**Tutorial Part 10:**

**Testing a Django web application**

As websites grow they become harder to test manually. Not only is there more to test, but, as interactions between components become more complex, a small change in one area can impact other areas, so more changes will be required to ensure everything keeps working and errors are not introduced as more changes are made. One way to mitigate these problems is to write automated tests, which can easily and reliably be run every time you make a change. This tutorial shows how to automate *unit testing* of your website using Django's test framework.

|  |  |
| --- | --- |
| **Prerequisites:** | Complete all previous tutorial topics, including Tutorial Part-9: Working with forms. |
| **Objective:** | To understand how to write unit tests for Django-based websites. |

**Overview**

The Local Library currently has pages to display lists of all books and authors, detail views for Book and Author items, a page to renew BookInstances, and pages to create, update, and delete Author items (and Book records too, if you completed the *challenge* in the forms tutorial). Even with this relatively small site, manually navigating to each page and *superficially* checking that everything works as expected can take several minutes. As we make changes and grow the site, the time required to manually check that everything works "properly" will only grow. If we were to continue as we are, eventually we'd be spending most of our time testing, and very little time improving our code.

Automated tests can really help with this problem! The obvious benefits are that they can be run much faster than manual tests, can test to a much lower level of detail, and test exactly the same functionality every time (human testers are nowhere near as reliable!) Because they are fast, automated tests can be executed more regularly, and if a test fails, they point to exactly where code is not performing as expected.

In addition, automated tests can act as the first real-world "user" of your code, forcing you to be rigorous about defining and documenting how your website should behave. Often they are the basis for your code examples and documentation. For these reasons, some software development processes start with test definition and implementation, after which the code is written to match the required behavior (e.g. [test-driven](https://en.wikipedia.org/wiki/Test-driven_development) and [behaviour-driven](https://en.wikipedia.org/wiki/Behavior-driven_development) development).

This tutorial shows how to write automated tests for Django, by adding a number of tests to the *LocalLibrary* website.

Types of testing

There are numerous types, levels, and classifications of tests and testing approaches. The most important automated tests are:

**Unit tests**

Verify functional behavior of individual components, often to class and function level.

**Regression tests**

Tests that reproduce historic bugs. Each test is initially run to verify that the bug has been fixed, and then re-run to ensure that it has not been reintroduced following later changes to the code.

**Integration tests**

Verify how groupings of components work when used together. Integration tests are aware of the required interactions between components, but not necessarily of the internal operations of each component. They may cover simple groupings of components through to the whole website.

**Note:**Other common types of tests include black box, white box, manual, automated, canary, smoke, conformance, acceptance, functional, system, performance, load, and stress tests. Look them up for more information.

What does Django provide for testing?

Testing a website is a complex task, because it is made of several layers of logic – from HTTP-level request handling, queries models, to form validation and processing, and template rendering.

Django provides a test framework with a small hierarchy of classes that build on the Python standard [unittest](https://docs.python.org/3/library/unittest.html%22%20%5Cl%20%22module-unittest%22%20%5Co%20%22%28in%20Python%20v3.5%29) library. Despite the name, this test framework is suitable for both unit and integration tests. The Django framework adds API methods and tools to help test web and Django-specific behaviour. These allow you to simulate requests, insert test data, and inspect your application's output. Django also provides an API ([LiveServerTestCase](https://docs.djangoproject.com/en/2.1/topics/testing/tools/%22%20%5Cl%20%22liveservertestcase)) and tools for [using different testing frameworks](https://docs.djangoproject.com/en/2.1/topics/testing/advanced/#other-testing-frameworks), for example you can integrate with the popular [Selenium](https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Cross_browser_testing/Your_own_automation_environment) framework to simulate a user interacting with a live browser.

To write a test you derive from any of the Django (or *unittest*) test base classes ([SimpleTestCase](https://docs.djangoproject.com/en/2.1/topics/testing/tools/#simpletestcase), [TransactionTestCase](https://docs.djangoproject.com/en/2.1/topics/testing/tools/#transactiontestcase), [TestCase](https://docs.djangoproject.com/en/2.1/topics/testing/tools/#testcase), [LiveServerTestCase](https://docs.djangoproject.com/en/2.1/topics/testing/tools/#liveservertestcase)) and then write separate methods to check that specific functionality works as expected (tests use "assert" methods to test that expressions result in True or False values, or that two values are equal, etc.) When you start a test run, the framework executes the chosen test methods in your derived classes. The test methods are run independently, with common setup and/or tear-down behaviour defined in the class, as shown below.

class YourTestClass(TestCase):

    def setUp(self):

  # Setup run before every test method.

        pass

    def tearDown(self):

  # Clean up run after every test method.

        pass

    def test\_something\_that\_will\_pass(self):

        self.assertFalse(False)

    def test\_something\_that\_will\_fail(self):

        self.assertTrue(False)

The best base class for most tests is [django.test.TestCase](https://docs.djangoproject.com/en/2.1/topics/testing/tools/%22%20%5Cl%20%22testcase). This test class creates a clean database before its tests are run, and runs every test function in its own transaction. The class also owns a test [Client](https://docs.djangoproject.com/en/2.1/topics/testing/tools/#django.test.Client) that you can use to simulate a user interacting with the code at the view level. In the following sections we're going to concentrate on unit tests, created using this [TestCase](https://docs.djangoproject.com/en/2.1/topics/testing/tools/%22%20%5Cl%20%22testcase) base class.

**Note:** The [django.test.TestCase](https://docs.djangoproject.com/en/2.1/topics/testing/tools/%22%20%5Cl%20%22testcase) class is very convenient, but may result in some tests being slower than they need to be (not every test will need to set up its own database or simulate the view interaction). Once you're familiar with what you can do with this class, you may want to replace some of your tests with the available simpler test classes.

What should you test?

You should test all aspects of your own code, but not any libraries or functionality provided as part of Python or Django.

So for example, consider the Author model defined below. You don't need to explicitly test that first\_name and last\_name have been stored properly as CharField in the database because that is something defined by Django (though of course in practice you will inevitably test this functionality during development). Nor do you need to test that the date\_of\_birth has been validated to be a date field, because that is again something implemented in Django.

However you should check the text used for the labels (*First name, Last name, Date of birth, Died*), and the size of the field allocated for the text (*100 chars*), because these are part of your design and something that could be broken/changed in future.

class Author(models.Model):

    first\_name = models.CharField(max\_length=100)

    last\_name = models.CharField(max\_length=100)

    date\_of\_birth = models.DateField(null=True, blank=True)

    date\_of\_death = models.DateField('Died', null=True, blank=True)

    def get\_absolute\_url(self):

        return reverse('author-detail', args=[str(self.id)])

    def \_\_str\_\_(self):

        return '%s, %s' % (self.last\_name, self.first\_name)

Similarly, you should check that the custom methods get\_absolute\_url() and \_\_str\_\_() behave as required because they are your code/business logic. In the case of get\_absolute\_url() you can trust that the Django reverse() method has been implemented properly, so what you're testing is that the associated view has actually been defined.

**Note:** Astute readers may note that we would also want to constrain the date of birth and death to sensible values, and check that death comes after birth. In Django this constraint would be added to your form classes (although you can define validators for model fields and model validators these are only used at the form level if they are called by the model's clean() method. This requires a ModelForm or the model's clean() method needs to be specifically called.)

With that in mind let's start looking at how to define and run tests.

**Test structure overview**

Before we go into the detail of "what to test", let's first briefly look at *where* and *how* tests are defined.

Django uses the unittest module’s [built-in test discovery](https://docs.python.org/3/library/unittest.html#unittest-test-discovery), which will discover tests under the current working directory in any file named with the pattern **test\*.py**. Provided you name the files appropriately, you can use any structure you like. We recommend that you create a module for your test code, and have separate files for models, views, forms, and any other types of code you need to test. For example:

catalog/

 /tests/

 \_\_init\_\_.py

 test\_models.py

 test\_forms.py

 test\_views.py

Create a file structure as shown above in your *LocalLibrary* project. The **\_\_init\_\_.py** should be an empty file (this tells Python that the directory is a package). You can create the three test files by copying and renaming the skeleton test file **/catalog/tests.py**.

**Note:** The skeleton test file **/catalog/tests.py** was created automatically when we [built the Django skeleton website](https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/skeleton_website). It is perfectly "legal" to put all your tests inside it, but if you test properly, you'll quickly end up with a very large and unmanageable test file.

Delete the skeleton file as we won't need it.

Open **/catalog/tests/test\_models.py**. The file should import django.test.TestCase, as shown:

from django.test import TestCase

# Create your tests here.

Often you will add a test class for each model/view/form you want to test, with individual methods for testing specific functionality. In other cases you may wish to have a separate class for testing a specific use case, with individual test functions that test aspects of that use-case (for example, a class to test that a model field is properly validated, with functions to test each of the possible failure cases). Again, the structure is very much up to you, but it is best if you are consistent.

Add the test class below to the bottom of the file. The class demonstrates how to construct a test case class by deriving from TestCase.

class YourTestClass(TestCase):

    @classmethod

    def setUpTestData(cls):

        print("setUpTestData: Run once to set up non-modified data for all class methods.")

        pass

    def setUp(self):

        print("setUp: Run once for every test method to setup clean data.")

        pass

    def test\_false\_is\_false(self):

        print("Method: test\_false\_is\_false.")

        self.assertFalse(False)

    def test\_false\_is\_true(self):

        print("Method: test\_false\_is\_true.")

        self.assertTrue(False)

    def test\_one\_plus\_one\_equals\_two(self):

        print("Method: test\_one\_plus\_one\_equals\_two.")

        self.assertEqual(1 + 1, 2)

The new class defines two methods that you can use for pre-test configuration (for example, to create any models or other objects you will need for the test):

* setUpTestData() is called once at the beginning of the test run for class-level setup. You'd use this to create objects that aren't going to be modified or changed in any of the test methods.
* setUp() is called before every test function to set up any objects that may be modified by the test (every test function will get a "fresh" version of these objects).

The test classes also have a tearDown() method which we haven't used. This method isn't particularly useful for database tests, since the TestCase base class takes care of database teardown for you.

Below those we have a number of test methods, which use Assert functions to test whether conditions are true, false or equal (AssertTrue, AssertFalse, AssertEqual). If the condition does not evaluate as expected then the test will fail and report the error to your console.

The AssertTrue, AssertFalse, AssertEqual are standard assertions provided by **unittest**. There are other standard assertions in the framework, and also [Django-specific assertions](https://docs.djangoproject.com/en/2.1/topics/testing/tools/#assertions) to test if a view redirects (assertRedirects), to test if a particular template has been used (assertTemplateUsed), etc.

You should **not**normally include **print()** functions in your tests as shown above. We do that here only so that you can see the order that the setup functions are called in the console (in the following section).

**How to run the tests**

The easiest way to run all the tests is to use the command:

python3 manage.py test

This will discover all files named with the pattern **test\*.py** under the current directory and run all tests defined using appropriate base classes (here we have a number of test files, but only **/catalog/tests/test\_models.py** currently contains any tests.) By default the tests will individually report only on test failures, followed by a test summary.

If you get errors similar to: ValueError: Missing staticfiles manifest entry ... this may be because testing does not run *collectstatic* by default and your app is using a storage class that requires it (see [manifest\_strict](https://docs.djangoproject.com/en/2.1/ref/contrib/staticfiles/%22%20%5Cl%20%22django.contrib.staticfiles.storage.ManifestStaticFilesStorage.manifest_strict) for more information). There are a number of ways you can overcome this problem - the easiest is to simply run *collectstatic* before running the tests:

python3 manage.py collectstatic

Run the tests in the root directory of *LocalLibrary*. You should see an output like the one below.

> python3 manage.py test

Creating test database for alias 'default'...

setUpTestData: Run once to set up non-modified data for all class methods.

setUp: Run once for every test method to setup clean data.

Method: test\_false\_is\_false.

setUp: Run once for every test method to setup clean data.

Method: test\_false\_is\_true.

setUp: Run once for every test method to setup clean data.

Method: test\_one\_plus\_one\_equals\_two.

.

======================================================================

FAIL: test\_false\_is\_true (catalog.tests.tests\_models.YourTestClass)

----------------------------------------------------------------------

Traceback (most recent call last):

  File "D:\Github\django\_tmp\library\_w\_t\_2\locallibrary\catalog\tests\tests\_models.py", line 22, in test\_false\_is\_true

    self.assertTrue(False)

AssertionError: False is not true

----------------------------------------------------------------------

Ran 3 tests in 0.075s

FAILED (failures=1)

Destroying test database for alias 'default'...

Here we see that we had one test failure, and we can see exactly what function failed and why (this failure is expected, because False is not True!).

**Tip:**The most important thing to learn from the test output above is that it is much more valuable if you use descriptive/informative names for your objects and methods.

The text shown in **bold** above would not normally appear in the test output (this is generated by the print() functions in our tests). This shows how the setUpTestData() method is called once for the class and setUp() is called before each method.

The next sections show how you can run specific tests, and how to control how much information the tests display.

Showing more test information

If you want to get more information about the test run you can change the *verbosity*. For example, to list the test successes as well as failures (and a whole bunch of information about how the testing database is set up) you can set the verbosity to "2" as shown:

python3 manage.py test --verbosity 2

The allowed verbosity levels are 0, 1, 2, and 3, with the default being "1".

Running specific tests

If you want to run a subset of your tests you can do so by specifying the full dot path to the package(s), module, TestCase subclass or method:

# Run the specified module

python3 manage.py test catalog.tests

# Run the specified module

python3 manage.py test catalog.tests.test\_models

# Run the specified class

python3 manage.py test catalog.tests.test\_models.YourTestClass

# Run the specified method

python3 manage.py test catalog.tests.test\_models.YourTestClass.test\_one\_plus\_one\_equals\_two

**LocalLibrary tests**

Now we know how to run our tests and what sort of things we need to test, let's look at some practical examples.

**Note:**We won't write every possible test, but this should give you an idea of how tests work, and what more you can do.

Models

As discussed above, we should test anything that is part of our design or that is defined by code that we have written, but not libraries/code that is already tested by Django or the Python development team.

For example, consider the Author model below. Here we should test the labels for all the fields, because even though we haven't explicitly specified most of them, we have a design that says what these values should be. If we don't test the values, then we don't know that the field labels have their intended values. Similarly, while we trust that Django will create a field of the specified length, it is worthwhile to specify a test for this length to ensure that it was implemented as planned.

class Author(models.Model):

    first\_name = models.CharField(max\_length=100)

    last\_name = models.CharField(max\_length=100)

    date\_of\_birth = models.DateField(null=True, blank=True)

    date\_of\_death = models.DateField('Died', null=True, blank=True)

    def get\_absolute\_url(self):

        return reverse('author-detail', args=[str(self.id)])

    def \_\_str\_\_(self):

        return f'{self.last\_name}, {self.first\_name}'

Open our **/catalog/tests/test\_models.py**, and replace any existing code with the following test code for the Author model.

Here you'll see that we first import TestCase and derive our test class (AuthorModelTest) from it, using a descriptive name so we can easily identify any failing tests in the test output. We then call setUpTestData() to create an author object that we will use but not modify in any of the tests.

from django.test import TestCase

from catalog.models import Author

class AuthorModelTest(TestCase):

 @classmethod

 def setUpTestData(cls):

 # Set up non-modified objects used by all test methods

 Author.objects.create(first\_name='Big', last\_name='Bob')

 def test\_first\_name\_label(self):

 author = Author.objects.get(id=1)

 field\_label = author.\_meta.get\_field('first\_name').verbose\_name

 self.assertEqual(field\_label, 'first name')

 def test\_date\_of\_death\_label(self):

 author=Author.objects.get(id=1)

 field\_label = author.\_meta.get\_field('date\_of\_death').verbose\_name

 self.assertEqual(field\_label, 'died')

 def test\_first\_name\_max\_length(self):

 author = Author.objects.get(id=1)

 max\_length = author.\_meta.get\_field('first\_name').max\_length

 self.assertEqual(max\_length, 100)

 def test\_object\_name\_is\_last\_name\_comma\_first\_name(self):

 author = Author.objects.get(id=1)

 expected\_object\_name = f'{author.last\_name}, {author.first\_name}'

 self.assertEqual(expected\_object\_name, str(author))

 def test\_get\_absolute\_url(self):

 author = Author.objects.get(id=1)

 # This will also fail if the urlconf is not defined.

 self.assertEqual(author.get\_absolute\_url(), '/catalog/author/1')

The field tests check that the values of the field labels (verbose\_name) and that the size of the character fields are as expected. These methods all have descriptive names, and follow the same pattern:

# Get an author object to test

author = Author.objects.get(id=1)

# Get the metadata for the required field and use it to query the required field data

field\_label = author.\_meta.get\_field('first\_name').verbose\_name

# Compare the value to the expected result

self.assertEqual(field\_label, 'first name')

The interesting things to note are:

* We can't get the verbose\_name directly using author.first\_name.verbose\_name, because author.first\_name is a *string* (not a handle to the first\_name object that we can use to access its properties). Instead we need to use the author's \_meta attribute to get an instance of the field and use that to query for the additional information.
* We chose to use assertEqual(field\_label,'first name') rather than assertTrue(field\_label == 'first name'). The reason for this is that if the test fails the output for the former tells you what the label actually was, which makes debugging the problem just a little easier.

**Note:** Tests for the last\_name and date\_of\_birth labels, and also the test for the length of the last\_name field have been omitted. Add your own versions now, following the naming conventions and approaches shown above.

We also need to test our custom methods. These essentially just check that the object name was constructed as we expected using "Last Name", "First Name" format, and that the URL we get for an Author item is as we would expect.

def test\_object\_name\_is\_last\_name\_comma\_first\_name(self):

    author = Author.objects.get(id=1)

    expected\_object\_name = f'{author.last\_name}, {author.first\_name}'

    self.assertEqual(expected\_object\_name, str(author))

def test\_get\_absolute\_url(self):

    author = Author.objects.get(id=1)

    # This will also fail if the urlconf is not defined.

    self.assertEqual(author.get\_absolute\_url(), '/catalog/author/1')

Run the tests now. If you created the Author model as we described in the models tutorial it is quite likely that you will get an error for the date\_of\_death label as shown below. The test is failing because it was written expecting the label definition to follow Django's convention of not capitalising the first letter of the label (Django does this for you).

======================================================================

FAIL: test\_date\_of\_death\_label (catalog.tests.test\_models.AuthorModelTest)

----------------------------------------------------------------------

Traceback (most recent call last):

 File "D:\...\locallibrary\catalog\tests\test\_models.py", line 32, in test\_date\_of\_death\_label

 self.assertEqual(field\_label,'died')

AssertionError: 'Died' != 'died'

- Died

? ^

+ died

? ^

This is a very minor bug, but it does highlight how writing tests can more thoroughly check any assumptions you may have made.

**Note:**Change the label for the date\_of\_death field (/catalog/models.py) to "died" and re-run the tests.

The patterns for testing the other models are similar so we won't continue to discuss these further. Feel free to create your own tests for our other models.

Forms

The philosophy for testing your forms is the same as for testing your models; you need to test anything that you've coded or your design specifies, but not the behavior of the underlying framework and other third party libraries.

Generally this means that you should test that the forms have the fields that you want, and that these are displayed with appropriate labels and help text. You don't need to verify that Django validates the field type correctly (unless you created your own custom field and validation) — i.e. you don't need to test that an email field only accepts emails. However, you would need to test any additional validation that you expect to be performed on the fields and any messages that your code will generate for errors.

Consider our form for renewing books. This has just one field for the renewal date, which will have a label and help text that we will need to verify.

class RenewBookForm(forms.Form):

 """Form for a librarian to renew books."""

 renewal\_date = forms.DateField(help\_text="Enter a date between now and 4 weeks (default 3).")

 def clean\_renewal\_date(self):

 data = self.cleaned\_data['renewal\_date']

 # Check if a date is not in the past.

 if data < datetime.date.today():

 raise ValidationError(\_('Invalid date - renewal in past'))

 # Check if date is in the allowed range (+4 weeks from today).

 if data > datetime.date.today() + datetime.timedelta(weeks=4):

 raise ValidationError(\_('Invalid date - renewal more than 4 weeks ahead'))

 # Remember to always return the cleaned data.

 return data

Open our **/catalog/tests/test\_forms.py** file and replace any existing code with the following test code for the RenewBookForm form. We start by importing our form and some Python and Django libraries to help test time-related functionality. We then declare our form test class in the same way as we did for models, using a descriptive name for our TestCase-derived test class.

import datetime

from django.test import TestCase

from django.utils import timezone

from catalog.forms import RenewBookForm

class RenewBookFormTest(TestCase):

    def test\_renew\_form\_date\_field\_label(self):

        form = RenewBookForm()

        self.assertTrue(form.fields['renewal\_date'].label == None or form.fields['renewal\_date'].label == 'renewal date')

    def test\_renew\_form\_date\_field\_help\_text(self):

        form = RenewBookForm()

        self.assertEqual(form.fields['renewal\_date'].help\_text, 'Enter a date between now and 4 weeks (default 3).')

    def test\_renew\_form\_date\_in\_past(self):

        date = datetime.date.today() - datetime.timedelta(days=1)

        form = RenewBookForm(data={'renewal\_date': date})

        self.assertFalse(form.is\_valid())

    def test\_renew\_form\_date\_too\_far\_in\_future(self):

        date = datetime.date.today() + datetime.timedelta(weeks=4) + datetime.timedelta(days=1)

        form = RenewBookForm(data={'renewal\_date': date})

        self.assertFalse(form.is\_valid())

    def test\_renew\_form\_date\_today(self):

        date = datetime.date.today()

        form = RenewBookForm(data={'renewal\_date': date})

        self.assertTrue(form.is\_valid())

    def test\_renew\_form\_date\_max(self):

        date = timezone.localtime() + datetime.timedelta(weeks=4)

        form = RenewBookForm(data={'renewal\_date': date})

        self.assertTrue(form.is\_valid())

The first two functions test that the field's label and help\_text are as expected. We have to access the field using the fields dictionary (e.g. form.fields['renewal\_date']). Note here that we also have to test whether the label value is None, because even though Django will render the correct label it returns None if the value is not *explicitly* set.

The rest of the functions test that the form is valid for renewal dates just inside the acceptable range and invalid for values outside the range. Note how we construct test date values around our current date (datetime.date.today()) using datetime.timedelta() (in this case specifying a number of days or weeks). We then just create the form, passing in our data, and test if it is valid.

**Note:** Here we don't actually use the database or test client. Consider modifying these tests to use [SimpleTestCase](https://docs.djangoproject.com/en/2.1/topics/testing/tools/%22%20%5Cl%20%22django.test.SimpleTestCase).

We also need to validate that the correct errors are raised if the form is invalid, however this is usually done as part of view processing, so we'll take care of that in the next section.

That's all for forms; we do have some others, but they are automatically created by our generic class-based editing views, and should be tested there! Run the tests and confirm that our code still passes!

Views

To validate our view behaviour we use the Django test [Client](https://docs.djangoproject.com/en/2.1/topics/testing/tools/#django.test.Client). This class acts like a dummy web browser that we can use to simulate GET and POST requests on a URL and observe the response. We can see almost everything about the response, from low-level HTTP (result headers and status codes) through to the template we're using to render the HTML and the context data we're passing to it. We can also see the chain of redirects (if any) and check the URL and status code at each step. This allows us to verify that each view is doing what is expected.

Let's start with one of our simplest views, which provides a list of all Authors. This is displayed at URL **/catalog/authors/** (an URL named 'authors' in the URL configuration).

class AuthorListView(generic.ListView):

 model = Author

 paginate\_by = 10

As this is a generic list view almost everything is done for us by Django. Arguably if you trust Django then the only thing you need to test is that the view is accessible at the correct URL and can be accessed using its name. However if you're using a test-driven development process you'll start by writing tests that confirm that the view displays all Authors, paginating them in lots of 10.

Open the **/catalog/tests/test\_views.py** file and replace any existing text with the following test code for AuthorListView. As before we import our model and some useful classes. In the setUpTestData() method we set up a number of Author objects so that we can test our pagination.

from django.test import TestCase

from django.urls import reverse

from catalog.models import Author

class AuthorListViewTest(TestCase):

    @classmethod

    def setUpTestData(cls):

        # Create 13 authors for pagination tests

        number\_of\_authors = 13

        for author\_id in range(number\_of\_authors):

            Author.objects.create(

 first\_name=f'Christian {author\_id}',

 last\_name=f'Surname {author\_id}',

 )

    def test\_view\_url\_exists\_at\_desired\_location(self):

        response = self.client.get('/catalog/authors/')

        self.assertEqual(response.status\_code, 200)

    def test\_view\_url\_accessible\_by\_name(self):

        response = self.client.get(reverse('authors'))

        self.assertEqual(response.status\_code, 200)

    def test\_view\_uses\_correct\_template(self):

        response = self.client.get(reverse('authors'))

        self.assertEqual(response.status\_code, 200)

        self.assertTemplateUsed(response, 'catalog/author\_list.html')

    def test\_pagination\_is\_ten(self):

        response = self.client.get(reverse('authors'))

        self.assertEqual(response.status\_code, 200)

        self.assertTrue('is\_paginated' in response.context)

        self.assertTrue(response.context['is\_paginated'] == True)

        self.assertTrue(len(response.context['author\_list']) == 10)

    def test\_lists\_all\_authors(self):

        # Get second page and confirm it has (exactly) remaining 3 items

        response = self.client.get(reverse('authors')+'?page=2')

        self.assertEqual(response.status\_code, 200)

        self.assertTrue('is\_paginated' in response.context)

        self.assertTrue(response.context['is\_paginated'] == True)

        self.assertTrue(len(response.context['author\_list']) == 3)

All the tests use the client (belonging to our TestCase's derived class) to simulate a GET request and get a response. The first version checks a specific URL (note, just the specific path without the domain) while the second generates the URL from its name in the URL configuration.

response = self.client.get('/catalog/authors/')

response = self.client.get(reverse('authors'))

Once we have the response we query it for its status code, the template used, whether or not the response is paginated, the number of items returned, and the total number of items.

Note: If you set the paginate\_by variable in your **/catalog/views.py** file to a number other than 10, make sure to update the lines that test that the correct number of items are displayed in paginated templates above and in following sections. For example, if you set the variable for the author list page to 5, update the line above to:

self.assertTrue(len(response.context['author\_list']) == 5)

The most interesting variable we demonstrate above is response.context, which is the context variable passed to the template by the view. This is incredibly useful for testing, because it allows us to confirm that our template is getting all the data it needs. In other words we can check that we're using the intended template and what data the template is getting, which goes a long way to verifying that any rendering issues are solely due to template.

**Views that are restricted to logged in users**

In some cases you'll want to test a view that is restricted to just logged in users. For example our LoanedBooksByUserListView is very similar to our previous view but is only available to logged in users, and only displays BookInstance records that are borrowed by the current user, have the 'on loan' status, and are ordered "oldest first".

from django.contrib.auth.mixins import LoginRequiredMixin

class LoanedBooksByUserListView(LoginRequiredMixin, generic.ListView):

 """Generic class-based view listing books on loan to current user."""

 model = BookInstance

 template\_name ='catalog/bookinstance\_list\_borrowed\_user.html'

 paginate\_by = 10

 def get\_queryset(self):

 return BookInstance.objects.filter(borrower=self.request.user).filter(status\_\_exact='o').order\_by('due\_back')

Add the following test code to **/catalog/tests/test\_views.py**. Here we first use SetUp() to create some user login accounts and BookInstance objects (along with their associated books and other records) that we'll use later in the tests. Half of the books are borrowed by each test user, but we've initially set the status of all books to "maintenance". We've used SetUp() rather than setUpTestData() because we'll be modifying some of these objects later.

**Note:** The setUp() code below creates a book with a specified Language, but *your* code may not include the Language model as this was created as a *challenge*. If this is the case, simply comment out the parts of the code that create or import Language objects. You should also do this in the RenewBookInstancesViewTest section that follows.

import datetime

from django.utils import timezone

from django.contrib.auth.models import User # Required to assign User as a borrower

from catalog.models import BookInstance, Book, Genre, Language

class LoanedBookInstancesByUserListViewTest(TestCase):

    def setUp(self):

        # Create two users

        test\_user1 = User.objects.create\_user(username='testuser1', password='1X<ISRUkw+tuK')

        test\_user2 = User.objects.create\_user(username='testuser2', password='2HJ1vRV0Z&3iD')

 test\_user1.save()

        test\_user2.save()

        # Create a book

        test\_author = Author.objects.create(first\_name='John', last\_name='Smith')

        test\_genre = Genre.objects.create(name='Fantasy')

        test\_language = Language.objects.create(name='English')

        test\_book = Book.objects.create(

 title='Book Title',

 summary='My book summary',

 isbn='ABCDEFG',

 author=test\_author,

 language=test\_language,

 )

 # Create genre as a post-step

        genre\_objects\_for\_book = Genre.objects.all()

  test\_book.genre.set(genre\_objects\_for\_book) # Direct assignment of many-to-many types not allowed.

        test\_book.save()

        # Create 30 BookInstance objects

        number\_of\_book\_copies = 30

        for book\_copy in range(number\_of\_book\_copies):

            return\_date = timezone.localtime() + datetime.timedelta(days=book\_copy%5)

            the\_borrower = test\_user1 if book\_copy % 2 else test\_user2

            status = 'm'

            BookInstance.objects.create(

 book=test\_book,

 imprint='Unlikely Imprint, 2016',

 due\_back=return\_date,

 borrower=the\_borrower,

 status=status,

 )

    def test\_redirect\_if\_not\_logged\_in(self):

        response = self.client.get(reverse('my-borrowed'))

        self.assertRedirects(response, '/accounts/login/?next=/catalog/mybooks/')

    def test\_logged\_in\_uses\_correct\_template(self):

        login = self.client.login(username='testuser1', password='1X<ISRUkw+tuK')

        response = self.client.get(reverse('my-borrowed'))

        # Check our user is logged in

        self.assertEqual(str(response.context['user']), 'testuser1')

        # Check that we got a response "success"

        self.assertEqual(response.status\_code, 200)

        # Check we used correct template

        self.assertTemplateUsed(response, 'catalog/bookinstance\_list\_borrowed\_user.html')

To verify that the view will redirect to a login page if the user is not logged in we use assertRedirects, as demonstrated in test\_redirect\_if\_not\_logged\_in(). To verify that the page is displayed for a logged in user we first log in our test user, and then access the page again and check that we get a status\_code of 200 (success).

The rest of the tests verify that our view only returns books that are on loan to our current borrower. Copy the code below and paste it onto the end of the test class above.

    def test\_only\_borrowed\_books\_in\_list(self):

        login = self.client.login(username='testuser1', password='1X<ISRUkw+tuK')

        response = self.client.get(reverse('my-borrowed'))

        # Check our user is logged in

        self.assertEqual(str(response.context['user']), 'testuser1')

        # Check that we got a response "success"

        self.assertEqual(response.status\_code, 200)

        # Check that initially we don't have any books in list (none on loan)

        self.assertTrue('bookinstance\_list' in response.context)

        self.assertEqual(len(response.context['bookinstance\_list']), 0)

        # Now change all books to be on loan

        books = BookInstance.objects.all()[:10]

        for book in books:

            book.status = 'o'

            book.save()

        # Check that now we have borrowed books in the list

        response = self.client.get(reverse('my-borrowed'))

        # Check our user is logged in

        self.assertEqual(str(response.context['user']), 'testuser1')

        # Check that we got a response "success"

        self.assertEqual(response.status\_code, 200)

        self.assertTrue('bookinstance\_list' in response.context)

        # Confirm all books belong to testuser1 and are on loan

        for bookitem in response.context['bookinstance\_list']:

            self.assertEqual(response.context['user'], bookitem.borrower)

            self.assertEqual('o', bookitem.status)

    def test\_pages\_ordered\_by\_due\_date(self):

        # Change all books to be on loan

        for book in BookInstance.objects.all():

            book.status='o'

            book.save()

        login = self.client.login(username='testuser1', password='1X<ISRUkw+tuK')

        response = self.client.get(reverse('my-borrowed'))

        # Check our user is logged in

        self.assertEqual(str(response.context['user']), 'testuser1')

        # Check that we got a response "success"

        self.assertEqual(response.status\_code, 200)

        # Confirm that of the items, only 10 are displayed due to pagination.

        self.assertEqual(len(response.context['bookinstance\_list']), 10)

        last\_date = 0

        for book in response.context['bookinstance\_list']:

            if last\_date == 0:

                last\_date = book.due\_back

            else:

 self.assertTrue(last\_date <= book.due\_back)

                last\_date = book.due\_back

You could also add pagination tests, should you so wish!

**Testing views with forms**

Testing views with forms is a little more complicated than in the cases above, because you need to test more code paths: initial display, display after data validation has failed, and display after validation has succeeded. The good news is that we use the client for testing in almost exactly the same way as we did for display-only views.

To demonstrate, let's write some tests for the view used to renew books (renew\_book\_librarian()):

from catalog.forms import RenewBookForm

@permission\_required('catalog.can\_mark\_returned')

def renew\_book\_librarian(request, pk):

    """View function for renewing a specific BookInstance by librarian."""

    book\_instance = get\_object\_or\_404(BookInstance, pk=pk)

    # If this is a POST request then process the Form data

    if request.method == 'POST':

        # Create a form instance and populate it with data from the request (binding):

        book\_renewal\_form = RenewBookForm(request.POST)

        # Check if the form is valid:

        if form.is\_valid():

            # process the data in form.cleaned\_data as required (here we just write it to the model due\_back field)

            book\_instance.due\_back = form.cleaned\_data['renewal\_date']

            book\_instance.save()

            # redirect to a new URL:

            return HttpResponseRedirect(reverse('all-borrowed'))

    # If this is a GET (or any other method) create the default form

    else:

        proposed\_renewal\_date = datetime.date.today() + datetime.timedelta(weeks=3)

        book\_renewal\_form = RenewBookForm(initial={'renewal\_date': proposed\_renewal\_date})

 context = {

 'book\_renewal\_form': book\_renewal\_form,

 'book\_instance': book\_instance,

 }

    return render(request, 'catalog/book\_renew\_librarian.html', context)

We'll need to test that the view is only available to users who have the can\_mark\_returned permission, and that users are redirected to an HTTP 404 error page if they attempt to renew a BookInstance that does not exist. We should check that the initial value of the form is seeded with a date three weeks in the future, and that if validation succeeds we're redirected to the "all-borrowed books" view. As part of checking the validation-fail tests we'll also check that our form is sending the appropriate error messages.

Add the first part of the test class (shown below) to the bottom of **/catalog/tests/test\_views.py**. This creates two users and two book instances, but only gives one user the permission required to access the view. The code to grant permissions during tests is shown in bold:

import uuid

from django.contrib.auth.models import Permission # Required to grant the permission needed to set a book as returned.

class RenewBookInstancesViewTest(TestCase):

 def setUp(self):

 # Create a user

 test\_user1 = User.objects.create\_user(username='testuser1', password='1X<ISRUkw+tuK')

 test\_user2 = User.objects.create\_user(username='testuser2', password='2HJ1vRV0Z&3iD')

 test\_user1.save()

 test\_user2.save()

 **permission = Permission.objects.get(name='Set book as returned')**

 **test\_user2.user\_permissions.add(permission)**

 **test\_user2.save()**

 # Create a book

 test\_author = Author.objects.create(first\_name='John', last\_name='Smith')

 test\_genre = Genre.objects.create(name='Fantasy')

 test\_language = Language.objects.create(name='English')

 test\_book = Book.objects.create(

 title='Book Title',

 summary='My book summary',

 isbn='ABCDEFG',

 author=test\_author,

 language=test\_language,

 )

 # Create genre as a post-step

 genre\_objects\_for\_book = Genre.objects.all()

 test\_book.genre.set(genre\_objects\_for\_book) # Direct assignment of many-to-many types not allowed.

 test\_book.save()

 # Create a BookInstance object for test\_user1

 return\_date = datetime.date.today() + datetime.timedelta(days=5)

 self.test\_bookinstance1 = BookInstance.objects.create(

 book=test\_book,

 imprint='Unlikely Imprint, 2016',

 due\_back=return\_date,

 borrower=test\_user1,

 status='o',

 )

 # Create a BookInstance object for test\_user2

 return\_date = datetime.date.today() + datetime.timedelta(days=5)

 self.test\_bookinstance2 = BookInstance.objects.create(

 book=test\_book,

 imprint='Unlikely Imprint, 2016',

 due\_back=return\_date,

 borrower=test\_user2,

 status='o',

 )

Add the following tests to the bottom of the test class. These check that only users with the correct permissions (*testuser2*) can access the view. We check all the cases: when the user is not logged in, when a user is logged in but does not have the correct permissions, when the user has permissions but is not the borrower (should succeed), and what happens when they try to access a BookInstance that doesn't exist. We also check that the correct template is used.

 def test\_redirect\_if\_not\_logged\_in(self):

        response = self.client.get(reverse('renew-book-librarian', kwargs={'pk': self.test\_bookinstance1.pk}))

        # Manually check redirect (Can't use assertRedirect, because the redirect URL is unpredictable)

        self.assertEqual(response.status\_code, 302)

        self.assertTrue(response.url.startswith('/accounts/login/'))

    def test\_redirect\_if\_logged\_in\_but\_not\_correct\_permission(self):

        login = self.client.login(username='testuser1', password='1X<ISRUkw+tuK')

        response = self.client.get(reverse('renew-book-librarian', kwargs={'pk': self.test\_bookinstance1.pk}))

        self.assertEqual(response.status\_code, 403)

    def test\_logged\_in\_with\_permission\_borrowed\_book(self):

        login = self.client.login(username='testuser2', password='2HJ1vRV0Z&3iD')

        response = self.client.get(reverse('renew-book-librarian', kwargs={'pk': self.test\_bookinstance2.pk}))

        # Check that it lets us login - this is our book and we have the right permissions.

        self.assertEqual(response.status\_code, 200)

    def test\_logged\_in\_with\_permission\_another\_users\_borrowed\_book(self):

        login = self.client.login(username='testuser2', password='2HJ1vRV0Z&3iD')

        response = self.client.get(reverse('renew-book-librarian', kwargs={'pk': self.test\_bookinstance1.pk}))

        # Check that it lets us login. We're a librarian, so we can view any users book

        self.assertEqual(response.status\_code, 200)

    def test\_HTTP404\_for\_invalid\_book\_if\_logged\_in(self):

 # unlikely UID to match our bookinstance!

        test\_uid = uuid.uuid4()

        login = self.client.login(username='testuser2', password='2HJ1vRV0Z&3iD')

        response = self.client.get(reverse('renew-book-librarian', kwargs={'pk':test\_uid}))

        self.assertEqual(response.status\_code, 404)

    def test\_uses\_correct\_template(self):

        login = self.client.login(username='testuser2', password='2HJ1vRV0Z&3iD')

        response = self.client.get(reverse('renew-book-librarian', kwargs={'pk': self.test\_bookinstance1.pk}))

        self.assertEqual(response.status\_code, 200)

        # Check we used correct template

        self.assertTemplateUsed(response, 'catalog/book\_renew\_librarian.html')

Add the next test method, as shown below. This checks that the initial date for the form is three weeks in the future. Note how we are able to access the value of the initial value of the form field (shown in bold).

    def test\_form\_renewal\_date\_initially\_has\_date\_three\_weeks\_in\_future(self):

        login = self.client.login(username='testuser2', password='2HJ1vRV0Z&3iD')

        response = self.client.get(reverse('renew-book-librarian', kwargs={'pk': self.test\_bookinstance1.pk}))

        self.assertEqual(response.status\_code, 200)

        date\_3\_weeks\_in\_future = datetime.date.today() + datetime.timedelta(weeks=3)

        self.assertEqual(response**.context['form'].initial['renewal\_date'], date\_3\_weeks\_in\_future**)

If you use the form class RenewBookModelForm(forms.ModelForm) instead of class RenewBookForm(forms.Form), then the form field name is **'due\_back'**instead of **'renewal\_date'**.

The next test (add this to the class too) checks that the view redirects to a list of all borrowed books if renewal succeeds. What differs here is that for the first time we show how you can POST data using the client. The post *data* is the second argument to the post function, and is specified as a dictionary of key/values.

    def test\_redirects\_to\_all\_borrowed\_book\_list\_on\_success(self):

        login = self.client.login(username='testuser2', password='2HJ1vRV0Z&3iD')

        valid\_date\_in\_future = datetime.date.today() + datetime.timedelta(weeks=2)

        **response = self.client.post(reverse('renew-book-librarian', kwargs={'pk':self.test\_bookinstance1.pk,}), {'renewal\_date':valid\_date\_in\_future})**

**self.assertRedirects(response, reverse('all-borrowed'))**

The *all-borrowed* view was added as a *challenge*, and your code may instead redirect to the home page '/'. If so, modify the last two lines of the test code to be like the code below. The follow=True in the request ensures that the request returns the final destination URL (hence checking /catalog/ rather than /).

 response = self.client.post(reverse('renew-book-librarian', kwargs={'pk':self.test\_bookinstance1.pk,}), {'renewal\_date':valid\_date\_in\_future}, **follow=True** )

 **self.assertRedirects(response, '/catalog/')**

Copy the last two functions into the class, as seen below. These again test POST requests, but in this case with invalid renewal dates. We use assertFormError() to verify that the error messages are as expected.

    def test\_form\_invalid\_renewal\_date\_past(self):

        login = self.client.login(username='testuser2', password='2HJ1vRV0Z&3iD')

        date\_in\_past = datetime.date.today() - datetime.timedelta(weeks=1)

        response = self.client.post(reverse('renew-book-librarian', kwargs={'pk': self.test\_bookinstance1.pk}), {'renewal\_date': date\_in\_past})

        self.assertEqual(response.status\_code, 200)

**self.assertFormError(response, 'form', 'renewal\_date', 'Invalid date - renewal in past')**

    def test\_form\_invalid\_renewal\_date\_future(self):

        login = self.client.login(username='testuser2', password='2HJ1vRV0Z&3iD')

        invalid\_date\_in\_future = datetime.date.today() + datetime.timedelta(weeks=5)

        response = self.client.post(reverse('renew-book-librarian', kwargs={'pk': self.test\_bookinstance1.pk}), {'renewal\_date': invalid\_date\_in\_future})

        self.assertEqual(response.status\_code, 200)

        **self.assertFormError(response, 'form', 'renewal\_date', 'Invalid date - renewal more than 4 weeks ahead')**

The same sorts of techniques can be used to test the other view.

Templates

Django provides test APIs to check that the correct template is being called by your views, and to allow you to verify that the correct information is being sent. There is however no specific API support for testing in Django that your HTML output is rendered as expected.

**Other recommended test tools**

Django's test framework can help you write effective unit and integration tests — we've only scratched the surface of what the underlying **unittest** framework can do, let alone Django's additions (for example, check out how you can use [unittest.mock](https://docs.python.org/3.5/library/unittest.mock-examples.html) to patch third party libraries so you can more thoroughly test your own code).

While there are numerous other test tools that you can use, we'll just highlight two:

* [Coverage](http://coverage.readthedocs.io/en/latest/): This Python tool reports on how much of your code is actually executed by your tests. It is particularly useful when you're getting started, and you are trying to work out exactly what you should test.
* [Selenium](https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Cross_browser_testing/Your_own_automation_environment) is a framework to automate testing in a real browser. It allows you to simulate a real user interacting with the site, and provides a great framework for system testing your site (the next step up from integration testing).

**Challenge yourself**

There are a lot more models and views we can test. As a simple task, try to create a test case for the AuthorCreate view.

class AuthorCreate(PermissionRequiredMixin, CreateView):

 model = Author

 fields = '\_\_all\_\_'

 initial = {'date\_of\_death':'12/10/2016'}

 permission\_required = 'catalog.can\_mark\_returned'

Remember that you need to check anything that you specify or that is part of the design. This will include who has access, the initial date, the template used, and where the view redirects on success.

**Summary**

Writing test code is neither fun nor glamorous, and is consequently often left to last (or not at all) when creating a website. It is however an essential part of making sure that your code is safe to release after making changes, and cost-effective to maintain.

In this tutorial we've shown you how to write and run tests for your models, forms, and views. Most importantly we've provided a brief summary of what you should test, which is often the hardest thing to work out when you're getting started. There is a lot more to know, but even with what you've learned already you should be able to create effective unit tests for your websites.

The next and final tutorial shows how you can deploy your wonderful (and fully tested!) Django website.