used as the request body, and a `Content-Type` header is set to `content_type`.

The `follow`, `secure`, `headers`, `query_params`, and `extra` parameters act the same as for `Client.get()`.

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

`trace(path, follow=False, secure=False, *, headers=None, query_params=None, **extra)` [source]

Makes a TRACE request on the provided `path` and returns a `Response` object. Useful for simulating diagnostic probes.

Unlike the other request methods, `data` is not provided as a keyword parameter in order to comply with **RFC 9110#section-9.3.8**, which mandates that TRACE requests must not have a body.

The `follow`, `secure`, `headers`, `query_params`, and `extra` parameters act the same as for `Client.get()`.

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

`login(**credentials)`

`alogin(**credentials)`

Asynchronous version: `alogin()`

If your site uses Django's authentication system and you deal with logging in users, you can use the test client's `login()` method to simulate the effect of a user logging into the site.

After you call this method, the test client will have all the cookies and session data required to pass any login-based tests that may form part of a view.

The format of the `credentials` argument depends on which authentication backend you're using (which is configured by your `AUTHENTICATION_BACKENDS` setting). If you're using the standard authentication backend provided by Django (`ModelBackend`), `credentials` should be the user's username and password, provided as keyword arguments:

```
>>> c = Client()
>>> c.login(username="fred", password="secret")

# Now you can access a view that's only available to logged-in users.
```

If you're using a different authentication backend, this method may require different credentials. It requires whichever credentials are required by your backend's `authenticate()` method.

`login()` returns `True` if it the credentials were accepted and login was successful.

Finally, you'll need to remember to create user accounts before you can use this method. As we explained above, the test runner is executed using a test database, which contains no users by default. As a result, user accounts that are valid on your production site will not work under test conditions. You'll need to create users as part of the test suite – either manually (using the Django model API) or with a test fixture. Remember that if you want your test user to have a password, you can't set the user's password by setting the password attribute directly – you must use the `set_password()` function to store a correctly hashed password. Alternatively, you can use the `create_user()` helper method to create a new user with a correctly hashed password.

Changed in Django 5.0:

`alogin()` method was added.

`force_login(user, backend=None)`

`aforce_login(user, backend=None)`

Asynchronous version: `aforce_login()`

If your site uses Django's authentication system, you can use the `force_login()` method to simulate the effect of a user logging into the site. Use this method instead of `login()` when a test requires a user be logged in and the details of how a user logged in aren't important.

Unlike `login()`, this method skips the authentication and verification steps: inactive users (`is_active=False`) are permitted to login and the user's credentials don't need to be provided.

The user will have its `backend` attribute set to the value of the `backend` argument (which should be a dotted Python path string), or to `settings.AUTHENTICATION_BACKENDS[0]` if a value isn't provided. The `authenticate()` function called by `login()` normally annotates the user like this.

This method is faster than `login()` since the expensive password hashing algorithms are bypassed. Also, you can speed up `login()` by using a weaker hasher while testing.

Changed in Django 5.0:
`aforce_login()` method was added.

`logout()`

`alogout()`

Asynchronous version: `alogout()`

If your site uses Django's authentication system, the `logout()` method can be used to simulate the effect of a user logging out of your site.

After you call this method, the test client will have all the cookies and session data cleared to defaults. Subsequent requests will appear to come from an `AnonymousUser`.

Changed in Django 5.0:
`alogout()` method was added.

Testing responses

The `get()` and `post()` methods both return a `Response` object. This `Response` object is *not* the same as the `HttpResponse` object returned by Django views; the test response object has some additional data useful for test code to verify.

Specifically, a `Response` object has the following attributes:

`class Response`

`client`

The test client that was used to make the request that resulted in the response.

`content`

The body of the response, as a bytestring. This is the final page content as rendered by the view, or any error message.

`context`

The template `Context` instance that was used to render the template that produced the response content.

If the rendered page used multiple templates, then `context` will be a list of `Context` objects, in the order in which they were rendered.

Regardless of the number of templates used during rendering, you can retrieve context values using the `[]` operator. For example, the context variable `name` could be retrieved using:

```
>>> response = client.get("/foo/")
>>> response.context["name"]
'Arthur'
```

Not using Django templates?: This attribute is only populated when using the `DjangoTemplates` backend. If you're using another template engine, `context_data` may be a suitable alternative on responses with that attribute.

`exc_info`

A tuple of three values that provides information about the unhandled exception, if any, that occurred during the view.

The values are (type, value, traceback), the same as returned by Python's `sys.exc_info()`. Their meanings are:

- *type*: The type of the exception.
- *value*: The exception instance.
- *traceback*: A traceback object which encapsulates the call stack at the point where the exception originally occurred.

If no exception occurred, then `exc_info` will be `None`.

`json(**kwargs)`

The body of the response, parsed as JSON. Extra keyword arguments are passed to `json.loads()`. For example:

```
>>> response = client.get("/foo/")
>>> response.json()["name"]
'Arthur'
```

If the `Content-Type` header is not `"application/json"`, then a `ValueError` will be raised when trying to parse the response.

`request`

The request data that stimulated the response.

`wsgi_request`

The `WSGIRequest` instance generated by the test handler that generated the response.

`status_code`

The HTTP status of the response, as an integer. For a full list of defined codes, see the IANA status code registry.

`templates`

A list of `Template` instances used to render the final content, in the order they were rendered. For each template in the list, use `template.name` to get the template's file name, if the template was loaded from a file. (The name is a string such as `'admin/index.html'`.)

Not using Django templates?: This attribute is only populated when using the `DjangoTemplates` backend. If you're using another template engine, `template_name` may be a suitable alternative if you only need the name of the template used for rendering.

`resolver_match`

An instance of `ResolverMatch` for the response. You can use the `func` attribute, for example, to verify the view that served the response:

```
# my_view here is a function based view.
self.assertEqual(response.resolver_match.func, my_view)

# Class-based views need to compare the view_class, as the
# functions generated by as_view() won't be equal.
self.assertIs(response.resolver_match.func.view_class, MyView)
```

If the given URL is not found, accessing this attribute will raise a `Resolver404` exception.

As with a normal response, you can also access the headers through `HttpResponse.headers`. For example, you could determine the content type of a response using `response.headers['Content-Type']`.

Exceptions

If you point the test client at a view that raises an exception and `Client.raise_request_exception` is `True`, that exception will be visible in the test case. You can then use a standard `try ... except` block or `assertRaises()` to test for exceptions.

The only exceptions that are not visible to the test client are `Http404`, `PermissionDenied`, `SystemExit`, and `SuspiciousOperation`. Django catches these exceptions internally and converts them into the appropriate HTTP response codes. In these cases, you can check `response.status_code` in your test.

If `Client.raise_request_exception` is `False`, the test client will return a 500 response as would be returned to a browser. The response has the attribute `exc_info` to provide information about the unhandled exception.

Persistent state

The test client is stateful. If a response returns a cookie, then that cookie will be stored in the test client and sent with all subsequent `get()` and `post()` requests.

Expiration policies for these cookies are not followed. If you want a cookie to expire, either delete it manually or create a new `Client` instance (which will effectively delete all cookies).

A test client has attributes that store persistent state information. You can access these properties as part of a test condition.

`Client.cookies`

A Python `SimpleCookie` object, containing the current values of all the client cookies. See the documentation of the `http.cookies` module for more.

`Client.session`

A dictionary-like object containing session information. See the session documentation for full details.

To modify the session and then save it, it must be stored in a variable first (because a new `SessionStore` is created every time this property is accessed):

```
def test_something(self):
    session = self.client.session
    session["somekey"] = "test"
    session.save()
```

`Client.asession()`

New in Django 5.0.

This is similar to the `session` attribute but it works in async contexts.

Setting the language

When testing applications that support internationalization and localization, you might want to set the language for a test client request. The method for doing so depends on whether or not the `LocaleMiddleware` is enabled.

If the middleware is enabled, the language can be set by creating a cookie with a name of `LANGUAGE_COOKIE_NAME` and a value of the language code:

```
from django.conf import settings


def test_language_using_cookie(self):
    self.client.cookies.load({settings.LANGUAGE_COOKIE_NAME: "fr"})
    response = self.client.get("/")
    self.assertEqual(response.content, b"Bienvenue sur mon site.")
```

or by including the `Accept-Language` HTTP header in the request:

```
def test_language_using_header(self):
    response = self.client.get("/", headers={"accept-language": "fr"})
    self.assertEqual(response.content, b"Bienvenue sur mon site.")
```

Note: When using these methods, ensure to reset the active language at the end of each test:

```
def tearDown(self):
    translation.activate(settings.LANGUAGE_CODE)
```

More details are in How Django discovers language preference.

If the middleware isn't enabled, the active language may be set using `translation.override()`:

```
from django.utils import translation


def test_language_using_override(self):
    with translation.override("fr"):
        response = self.client.get("/")
    self.assertEqual(response.content, b"Bienvenue sur mon site.")
```

More details are in Explicitly setting the active language.

Example

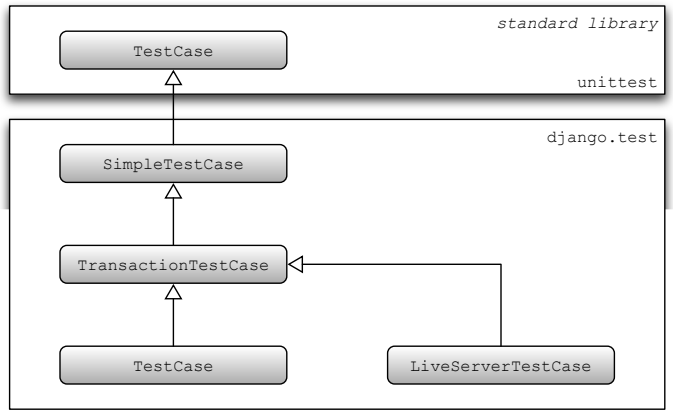
The following is a unit test using the test client:

```
import unittest
from django.test import Client


class SimpleTest(unittest.TestCase):
    def setUp(self):
        # Every test needs a client.
        self.client = Client()

    def test_details(self):
        # Issue a GET request.
        response = self.client.get("/customer/details/")

        # Check that the response is 200 OK.
        self.assertEqual(response.status_code, 200)

        # Check that the rendered context contains 5 customers.
        self.assertEqual(len(response.context["customers"]), 5)
```

See also: `django.test.RequestFactory`

Provided test case classes

Normal Python unit test classes extend a base class of `unittest.TestCase`. Django provides a few extensions of this base class:



Hierarchy of Django unit testing classes

You can convert a normal `unittest.TestCase` to any of the subclasses: change the base class of your test from `unittest.TestCase` to the subclass. All of the standard Python unit test functionality will be available, and it will be augmented with some useful additions as described in each section below.

SimpleTestCase

`class SimpleTestCase` [source]

A subclass of `unittest.TestCase` that adds this functionality:

- Some useful assertions like:
  - Checking that a callable `raises a certain exception`.
  - Checking that a callable `triggers a certain warning`.
  - Testing form field `rendering and error treatment`.
  - Testing `HTML responses for the presence/lack of a given fragment`.
  - Verifying that a template `has/hasn't been used to generate a given response content`.
  - Verifying that two `URLs` are equal.
  - Verifying an HTTP `redirect` is performed by the app.
  - Robustly testing two `HTML fragments` for equality/inequality or `containment`.
  - Robustly testing two `XML fragments` for equality/inequality.
  - Robustly testing two `JSON fragments` for equality.
- The ability to run tests with modified settings.
- Using the `client` `Client`.

If your tests make any database queries, use subclasses `TransactionTestCase` or `TestCase`.

SimpleTestCase.databases

`SimpleTestCase` disallows database queries by default. This helps to avoid executing write queries which will affect other tests since each `SimpleTestCase` test isn't run in a transaction. If you aren't concerned about this problem, you can disable this behavior by setting the `databases` class attribute to `'__all__'` on your test class.

Warning: `SimpleTestCase` and its subclasses (e.g. `TestCase`, ...) rely on `setUpClass()` and `tearDownClass()` to perform some class-wide initialization (e.g. overriding settings). If you need to override those methods, don't forget to call the `super` implementation:

```
class MyTestCase(TestCase):
    @classmethod
    def setUpClass(cls):
        super().setUpClass()
        ...

    @classmethod
    def tearDownClass(cls):
        ...
        super().tearDownClass()
```

Be sure to account for Python's behavior if an exception is raised during `setUpClass()`. If that happens, neither the tests in the class nor `tearDownClass()` are run. In the case of `django.test.TestCase`, this will leak the transaction created in `super()` which results in various symptoms including a segmentation fault on some platforms (reported on macOS). If you want to intentionally raise an exception such as `unittest.SkipTest` in `setUpClass()`, be sure to do it before calling `super()` to avoid this.

TransactionTestCase

`class TransactionTestCase` [source]

`TransactionTestCase` inherits from `SimpleTestCase` to add some database-specific features:

- Resetting the database to a known state at the beginning of each test to ease testing and using the ORM.
- Database `fixtures`.
- Test skipping based on database backend features.
- The remaining specialized `assert*` methods.

Django's `TestCase` class is a more commonly used subclass of `TransactionTestCase` that makes use of database transaction facilities to speed up the process of resetting the database to a known state at the beginning of each test. A consequence of this, however, is that some database behaviors cannot be tested within a Django `TestCase` class. For instance, you cannot test that a block of code is executing within a transaction, as is required when using `select_for_update()`. In those cases, you should use `TransactionTestCase`.

`TransactionTestCase` and `TestCase` are identical except for the manner in which the database is reset to a known state and the ability for test code to test the effects of commit and rollback:

- A `TransactionTestCase` resets the database after the test runs by truncating all tables. A `TransactionTestCase` may call commit and rollback and observe the effects of these calls on the database.
- A `TestCase`, on the other hand, does not truncate tables after a test. Instead, it encloses the test code in a database transaction that is rolled back at the end of the test. This guarantees that the rollback at the end of the test restores the database to its initial state.

Warning: `TestCase` running on a database that does not support rollback (e.g. MySQL with the MyISAM storage engine), and all instances of `TransactionTestCase`, will roll back at the end of the test by deleting all data from the test database.

Apps will not see their data reloaded; if you need this functionality (for example, third-party apps should enable this) you can set `serialized_rollback = True` inside the `TestCase` body.

`TestCase`

`class TestCase` [source]

This is the most common class to use for writing tests in Django. It inherits from `TransactionTestCase` (and by extension `SimpleTestCase`). If your Django application doesn't use a database, use `SimpleTestCase`.

The class:

- Wraps the tests within two nested `atomic()` blocks: one for the whole class and one for each test. Therefore, if you want to test some specific database transaction behavior, use `TransactionTestCase`.
- Checks deferrable database constraints at the end of each test.

It also provides an additional method:

`classmethod TestCase.setUpTestData()` [source]

The class-level `atomic` block described above allows the creation of initial data at the class level, once for the whole `TestCase`. This technique allows for faster tests as compared to using `setUp()`.

For example:

```
from django.test import TestCase


class MyTests(TestCase):
    @classmethod
    def setUpTestData(cls):
        # Set up data for the whole TestCase
        cls.foo = Foo.objects.create(bar="Test")
        ...

    def test1(self):
        # Some test using self.foo
        ...

    def test2(self):
        # Some other test using self.foo
        ...
```

Note that if the tests are run on a database with no transaction support (for instance, MySQL with the MyISAM engine), `setUpTestData()` will be called before each test, negating the speed benefits.

Objects assigned to class attributes in `setUpTestData()` must support creating deep copies with `copy.deepcopy()` in order to isolate them from alterations performed by each test methods.

`classmethod TestCase.captureOnCommitCallbacks(using=DEFAULT_DB_ALIAS, execute=False)` [source]

Returns a context manager that captures `transaction.on_commit()` callbacks for the given database connection. It returns a list that contains, on exit of the context, the captured callback functions. From this list you can make assertions on the callbacks or call them to invoke their side effects, emulating a commit.

`using` is the alias of the database connection to capture callbacks for.

If `execute` is `True`, all the callbacks will be called as the context manager exits, if no exception occurred. This emulates a commit after the wrapped block of code.

For example:

```
from django.core import mail
from django.test import TestCase


class ContactTests(TestCase):
    def test_post(self):
        with self.captureOnCommitCallbacks(execute=True) as callbacks:
            response = self.client.post(
                "/contact/",
                {"message": "I like your site"},
            )

        self.assertEqual(response.status_code, 200)
        self.assertEqual(len(callbacks), 1)
        self.assertEqual(len(mail.outbox), 1)
        self.assertEqual(mail.outbox[0].subject, "Contact Form")
        self.assertEqual(mail.outbox[0].body, "I like your site")
```

`LiveServerTestCase`

`class LiveServerTestCase` [source]

`LiveServerTestCase` does basically the same as `TransactionTestCase` with one extra feature: it launches a live Django server in the background on setup, and shuts it down on teardown. This allows the use of automated test clients other than the Django dummy client such as, for example, the Selenium client, to execute a series of functional tests inside a browser and simulate a real user's actions.

The live server listens on `localhost` and binds to port 0 which uses a free port assigned by the operating system. The server's URL can be accessed with `self.live_server_url` during the tests.

To demonstrate how to use `LiveServerTestCase`, let's write a Selenium test. First of all, you need to install the selenium package:

```
$ python -m pip install "selenium >= 4.8.0"
```

```
...\> py -m pip install "selenium >= 4.8.0"
```

Then, add a `LiveServerTestCase`-based test to your app's tests module (for example: `myapp/tests.py`). For this example, we'll assume you're using the `staticfiles` app and want to have static files served during the execution of your tests similar to what we get at development time with `DEBUG=True`, i.e. without having to collect them using `collectstatic`. We'll use the `StaticLiveServerTestCase` subclass which provides that functionality. Replace it with `django.test.LiveServerTestCase` if you don't need that.

The code for this test may look as follows:

```
from django.contrib.staticfiles.testing import StaticLiveServerTestCase
from selenium.webdriver.common.by import By
from selenium.webdriver.firefox.webdriver import WebDriver


class MySeleniumTests(StaticLiveServerTestCase):
    fixtures = ["user-data.json"]

    @classmethod
    def setUpClass(cls):
        super().setUpClass()
        cls.selenium = WebDriver()
        cls.selenium.implicitly_wait(10)

    @classmethod
    def tearDownClass(cls):
        cls.selenium.quit()
        super().tearDownClass()

    def test_login(self):
        self.selenium.get(f"{self.live_server_url}/login/")
```

```
        username_input = self.selenium.find_element(By.NAME, "username")
        username_input.send_keys("myuser")
        password_input = self.selenium.find_element(By.NAME, "password")
        password_input.send_keys("secret")
        self.selenium.find_element(By.XPATH, '//input[@value="Log in"]').click()
```

Finally, you may run the test as follows:

```
$ ./manage.py test myapp.tests.MySeleniumTests.test_login
```

```
...\> manage.py test myapp.tests.MySeleniumTests.test_login
```

This example will automatically open Firefox then go to the login page, enter the credentials and press the “Log in” button. Selenium offers other drivers in case you do not have Firefox installed or wish to use another browser. The example above is just a tiny fraction of what the Selenium client can do; check out the full reference for more details.

Note: When using an in-memory SQLite database to run the tests, the same database connection will be shared by two threads in parallel: the thread in which the live server is run and the thread in which the test case is run. It’s important to prevent simultaneous database queries via this shared connection by the two threads, as that may sometimes randomly cause the tests to fail. So you need to ensure that the two threads don’t access the database at the same time. In particular, this means that in some cases (for example, just after clicking a link or submitting a form), you might need to check that a response is received by Selenium and that the next page is loaded before proceeding with further test execution. Do this, for example, by making Selenium wait until the `<body>` HTML tag is found in the response (requires Selenium > 2.13):

```
def test_login(self):
    from selenium.webdriver.support.wait import WebDriverWait

    timeout = 2
    ...
    self.selenium.find_element(By.XPATH, '//input[@value="Log in"]').click()
    # Wait until the response is received
    WebDriverWait(self.selenium, timeout).until(
        lambda driver: driver.find_element(By.TAG_NAME, "body")
    )
```

The tricky thing here is that there’s really no such thing as a “page load,” especially in modern web apps that generate HTML dynamically after the server generates the initial document. So, checking for the presence of `<body>` in the response might not necessarily be appropriate for all use cases. Please refer to the Selenium FAQ and Selenium documentation for more information.

Test cases features

Default test client

```
SimpleTestCase.client
```

Every test case in a `django.test.*TestCase` instance has access to an instance of a Django test client. This client can be accessed as `self.client`. This client is recreated for each test, so you don’t have to worry about state (such as cookies) carrying over from one test to another.

This means, instead of instantiating a `Client` in each test:

```
import unittest
from django.test import Client


class SimpleTest(unittest.TestCase):
    def test_details(self):
        client = Client()
        response = client.get("/customer/details/")
        self.assertEqual(response.status_code, 200)

    def test_index(self):
        client = Client()
        response = client.get("/customer/index/")
        self.assertEqual(response.status_code, 200)
```

...you can refer to `self.client`, like so:

```
from django.test import TestCase


class SimpleTest(TestCase):
    def test_details(self):
        response = self.client.get("/customer/details/")
        self.assertEqual(response.status_code, 200)

    def test_index(self):
        response = self.client.get("/customer/index/")
        self.assertEqual(response.status_code, 200)
```

Customizing the test client

`SimpleTestCase.client_class`

If you want to use a different `Client` class (for example, a subclass with customized behavior), use the `client_class` class attribute:

```
from django.test import Client, TestCase


class MyTestClient(Client):
    # Specialized methods for your environment
    ...


class MyTest(TestCase):
    client_class = MyTestClient

    def test_my_stuff(self):
        # Here self.client is an instance of MyTestClient...
        call_some_test_code()
```

Fixture loading

`TransactionTestCase.fixtures`

A test case class for a database-backed website isn't much use if there isn't any data in the database. Tests are more readable and it's more maintainable to create objects using the ORM, for example in `TestCase.setUpTestData()`, however, you can also use fixtures.

A fixture is a collection of data that Django knows how to import into a database. For example, if your site has user accounts, you might set up a fixture of fake user accounts in order to populate your database during tests.

The most straightforward way of creating a fixture is to use the `manage.py dumpdata` command. This assumes you already have some data in your database. See the `dumpdata documentation` for more details.

Once you've created a fixture and placed it in a `fixtures` directory in one of your `INSTALLED_APPS`, you can use it in your unit tests by specifying a `fixtures` class attribute on your `django.test.TestCase` subclass:

```
from django.test import TestCase
from myapp.models import Animal


class AnimalTestCase(TestCase):
    fixtures = ["mammals.json", "birds"]

    def setUp(self):
        # Test definitions as before.
        call_setup_methods()

    def test_fluffy_animals(self):
        # A test that uses the fixtures.
        call_some_test_code()
```

Here's specifically what will happen:

- At the start of each test, before `setUp()` is run, Django will flush the database, returning the database to the state it was in directly after `migrate` was called.

- Then, all the named fixtures are installed. In this example, Django will install any JSON fixture named `mammals`, followed by any fixture named `birds`. See the Fixtures topic for more details on defining and installing fixtures.

For performance reasons, `TestCase` loads fixtures once for the entire test class, before `setUpTestData()`, instead of before each test, and it uses transactions to clean the database before each test. In any case, you can be certain that the outcome of a test will not be affected by another test or by the order of test execution.

By default, fixtures are only loaded into the `default` database. If you are using multiple databases and set `TransactionTestCase.databases`, fixtures will be loaded into all specified databases.

URLconf configuration

If your application provides views, you may want to include tests that use the test client to exercise those views. However, an end user is free to deploy the views in your application at any URL of their choosing. This means that your tests can't rely upon the fact that your views will be available at a particular URL. Decorate your test class or test method with `@override_settings(ROOT_URLCONF=...)` for URLconf configuration.

Multi-database support

`TransactionTestCase.databases`

Django sets up a test database corresponding to every database that is defined in the `DATABASES` definition in your settings and referred to by at least one test through `databases`.

However, a big part of the time taken to run a Django `TestCase` is consumed by the call to `flush` that ensures that you have a clean database at the start of each test run. If you have multiple databases, multiple flushes are required (one for each database), which can be a time consuming activity – especially if your tests don't need to test multi-database activity.

As an optimization, Django only flushes the `default` database at the start of each test run. If your setup contains multiple databases, and you have a test that requires every database to be clean, you can use the `databases` attribute on the test suite to request extra databases to be flushed.

For example:

```
class TestMyViews(TransactionTestCase):
    databases = {"default", "other"}

    def test_index_page_view(self):
        call_some_test_code()
```

This test case class will flush the `default` and `other` test databases before running `test_index_page_view`. You can also use `'__all__'` to specify that all of the test databases must be flushed.

The `databases` flag also controls which databases the `TransactionTestCase.fixtures` are loaded into. By default, fixtures are only loaded into the `default` database.

Queries against databases not in `databases` will give assertion errors to prevent state leaking between tests.

`TestCase.databases`

By default, only the `default` database will be wrapped in a transaction during a `TestCase`'s execution and attempts to query other databases will result in assertion errors to prevent state leaking between tests.

Use the `databases` class attribute on the test class to request transaction wrapping against non-`default` databases.

For example:

```
class OtherDBTests(TestCase):
    databases = {"other"}

    def test_other_db_query(self): ...
```

This test will only allow queries against the `other` database. Just like for `SimpleTestCase.databases` and `TransactionTestCase.databases`, the `'__all__'` constant can be used to specify that the test should allow queries to all databases.

Overriding settings

Warning: Use the functions below to temporarily alter the value of settings in tests. Don't manipulate `django.conf.settings` directly as Django won't restore the original values after such manipulations.

`SimpleTestCase.settings()` [source]

For testing purposes it's often useful to change a setting temporarily and revert to the original value after running the testing code. For this use case Django provides a standard Python context manager (see **PEP 343**) called `settings()`, which can be used like this:

```
from django.test import TestCase


class LoginTestCase(TestCase):
    def test_login(self):
        # First check for the default behavior
        response = self.client.get("/sekrit/")
        self.assertRedirects(response, "/accounts/login/?next=/sekrit/")

        # Then override the LOGIN_URL setting
        with self.settings(LOGIN_URL="/other/login/"):
            response = self.client.get("/sekrit/")
            self.assertRedirects(response, "/other/login/?next=/sekrit/")
```

This example will override the `LOGIN_URL` setting for the code in the `with` block and reset its value to the previous state afterward.

`SimpleTestCase.modify_settings()` [source]

It can prove unwieldy to redefine settings that contain a list of values. In practice, adding or removing values is often sufficient. Django provides the `modify_settings()` context manager for easier settings changes:

```
from django.test import TestCase


class MiddlewareTestCase(TestCase):
    def test_cache_middleware(self):
        with self.modify_settings(
            MIDDLEWARE={
                "append": "django.middleware.cache.FetchFromCacheMiddleware",
                "prepend": "django.middleware.cache.UpdateCacheMiddleware",
                "remove": [
                    "django.contrib.sessions.middleware.SessionMiddleware",
                    "django.contrib.auth.middleware.AuthenticationMiddleware",
                    "django.contrib.messages.middleware.MessageMiddleware",
                ],
            }
        ):
            response = self.client.get("/")
            # ...
```

For each action, you can supply either a list of values or a string. When the value already exists in the list, `append` and `prepend` have no effect; neither does `remove` when the value doesn't exist.

`override_settings(**kwargs)` [source]

In case you want to override a setting for a test method, Django provides the `override_settings()` decorator (see **PEP 318**). It's used like this:

```
from django.test import TestCase, override_settings


class LoginTestCase(TestCase):
    @override_settings(LOGIN_URL="/other/login/")
    def test_login(self):
        response = self.client.get("/sekrit/")
        self.assertRedirects(response, "/other/login/?next=/sekrit/")
```

The decorator can also be applied to `TestCase` classes:

```
from django.test import TestCase, override_settings


@override_settings(LOGIN_URL="/other/login/")
class LoginTestCase(TestCase):
    def test_login(self):
        response = self.client.get("/sekrit/")
```

```
        self.assertRedirects(response, "/other/login/?next=/sekrit/")
```

```
modify_settings(*args, **kwargs)                                    [source]
```

Likewise, Django provides the `modify_settings()` decorator:

```
from django.test import TestCase, modify_settings


class MiddlewareTestCase(TestCase):
    @modify_settings(
        MIDDLEWARE={
            "append": "django.middleware.cache.FetchFromCacheMiddleware",
            "prepend": "django.middleware.cache.UpdateCacheMiddleware",
        }
    )
    def test_cache_middleware(self):
        response = self.client.get("/")
        # ...
```

The decorator can also be applied to test case classes:

```
from django.test import TestCase, modify_settings


@modify_settings(
    MIDDLEWARE={
        "append": "django.middleware.cache.FetchFromCacheMiddleware",
        "prepend": "django.middleware.cache.UpdateCacheMiddleware",
    }
)
class MiddlewareTestCase(TestCase):
    def test_cache_middleware(self):
        response = self.client.get("/")
        # ...
```

Note: When given a class, these decorators modify the class directly and return it; they don't create and return a modified copy of it. So if you try to tweak the above examples to assign the return value to a different name than `LoginTestCase` or `MiddlewareTestCase`, you may be surprised to find that the original test case classes are still equally affected by the decorator. For a given class, `modify_settings()` is always applied after `override_settings()`.

Warning: The settings file contains some settings that are only consulted during initialization of Django internals. If you change them with `override_settings`, the setting is changed if you access it via the `django.conf.settings` module, however, Django's internals access it differently. Effectively, using `override_settings()` or `modify_settings()` with these settings is probably not going to do what you expect it to do.

We do not recommend altering the `DATABASES` setting. Altering the `CACHES` setting is possible, but a bit tricky if you are using internals that make using of caching, like `django.contrib.sessions`. For example, you will have to reinitialize the session backend in a test that uses cached sessions and overrides `CACHES`.

Finally, avoid aliasing your settings as module-level constants as `override_settings()` won't work on such values since they are only evaluated the first time the module is imported.

You can also simulate the absence of a setting by deleting it after settings have been overridden, like this:

```
@override_settings()
def test_something(self):
    del settings.LOGIN_URL
    ...
```

When overriding settings, make sure to handle the cases in which your app's code uses a cache or similar feature that retains state even if the setting is changed. Django provides the `django.test.signals.setting_changed` signal that lets you register callbacks to clean up and otherwise reset state when settings are changed.

Django itself uses this signal to reset various data:

Overridden settings	Data reset
USE_TZ, TIME_ZONE	Databases timezone
TEMPLATES	Template engines
FORM_RENDERER	Default renderer
SERIALIZATION_MODULES	Serializers cache
LOCALE_PATHS, LANGUAGE_CODE	Default translation and loaded translations
STATIC_ROOT, STATIC_URL, STORAGES	Storages configuration

Changed in Django 5.1:
Resetting the default renderer when the `FORM_RENDERER` setting is changed was added.

Isolating apps

```
utils.isolate_apps(*app_labels, attr_name=None, kwarg_name=None)
```

Registers the models defined within a wrapped context into their own isolated `apps` registry. This functionality is useful when creating model classes for tests, as the classes will be cleanly deleted afterward, and there is no risk of name collisions.

The app labels which the isolated registry should contain must be passed as individual arguments. You can use `isolate_apps()` as a decorator or a context manager. For example:

```
from django.db import models
from django.test import SimpleTestCase
from django.test.utils import isolate_apps


class MyModelTests(SimpleTestCase):
    @isolate_apps("app_label")
    def test_model_definition(self):
        class TestModel(models.Model):
            pass

        ...
```

... or:

```
with isolate_apps("app_label"):

    class TestModel(models.Model):
        pass

    ...
```

The decorator form can also be applied to classes.

Two optional keyword arguments can be specified:

- `attr_name` : attribute assigned the isolated registry if used as a class decorator.
- `kwarg_name` : keyword argument passing the isolated registry if used as a function decorator.

The temporary `Apps` instance used to isolate model registration can be retrieved as an attribute when used as a class decorator by using the `attr_name` parameter:

```
@isolate_apps("app_label", attr_name="apps")
class TestModelDefinition(SimpleTestCase):
    def test_model_definition(self):
        class TestModel(models.Model):
            pass

        self.assertIs(self.apps.get_model("app_label", "TestModel"), TestModel)
```

... or alternatively as an argument on the test method when used as a method decorator by using the `kwarg_name` parameter:

```
class TestModelDefinition(SimpleTestCase):
    @isolate_apps("app_label", kwarg_name="apps")
    def test_model_definition(self, apps):
        class TestModel(models.Model):
            pass

        self.assertIs(apps.get_model("app_label", "TestModel"), TestModel)
```

Emptying the test outbox

If you use any of Django's custom `TestCase` classes, the test runner will clear the contents of the test email outbox at the start of each test case.

For more detail on email services during tests, see Email services below.

Assertions

As Python's normal `unittest.TestCase` class implements assertion methods such as `assertTrue()` and `assertEqual()`, Django's custom `TestCase` class provides a number of custom assertion methods that are useful for testing web applications:

The failure messages given by most of these assertion methods can be customized with the `msg_prefix` argument. This string will be prefixed to any failure message generated by the assertion. This allows you to provide additional details that may help you to identify the location and cause of a failure in your test suite.

`SimpleTestCase.assertRaisesMessage(expected_exception, expected_message, callable, *args, **kwargs)` [source]

`SimpleTestCase.assertRaisesMessage(expected_exception, expected_message)`

Asserts that execution of `callable` raises `expected_exception` and that `expected_message` is found in the exception's message. Any other outcome is reported as a failure. It's a simpler version of `unittest.TestCase.assertRaisesRegex()` with the difference that `expected_message` isn't treated as a regular expression.

If only the `expected_exception` and `expected_message` parameters are given, returns a context manager so that the code being tested can be written inline rather than as a function:

```
with self.assertRaisesMessage(ValueError, "invalid literal for int()"):
    int("a")
```

`SimpleTestCase.assertWarnsMessage(expected_warning, expected_message, callable, *args, **kwargs)` [source]

`SimpleTestCase.assertWarnsMessage(expected_warning, expected_message)`

Analogous to `SimpleTestCase.assertRaisesMessage()` but for `assertWarnsRegex()` instead of `assertRaisesRegex()`.

`SimpleTestCase.assertFieldOutput(fieldclass, valid, invalid, field_args=None, field_kwargs=None, empty_value='')` [source]

Asserts that a form field behaves correctly with various inputs.

Parameters:

- **fieldclass** – the class of the field to be tested.
- **valid** – a dictionary mapping valid inputs to their expected cleaned values.
- **invalid** – a dictionary mapping invalid inputs to one or more raised error messages.
- **field_args** – the args passed to instantiate the field.
- **field_kwargs** – the kwargs passed to instantiate the field.
- **empty_value** – the expected clean output for inputs in `empty_values` .

For example, the following code tests that an `EmailField` accepts `a@a.com` as a valid email address, but rejects `aaa` with a reasonable error message:

```
self.assertFieldOutput(
    EmailField, {"a@a.com": "a@a.com"}, {"aaa": ["Enter a valid email address."]}
)
```

`SimpleTestCase.assertFormError(form, field, errors, msg_prefix='')` [source]

Asserts that a field on a form raises the provided list of errors.

`form` is a `Form` instance. The form must be bound but not necessarily validated (`assertFormError()` will automatically call `full_clean()` on the form).

`field` is the name of the field on the form to check. To check the form's `non-field errors`, use `field=None` .

`errors` is a list of all the error strings that the field is expected to have. You can also pass a single error string if you only expect one error which means that `errors='error message'` is the same as `errors=['error message']` .

`SimpleTestCase.assertFormSetError(formset, form_index, field, errors, msg_prefix='')` [source]

Asserts that the `formset` raises the provided list of errors when rendered.

`formset` is a `FormSet` instance. The formset must be bound but not necessarily validated (`assertFormSetError()` will automatically call the `full_clean()` on the formset).

`form_index` is the number of the form within the `FormSet` (starting from 0). Use `form_index=None` to check the formset's non-form errors, i.e. the errors you get when calling `formset.non_form_errors()` . In that case you must also use `field=None` .

`field` and `errors` have the same meaning as the parameters to `assertFormError()` .

`SimpleTestCase.assertContains(response, text, count=None, status_code=200, msg_prefix='', html=False)` [source]

Asserts that a `response` produced the given `status_code` and that `text` appears in its `content`. If `count` is provided, `text` must occur exactly `count` times in the response.

Set `html` to `True` to handle `text` as HTML. The comparison with the response content will be based on HTML semantics instead of character-by-character equality. Whitespace is ignored in most cases, attribute ordering is not significant. See `assertHTMLEqual()` for more details.

> **Changed in Django 5.1:**
> In older versions, error messages didn't contain the response content.

`SimpleTestCase.assertNotContains(response, text, status_code=200, msg_prefix='', html=False)` [source]

Asserts that a `response` produced the given `status_code` and that `text` does *not* appear in its `content`.

Set `html` to `True` to handle `text` as HTML. The comparison with the response content will be based on HTML semantics instead of character-by-character equality. Whitespace is ignored in most cases, attribute ordering is not significant. See `assertHTMLEqual()` for more details.

> **Changed in Django 5.1:**
> In older versions, error messages didn't contain the response content.

`SimpleTestCase.assertTemplateUsed(response, template_name, msg_prefix='', count=None)` [source]

Asserts that the template with the given name was used in rendering the response.

`response` must be a response instance returned by the `test client`.

`template_name` should be a string such as `'admin/index.html'` .

The `count` argument is an integer indicating the number of times the template should be rendered. Default is `None` , meaning that the template should be rendered one or more times.

You can use this as a context manager, like this:

```
with self.assertTemplateUsed("index.html"):
    render_to_string("index.html")
with self.assertTemplateUsed(template_name="index.html"):
    render_to_string("index.html")
```

`SimpleTestCase.assertTemplateNotUsed(response, template_name, msg_prefix='')` [source]

Asserts that the template with the given name was *not* used in rendering the response.

You can use this as a context manager in the same way as `assertTemplateUsed()`.

`SimpleTestCase.assertURLEqual(url1, url2, msg_prefix='')` [source]

Asserts that two URLs are the same, ignoring the order of query string parameters except for parameters with the same name. For example, `/path/?x=1&y=2` is equal to `/path/?y=2&x=1`, but `/path/?a=1&a=2` isn't equal to `/path/?a=2&a=1`.

`SimpleTestCase.assertRedirects(response, expected_url, status_code=302, target_status_code=200, msg_prefix='', fetch_redirect_response=True)` [source]

Asserts that the `response` returned a `status_code` redirect status, redirected to `expected_url` (including any `GET` data), and that the final page was received with `target_status_code`.

If your request used the `follow` argument, the `expected_url` and `target_status_code` will be the url and status code for the final point of the redirect chain.

If `fetch_redirect_response` is `False`, the final page won't be loaded. Since the test client can't fetch external URLs, this is particularly useful if `expected_url` isn't part of your Django app.

Scheme is handled correctly when making comparisons between two URLs. If there isn't any scheme specified in the location where we are redirected to, the original request's scheme is used. If present, the scheme in `expected_url` is the one used to make the comparisons to.

`SimpleTestCase.assertHTMLEqual(html1, html2, msg=None)` [source]

Asserts that the strings `html1` and `html2` are equal. The comparison is based on HTML semantics. The comparison takes following things into account:

- Whitespace before and after HTML tags is ignored.
- All types of whitespace are considered equivalent.
- All open tags are closed implicitly, e.g. when a surrounding tag is closed or the HTML document ends.
- Empty tags are equivalent to their self-closing version.
- The ordering of attributes of an HTML element is not significant.
- Boolean attributes (like `checked`) without an argument are equal to attributes that equal in name and value (see the examples).
- Text, character references, and entity references that refer to the same character are equivalent.

The following examples are valid tests and don't raise any `AssertionError`:

```
self.assertHTMLEqual(
    "<p>Hello <b>&#x27;world&#x27;!</p>",
    """<p>
        Hello   <b>&#39;world&#39;! </b>
    </p>""",
)
self.assertHTMLEqual(
    '<input type="checkbox" checked="checked" id="id_accept_terms" />',
    '<input id="id_accept_terms" type="checkbox" checked>',
)
```

`html1` and `html2` must contain HTML. An `AssertionError` will be raised if one of them cannot be parsed.

Output in case of error can be customized with the `msg` argument.

`SimpleTestCase.assertHTMLNotEqual(html1, html2, msg=None)` [source]

Asserts that the strings `html1` and `html2` are *not* equal. The comparison is based on HTML semantics. See `assertHTMLEqual()` for details.

`html1` and `html2` must contain HTML. An `AssertionError` will be raised if one of them cannot be parsed.

Output in case of error can be customized with the `msg` argument.

`SimpleTestCase.assertXMLEqual(xml1, xml2, msg=None)` [source]

Asserts that the strings `xml1` and `xml2` are equal. The comparison is based on XML semantics. Similarly to `assertHTMLEqual()`, the comparison is made on parsed content, hence only semantic differences are considered, not syntax differences. When invalid XML is passed in any parameter, an `AssertionError` is always raised, even if both strings are identical.

XML declaration, document type, processing instructions, and comments are ignored. Only the root element and its children are compared.

Output in case of error can be customized with the `msg` argument.

`SimpleTestCase.assertXMLNotEqual(xml1, xml2, msg=None)` [source]

Asserts that the strings `xml1` and `xml2` are *not* equal. The comparison is based on XML semantics. See `assertXMLEqual()` for details.

Output in case of error can be customized with the `msg` argument.

`SimpleTestCase.assertInHTML(needle, haystack, count=None, msg_prefix='')` [source]

Asserts that the HTML fragment `needle` is contained in the `haystack` once.

If the `count` integer argument is specified, then additionally the number of `needle` occurrences will be strictly verified.

Whitespace in most cases is ignored, and attribute ordering is not significant. See `assertHTMLEqual()` for more details.

> **Changed in Django 5.1:**
> In older versions, error messages didn't contain the `haystack`.

`SimpleTestCase.assertNotInHTML(needle, haystack, msg_prefix='')` [source]

> **New in Django 5.1.**

Asserts that the HTML fragment `needle` is *not* contained in the `haystack`.

Whitespace in most cases is ignored, and attribute ordering is not significant. See `assertHTMLEqual()` for more details.

`SimpleTestCase.assertJSONEqual(raw, expected_data, msg=None)` [source]

Asserts that the JSON fragments `raw` and `expected_data` are equal. Usual JSON non-significant whitespace rules apply as the heavyweight is delegated to the `json` library.

Output in case of error can be customized with the `msg` argument.

`SimpleTestCase.assertJSONNotEqual(raw, expected_data, msg=None)` [source]

Asserts that the JSON fragments `raw` and `expected_data` are *not* equal. See `assertJSONEqual()` for further details.

Output in case of error can be customized with the `msg` argument.

`TransactionTestCase.assertQuerySetEqual(qs, values, transform=None, ordered=True, msg=None)` [source]

Asserts that a queryset `qs` matches a particular iterable of values `values`.

If `transform` is provided, `values` is compared to a list produced by applying `transform` to each member of `qs`.

By default, the comparison is also ordering dependent. If `qs` doesn't provide an implicit ordering, you can set the `ordered` parameter to `False`, which turns the comparison into a `collections.Counter` comparison. If the order is undefined (if the given `qs` isn't ordered and the comparison is against more than one ordered value), a `ValueError` is raised.

Output in case of error can be customized with the `msg` argument.

`TransactionTestCase.assertNumQueries(num, func, *args, **kwargs)` [source]

Asserts that when `func` is called with `*args` and `**kwargs` that `num` database queries are executed.

If a `"using"` key is present in `kwargs` it is used as the database alias for which to check the number of queries:

```
self.assertNumQueries(7, using="non_default_db")
```

If you wish to call a function with a `using` parameter you can do it by wrapping the call with a `lambda` to add an extra parameter:

```
self.assertNumQueries(7, lambda: my_function(using=7))
```

You can also use this as a context manager:

```
with self.assertNumQueries(2):
    Person.objects.create(name="Aaron")
    Person.objects.create(name="Daniel")
```

Tagging tests

You can tag your tests so you can easily run a particular subset. For example, you might label fast or slow tests:

```
from django.test import tag


class SampleTestCase(TestCase):
    @tag("fast")
    def test_fast(self): ...

    @tag("slow")
    def test_slow(self): ...

    @tag("slow", "core")
    def test_slow_but_core(self): ...
```

You can also tag a test case class:

```
@tag("slow", "core")
class SampleTestCase(TestCase): ...
```

Subclasses inherit tags from superclasses, and methods inherit tags from their class. Given:

```
@tag("foo")
class SampleTestCaseChild(SampleTestCase):
    @tag("bar")
    def test(self): ...
```

`SampleTestCaseChild.test` will be labeled with `'slow'`, `'core'`, `'bar'`, and `'foo'`.

Then you can choose which tests to run. For example, to run only fast tests:

```
$ ./manage.py test --tag=fast
```

```
...\> manage.py test --tag=fast
```

Or to run fast tests and the core one (even though it's slow):

```
$ ./manage.py test --tag=fast --tag=core
```

```
...\> manage.py test --tag=fast --tag=core
```

You can also exclude tests by tag. To run core tests if they are not slow:

```
$ ./manage.py test --tag=core --exclude-tag=slow
```

```
...\> manage.py test --tag=core --exclude-tag=slow
```

`test --exclude-tag` has precedence over `test --tag`, so if a test has two tags and you select one of them and exclude the other, the test won't be run.

Testing asynchronous code

If you merely want to test the output of your asynchronous views, the standard test client will run them inside their own asynchronous loop without any extra work needed on your part.

However, if you want to write fully-asynchronous tests for a Django project, you will need to take several things into account.

Firstly, your tests must be `async def` methods on the test class (in order to give them an asynchronous context). Django will automatically detect any `async def` tests and wrap them so they run in their own event loop.

If you are testing from an asynchronous function, you must also use the asynchronous test client. This is available as `django.test.AsyncClient`, or as `self.async_client` on any test.

```
class AsyncClient(enforce_csrf_checks=False, raise_request_exception=True, *, headers=None, query_params=None, **defaults)  [source]
```

`AsyncClient` has the same methods and signatures as the synchronous (normal) test client, with the following exceptions:

- In the initialization, arbitrary keyword arguments in `defaults` are added directly into the ASGI scope.
- Headers passed as `extra` keyword arguments should not have the `HTTP_` prefix required by the synchronous client (see `Client.get()`). For example, here is how to set an HTTP `Accept` header:

```
>>> c = AsyncClient()
>>> c.get("/customers/details/", {"name": "fred", "age": 7}, ACCEPT="application/json")
```

Changed in Django 5.0:
Support for the `follow` parameter was added to the `AsyncClient`.

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

Using `AsyncClient` any method that makes a request must be awaited:

```
async def test_my_thing(self):
    response = await self.async_client.get("/some-url/")
    self.assertEqual(response.status_code, 200)
```

The asynchronous client can also call synchronous views; it runs through Django's asynchronous request path, which supports both. Any view called through the `AsyncClient` will get an `ASGIRequest` object for its `request` rather than the `WSGIRequest` that the normal client creates.

Warning: If you are using test decorators, they must be async-compatible to ensure they work correctly. Django's built-in decorators will behave correctly, but third-party ones may appear to not execute (they will "wrap" the wrong part of the execution flow and not your test).

If you need to use these decorators, then you should decorate your test methods with `async_to_sync()` *inside* of them instead:

```
from asgiref.sync import async_to_sync
from django.test import TestCase


class MyTests(TestCase):
    @mock.patch(...)
    @async_to_sync
    async def test_my_thing(self): ...
```

Email services

If any of your Django views send email using Django's email functionality, you probably don't want to send email each time you run a test using that view. For this reason, Django's test runner automatically redirects all Django-sent email to a dummy outbox. This lets you test every aspect of sending email – from the number of messages sent to the contents of each message – without actually sending the messages.

The test runner accomplishes this by transparently replacing the normal email backend with a testing backend. (Don't worry – this has no effect on any other email senders outside of Django, such as your machine's mail server, if you're running one.)

```
django.core.mail.outbox
```

During test running, each outgoing email is saved in `django.core.mail.outbox`. This is a list of all `EmailMessage` instances that have been sent. The `outbox` attribute is a special attribute that is created *only* when the `locmem` email backend is used. It doesn't normally exist as part of the `django.core.mail` module and you can't import it directly. The code below shows how to access this attribute correctly.

Here's an example test that examines `django.core.mail.outbox` for length and contents:

```
from django.core import mail
from django.test import TestCase


class EmailTest(TestCase):
    def test_send_email(self):
        # Send message.
        mail.send_mail(
```

```
            "Subject here",
            "Here is the message.",
            "from@example.com",
            ["to@example.com"],
            fail_silently=False,
        )

        # Test that one message has been sent.
        self.assertEqual(len(mail.outbox), 1)

        # Verify that the subject of the first message is correct.
        self.assertEqual(mail.outbox[0].subject, "Subject here")
```

As noted previously, the test outbox is emptied at the start of every test in a Django `*TestCase`. To empty the outbox manually, assign the empty list to `mail.outbox`:

```
from django.core import mail

# Empty the test outbox
mail.outbox = []
```

Management Commands

Management commands can be tested with the `call_command()` function. The output can be redirected into a `StringIO` instance:

```
from io import StringIO
from django.core.management import call_command
from django.test import TestCase


class ClosepollTest(TestCase):
    def test_command_output(self):
        out = StringIO()
        call_command("closepoll", poll_ids=[1], stdout=out)
        self.assertIn('Successfully closed poll "1"', out.getvalue())
```

Skipping tests

The unittest library provides the `@skipIf` and `@skipUnless` decorators to allow you to skip tests if you know ahead of time that those tests are going to fail under certain conditions.

For example, if your test requires a particular optional library in order to succeed, you could decorate the test case with `@skipIf`. Then, the test runner will report that the test wasn't executed and why, instead of failing the test or omitting the test altogether.

To supplement these test skipping behaviors, Django provides two additional skip decorators. Instead of testing a generic boolean, these decorators check the capabilities of the database, and skip the test if the database doesn't support a specific named feature.

The decorators use a string identifier to describe database features. This string corresponds to attributes of the database connection features class. See django.db.backends.base.features.BaseDatabaseFeatures class for a full list of database features that can be used as a basis for skipping tests.

`skipIfDBFeature(*feature_name_strings)` [source]

Skip the decorated test or `TestCase` if all of the named database features are supported.

For example, the following test will not be executed if the database supports transactions (e.g., it would *not* run under PostgreSQL, but it would under MySQL with MyISAM tables):

```
class MyTests(TestCase):
    @skipIfDBFeature("supports_transactions")
    def test_transaction_behavior(self):
        # ... conditional test code
        pass
```

`skipUnlessDBFeature(*feature_name_strings)` [source]

Skip the decorated test or `TestCase` if any of the named database features are *not* supported.

For example, the following test will only be executed if the database supports transactions (e.g., it would run under PostgreSQL, but *not* under MySQL with MyISAM tables):

```
class MyTests(TestCase):
    @skipUnlessDBFeature("supports_transactions")
    def test_transaction_behavior(self):
        # ... conditional test code
        pass
```