

This page was translated from English by the community. Learn more and join the MDN Web Docs community.

# Tutorial de Django Parte 10: Probando una aplicación web Django

A medida que crecen los sitios web se vuelven más difíciles de probar a mano — no sólo hay más para probar, sino que además, a medida que las interacciones entre los componentes se vuelven más complejas, un pequeño cambio en un área puede suponer muchas pruebas adicionales para verificar su impacto en otras áreas. Una forma de mitigar estos problemas es escribir tests automatizados, que pueden ser ejecutados de manera fácil y fiable cada vez que hagas un cambio. Este tutorial muestra cómo automatizar la unidad de pruebas de tu sitio web usando el framework de pruebas de Django.

Prerequisites:	Completa todos los tópicos anteriores, incluyendo <a href="#">Tutorial Django Parte 9: Trabajando con formularios</a> .
Objective:	Entender como escribir pruebas unidatarias para django basado en Páginas web.

## Vista previa

El [Local Library](#) actualmente tiene páginas para mostrar las listas con todos los libros y autores, vistas detalladas para los items de `Book` y `Author`, una página para renovar `BookInstances` y páginas para crear, actualizar y eliminar elementos de autor (y también registros de libros, si usted completó el desafío en el tutorial de formularios). Incluso con este sitio relativamente pequeño, navegar manualmente a cada página y verificar superficialmente que todo funcione como se espera, puede llevar varios minutos. A

medida que hagamos cambios y el sitio vaya creciendo, el tiempo requerido para verificar manualmente que todo funcione "correctamente", aumentará de forma muy perniciosa. Si continuamos como estamos, pasaríamos la mayor parte de nuestro tiempo probando, y muy poco tiempo mejorando nuestro código.

¡Las pruebas automatizadas realmente pueden ayudar con este problema! Los beneficios obvios son que pueden ejecutarse mucho más rápido que las pruebas manuales, pueden probar con un nivel de detalle mucho más bajo y probar exactamente la misma funcionalidad cada vez (¡los testers humanos no son tan confiables!) Porque son pruebas rápidas y automatizadas se puede ejecutar más regularmente, y si falla una prueba, señalan exactamente dónde el código no está funcionando como se esperaba.

Además, las pruebas automatizadas pueden actuar como el primer "usuario" del mundo real de su código, lo que le obliga a ser riguroso a la hora de definir y documentar bien, cómo debe comportarse su sitio web. A menudo son la base de sus ejemplos de código y documentación. Por estas razones, algunos procesos de desarrollo de software comienzan con la definición e implementación de la prueba, después de lo cual el código se escribe para que coincida con el comportamiento requerido (por ejemplo, desarrollo basado en pruebas y en comportamiento).

Este tutorial muestra cómo escribir pruebas automatizadas para Django, agregando una serie de pruebas al sitio web LocalLibrary.

## Tipos de pruebas

Hay numeroso tipos, niveles y clasificaciones de pruebas y enfoques de pruebas. Las pruebas automáticas más importantes son:

### Pruebas unitarias

Verifica el comportamiento funcional de un componente individual, a menudo de una clase y su nivel de funcional.

### Pruebas de regresión

Pruebas que reproducen errores históricos. Cada prueba es inicialmente ejecutada para verificar que el error ha sido corregido, y estos son ejecutados de nuevo para

asegurarnos que los errores no fueron reintroducidos con los futuros cambios en el código.

## Pruebas de integración

Verifica cómo funcionan los grupos de componentes cuando se usan juntos. Las pruebas de integración son conscientes de las interacciones requeridas entre componentes, pero no necesariamente de las operaciones internas de cada componente. Pueden cubrir agrupaciones simples de componentes hasta todo el sitio web.

**Nota:** Otros tipos comunes de pruebas incluyen pruebas de caja negra, caja blanca, manuales, automatizadas, canarias, de humo, de conformidad, de aceptación, funcionales, de rendimiento, de carga y de esfuerzo. Búscalos para más información.

## Que provee Django para pruebas?

Probar un sitio web es una tarea compleja, porque está compuesto por varias capas de lógica, desde el manejo de solicitudes a nivel HTTP, modelos de consultas, hasta la validación y procesamiento de formularios y la representación de plantillas.

Django proporciona un marco de prueba con una pequeña jerarquía de clases que se basan en la librería [unittest](#) estándar Python. A pesar del nombre, este marco de prueba es adecuado tanto para pruebas unitarias como de integración. El marco de Django agrega métodos y herramientas API para ayudar a probar el comportamiento web y específico de Django. Estos le permiten simular solicitudes, insertar datos de prueba e inspeccionar la salida de su aplicación. Django también proporciona una API([LiveServerTestCase](#)) y herramientas para [usar diferentes frameworks de pruebas](#), por ejemplo, puede integrarse con el popular framework [Selenium \(en-US\)](#) para simular la interacción de un usuario con un navegador en vivo.

Para escribir una prueba, se deriva de cualquiera de las clases base de prueba de Django (o unittest)([SimpleTestCase](#) , [TransactionTestCase](#) , [TestCase](#) , [LiveServerTestCase](#) ) y luego escribir métodos separados para verificar que la funcionalidad específica funcione como se esperaba (las pruebas usan métodos "assert" para probar que las expresiones

dan valores `True` o `False`, o que dos valores son iguales, etc.) Cuando inicia una ejecución de prueba, el marco ejecuta los métodos de prueba elegidos en sus clases derivadas. Los métodos de prueba se ejecutan de forma independiente, con un comportamiento común de configuración y / o desmontaje definido en la clase, como se muestra a continuación.

```
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)
```

La mejor clase base para la mayoría de las pruebas es [django.test.TestCase](#). Esta clase de prueba crea una base de datos limpia antes de que se ejecuten sus pruebas y ejecuta cada función de prueba en su propia transacción. La clase también posee una prueba [Client](#) que puede utilizar para simular la interacción de un usuario con el código en el nivel de vista. En las siguientes secciones, nos concentraremos en las pruebas unitarias, creadas con esta clase [TestCase](#)

**Nota:** La clase [django.test.TestCase](#) es muy conveniente, pero puede resultar en que algunas pruebas sean más lentas de lo necesario (no todas las pruebas necesitarán configurar su propia base de datos o simular la interacción de la vista). Una vez que esté familiarizado con lo que puede hacer con esta clase, es posible que desee reemplazar algunas de sus pruebas con las clases de prueba más simples disponibles.

Que deberías probar?

Debe probar todos los aspectos de su propio código, pero no ninguna biblioteca o funcionalidad proporcionada como parte de Python o Django.

Por ejemplo, considere el modelo `Author` definido abajo. No es necesario probarlo explícitamente `first_name` y `last_name` han sido almacenados correctamente como `CharField` en la base de datos porque eso es algo definido por Django (aunque, por supuesto, en la práctica, inevitablemente probará esta funcionalidad durante el desarrollo). Tampoco es necesario probar que el `date_of_birth` ha sido validado para ser un campo de fecha, porque nuevamente es algo implementado en Django.

Sin embargo, debe verificar el texto utilizado para las etiquetas (nombre, apellido, fecha de nacimiento, fallecimiento) y el tamaño del campo asignado para el texto (100 caracteres), porque estos son parte de su diseño y algo que podría ser roto / cambiado en el futuro.

```
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)
```

Del mismo modo, debe verificar que los métodos personalizados `get_absolute_url()` y `__str__()` comportarse como sea necesario porque son su código / lógica empresarial. En el caso de `get_absolute_url()` puedes confiar en que el método de Django `reverse()` se ha implementado correctamente, por lo que lo que está probando es que la vista asociada se haya definido realmente.

**Nota:** Los lectores astutos pueden notar que también queríamos restringir la fecha de nacimiento y muerte a valores sensibles, y comprobar que la muerte viene después del nacimiento. En Django, esta restricción se agregaría a sus

clases de formulario (aunque puede definir validadores para los campos, estos parecen usarse solo en el nivel del formulario, no en el nivel del modelo).

Con eso en mente, comenzemos a ver cómo definir y ejecutar pruebas.

## Descripción general de la estructura de prueba

Antes de entrar en los detalles de "qué probar", primero veamos brevemente dónde y cómo se definen las pruebas..

Django utiliza el descubrimiento de pruebas integrado del módulo `unittest` ([built-in test](#))



---

forma adecuada, puede utilizar la estructura que desee. Le recomendamos que cree un módulo para su código de prueba y que tenga archivos separados para modelos, vistas, formularios y cualquier otro tipo de código que necesite probar. Por ejemplo:

```
catalog/  
  /tests/  
    __init__.py  
    test_models.py  
    test_forms.py  
    test_views.py
```

Cree una estructura de archivo como se muestra arriba en su proyecto `LocalLibrary`. El `__init__.py` debe ser un archivo vacío (esto le dice a Python que el directorio es un paquete). Puede crear los tres archivos de prueba copiando y cambiando el nombre del archivo de prueba de esqueleto `/catalog/tests.py`.

**Nota:** El archivo de prueba `/catalog/tests.py` se creó automáticamente cuando creamos el sitio web esqueleto de Django ([built the Django skeleton website](#)). Es perfectamente "legal" poner todas sus pruebas dentro de él, pero si prueba correctamente, rápidamente terminará con un archivo de prueba muy grande e inmanejable. Elimina el archivo esqueleto ya que no lo necesitaremos.

Abre el archivo `/catalog/tests/test_models.py`. El archivo debe importar

`django.test.TestCase`, como se muestra:

```
from django.test import TestCase

# Create your tests here.
```

A menudo, agregará una clase de prueba para cada modelo / vista / formulario que desee probar, con métodos individuales para probar una funcionalidad específica. En otros casos, es posible que desee tener una clase separada para probar un caso de uso específico, con funciones de prueba individuales que prueben aspectos de ese caso de uso (por ejemplo, una clase para probar que un campo de modelo está validado correctamente, con funciones para probar cada uno de los posibles casos de falla). Una vez más, la estructura depende en gran medida de usted, pero es mejor si es coherente.

Agregue la clase de prueba a continuación al final del archivo. La clase demuestra cómo construir una clase de caso de prueba derivando de `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)
```

La nueva clase define dos métodos que puede utilizar para la configuración previa a la prueba (por ejemplo, para crear modelos u otros objetos que necesitará para la prueba):

- `setUpTestData()` se llama una vez al comienzo de la ejecución de prueba para la configuración a nivel de clase. Usaría esto para crear objetos que no se modificarán ni cambiarán en ninguno de los métodos de prueba.
- `setUp()` se llama antes de cada función de prueba para configurar cualquier objeto que pueda ser modificado por la prueba (cada función de prueba obtendrá una versión "nueva" de estos objetos).

**Nota:** Las clases de prueba también tienen un método `tearDown()` que no hemos utilizado. Este método no es particularmente útil para las pruebas de bases de datos, ya que `TestCase` la clase base se encarga del desmontaje de la base de datos por usted.

Debajo de ellos tenemos una serie de métodos de prueba, que utilizamos funciones `Assert` para probar si las condiciones son verdaderas, falsas o iguales (`assertTrue`, `assertFalse`, `assertEqual`). Si la condición no se evalúa como se esperaba, la prueba fallará y reportará el error a su consola.

Los `assertTrue`, `assertFalse`, `assertEqual` son afirmaciones estándar proporcionadas por `unittest`. Hay otras aserciones estándar en el marco y también aserciones específicas de Django ([Django-specific assertions](#)) para probar si una vista redirecciona (`assertRedirects`), para probar si se ha utilizado una plantilla en particular (`assertTemplateUsed`), etc.

**Nota:** Normalmente no debería incluir funciones `print()` en sus pruebas como se muestra arriba. Lo hacemos aquí solo para que pueda ver el orden en que se llaman las funciones de configuración en la consola (en la siguiente sección).

## Como correr las pruebas

La forma más sencilla de ejecutar todas las pruebas es utilizar el comando:

```
python3 manage.py test
```

Esto descubrirá todos los archivos nombrados con el patrón `test*.py` bajo el directorio actual y ejecute todas las pruebas definidas usando las clases base apropiadas (aquí tenemos una serie de archivos de prueba, pero solo `/catalog/tests/test_models.py` contiene actualmente cualquier prueba). De forma predeterminada, las pruebas informarán individualmente solo sobre las fallas de las pruebas, seguidas de un resumen de la prueba.

**Nota:** Si recibe errores similares a: `ValueError: Missing staticfiles manifest entry ...` esto puede deberse a que las pruebas no ejecutan `collectstatic` de forma predeterminada y su aplicación usa una clase de almacenamiento que lo requiere (consulte `manifest.strict` para obtener más información). Hay varias formas de superar este problema; la más fácil es simplemente ejecutar `collectstatic` antes de ejecutar las pruebas:

```
python3 manage.py collectstatic
```

Ejecute las pruebas en el directorio raíz de LocalLibrary. Debería ver un resultado como el siguiente.

```
>python 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.
FsetUp: 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
https://developer.mozilla.org/es/docs/Learn/Server-side/Django/Testing 9/30
```

```
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'...
```

Aquí vemos que tuvimos una falla de prueba, y podemos ver exactamente qué función falló y por qué (se espera esta falla, porque `False` no es `True`!).

**Nota:** Sugerencia: Lo más importante que debe aprender del resultado de la prueba anterior es que es mucho más valioso si usa nombres descriptivos / informativos para sus objetos y métodos.

El texto que se muestra en **negritas** anterior normalmente no aparecería en la salida de prueba (esto es generado por la función `print()` en nuestra prueba). Esto muestra el método `setUpTestData()` es llamado una vez para la clase y `setUp()` se llama antes de cada método.

Las siguientes secciones muestran cómo puede ejecutar pruebas específicas y cómo controlar cuánta información muestran las pruebas.

## Mostrando más información de las pruebas

Si desea obtener más información sobre la ejecución de prueba, puede cambiar el nivel de detalle. Por ejemplo, para enumerar los éxitos y fallas de la prueba (y una gran cantidad de información sobre cómo está configurada la base de datos de prueba), puede establecer la verbosidad en "2" como se muestra:

```
python3 manage.py test --verbosity 2
```

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

## Ejecutando pruebas específicas

Si desea ejecutar un subconjunto de sus pruebas, puede hacerlo especificando la ruta de puntos completa al paquete (s), módulo, `TestCase` subclase o método:

```
python3 manage.py test catalog.tests    # Run the specified module  
python3 manage.py test catalog.tests.test_models # Run the specified module  
python3 manage.py test catalog.tests.test_models.YourTestClass # Run the specified class  
python3 manage.py test catalog.tests.test_models.YourTestClass.test_one_plus_one_equals_two  
# Run the specified method
```

## Pruebas en el proyecto LocalLibrary

Ahora que sabemos cómo ejecutar nuestras pruebas y qué tipo de cosas necesitamos probar, veamos algunos ejemplos prácticos.

**Nota:** No escribiremos todas las pruebas posibles, pero esto debería darle una idea de cómo funcionan las pruebas y qué más puede hacer.

### Modelos

Como se discutió anteriormente, debemos probar todo lo que sea parte de nuestro diseño o que esté definido por el código que hayamos escrito, pero no las bibliotecas / código que ya haya probado Django o el equipo de desarrollo de Python.

Por ejemplo, considere el modelo de `Author` a continuación. Aquí deberíamos probar las etiquetas para todos los campos, porque aunque no hemos especificado explícitamente la mayoría de ellos, tenemos un diseño que dice cuáles deberían ser estos valores. Si no probamos los valores, entonces no sabemos que las etiquetas de los campos tienen sus valores deseados. De manera similar, aunque confiamos en que Django creará un campo de la longitud especificada, vale la pena especificar una prueba para esta longitud para asegurarse de que se implementó según lo planeado.

```
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)
```

Abra su `/catalog/tests/test_models.py`, y reemplace cualquier código existente con el siguiente código de prueba para el modelo de `Author`.

Aquí usted verá que primero importamos `TestCase` y derivamos nuestras clases de prueba (`AuthorModelTest`) de ello, usando un nombre descriptivo para que así podamos fácilmente cualquier pruebas fallidas en el output de la prueba. Luego llamamos a `setUpTestData()` para crear un objeto de autor que usaremos pero no modificaremos en ninguna de las pruebas.

```
from django.test import TestCase

# Create your tests here.

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 = '%s, %s' % (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:

```

author=Author.objects.get(id=1)    # Get an author object to test
field_label = author._meta.get_field('first_name').verbose_name    # Get the metadata for the
required field and use it to query the required field data
self.assertEqual(field_label,'first name') # Compare the value to the expected result

```

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 `assertEquals(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.

**Nota:** 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 = '%s, %s' % (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.

**Nota:** 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 the our other models.

## Formularios

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 behaviour 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 date is not in past.
        if data < datetime.date.today():
            raise ValidationError(_('Invalid date - renewal in past'))
        #Check date is in range librarian allowed to change (+4 weeks)
        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 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.

```
from django.test import TestCase

# Create your tests here.

import datetime
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_data = {'renewal_date': date}
        form = RenewBookForm(data=form_data)
        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_data = {'renewal_date': date}
        form = RenewBookForm(data=form_data)
        self.assertFalse(form.is_valid())

    def test_renew_form_date_today(self):
        date = datetime.date.today()
        form_data = {'renewal_date': date}
        form = RenewBookForm(data=form_data)
        self.assertTrue(form.is_valid())

    def test_renew_form_date_max(self):
```

```
date = timezone.now() + datetime.timedelta(weeks=4)
form_data = { 'renewal_date': date}
form = RenewBookForm(data=form_data)
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.

**Nota:** Here we don't actually use the database or test client. Consider modifying these tests to use [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!

## Vistas

To validate our view behaviour we use the 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  
  
# Create your tests here.  
  
from catalog.models import Author  
from django.urls import reverse  
  
class AuthorListViewTest(TestCase):  
  
    @classmethod  
    def setUpTestData(cls):  
        #Create 13 authors for pagination tests  
        number_of_authors = 13  
        for author_num in range(number_of_authors):  
            Author.objects.create(first_name='Christian %s' % author_num, last_name =  
'Surname %s' % author_num,)  
  
    def test_view_url_exists_at_desired_location(self):  
        resp = self.client.get('/catalog/authors/')  
        self.assertEqual(resp.status_code, 200)  
  
    def test_view_url_accessible_by_name(self):
```

```

resp = self.client.get(reverse('authors'))
self.assertEqual(resp.status_code, 200)

def test_view_uses_correct_template(self):
    resp = self.client.get(reverse('authors'))
    self.assertEqual(resp.status_code, 200)

    self.assertTemplateUsed(resp, 'catalog/author_list.html')

def test_pagination_is_ten(self):
    resp = self.client.get(reverse('authors'))
    self.assertEqual(resp.status_code, 200)
    self.assertTrue('is_paginated' in resp.context)
    self.assertTrue(resp.context['is_paginated'] == True)
    self.assertTrue(len(resp.context['author_list']) == 10)

def test_lists_all_authors(self):
    #Get second page and confirm it has (exactly) remaining 3 items
    resp = self.client.get(reverse('authors')+'?page=2')
    self.assertEqual(resp.status_code, 200)
    self.assertTrue('is_paginated' in resp.context)
    self.assertTrue(resp.context['is_paginated'] == True)
    self.assertTrue(len(resp.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 (`resp`). 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.

```

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

The most interesting variable we demonstrate above is `resp.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.

**Nota:** 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 catalog.models import BookInstance, Book, Genre, Language
```

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

class LoanedBookInstancesByUserListViewTest(TestCase):

    def setUp(self):
        #Create two users
        test_user1 = User.objects.create_user(username='testuser1', password='12345')
        test_user1.save()
        test_user2 = User.objects.create_user(username='testuser2', password='12345')
        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.now() + datetime.timedelta(days=book_copy%5)
            if book_copy % 2:
                the_borrower=test_user1
            else:
                the_borrower=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):
        resp = self.client.get(reverse('my-borrowed'))
        self.assertRedirects(resp, '/accounts/login/?next=/catalog/mybooks/')

    def test_logged_in_uses_correct_template(self):
        login = self.client.login(username='testuser1', password='12345')
        resp = self.client.get(reverse('my-borrowed'))

        #Check our user is logged in
```

```

self.assertEqual(str(resp.context['user']), 'testuser1')
#Check that we got a response "success"
self.assertEqual(resp.status_code, 200)

#Check we used correct template
self.assertTemplateUsed(resp, '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 test verify that our view only returns books that are on loan to our current borrower. Copy the (self-explanatory) code at the end of the test class above.

```

def test_only_borrowed_books_in_list(self):
    login = self.client.login(username='testuser1', password='12345')
    resp = self.client.get(reverse('my-borrowed'))

    #Check our user is logged in
    self.assertEqual(str(resp.context['user']), 'testuser1')
    #Check that we got a response "success"
    self.assertEqual(resp.status_code, 200)

    #Check that initially we don't have any books in list (none on loan)
    self.assertTrue('bookinstance_list' in resp.context)
    self.assertEqual(len(resp.context['bookinstance_list']),0)

    #Now change all books to be on loan
    get_ten_books = BookInstance.objects.all()[:10]

    for copy in get_ten_books:
        copy.status='o'
        copy.save()

    #Check that now we have borrowed books in the list
    resp = self.client.get(reverse('my-borrowed'))
    #Check our user is logged in
    self.assertEqual(str(resp.context['user']), 'testuser1')
    #Check that we got a response "success"
    self.assertEqual(resp.status_code, 200)

```

```

self.assertTrue('bookinstance_list' in resp.context)

#Confirm all books belong to testuser1 and are on loan
for bookitem in resp.context['bookinstance_list']:
    self.assertEqual(resp.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 copy in BookInstance.objects.all():
        copy.status='o'
        copy.save()

    login = self.client.login(username='testuser1', password='12345')
    resp = self.client.get(reverse('my-borrowed'))

    #Check our user is logged in
    self.assertEqual(str(resp.context['user']), 'testuser1')
    #Check that we got a response "success"
    self.assertEqual(resp.status_code, 200)

    #Confirm that of the items, only 10 are displayed due to pagination.
    self.assertEqual( len(resp.context['bookinstance_list']),10)

    last_date=0
    for copy in resp.context['bookinstance_list']:
        if last_date==0:
            last_date=copy.due_back
        else:
            self.assertTrue(last_date <= copy.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 .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_inst=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):
        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_inst.due_back = form.cleaned_data['renewal_date']
            book_inst.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)
        form = RenewBookForm(initial={'renewal_date': proposed_renewal_date,})

    return render(request, 'catalog/book_renew_librarian.html', {'form': form,
    'bookinst':book_inst})
```

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 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:

```
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='12345')  
        test_user1.save()  
  
        test_user2 = User.objects.create_user(username='testuser2', password='12345')  
        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):
    resp = 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( resp.status_code,302)
    self.assertTrue( resp.url.startswith('/accounts/login/') )

def test_redirect_if_logged_in_but_not_correct_permission(self):
    login = self.client.login(username='testuser1', password='12345')
    resp = 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( resp.status_code,302)
    self.assertTrue( resp.url.startswith('/accounts/login/') )

def test_logged_in_with_permission_borrowed_book(self):
    login = self.client.login(username='testuser2', password='12345')
    resp = 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( resp.status_code,200)

def test_logged_in_with_permission_another_users_borrowed_book(self):
    login = self.client.login(username='testuser2', password='12345')
    resp = 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( resp.status_code,200)

def test_HTTP404_for_invalid_book_if_logged_in(self):
    import uuid

```

```

test_uid = uuid.uuid4() #unlikely UID to match our bookinstance!
login = self.client.login(username='testuser2', password='12345')
resp = self.client.get(reverse('renew-book-librarian', kwargs={'pk':test_uid,})) )
self.assertEqual( resp.status_code,404)

def test_uses_correct_template(self):
    login = self.client.login(username='testuser2', password='12345')
    resp = self.client.get(reverse('renew-book-librarian', kwargs=
{'pk':self.test_bookinstance1.pk,})) )
    self.assertEqual( resp.status_code,200)

    #Check we used correct template
    self.assertTemplateUsed(resp, '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='12345')
    resp = self.client.get(reverse('renew-book-librarian', kwargs=
{'pk':self.test_bookinstance1.pk,})) )
    self.assertEqual( resp.status_code,200)

    date_3_weeks_in_future = datetime.date.today() + datetime.timedelta(weeks=3)
    self.assertEqual(resp.context['form'].initial['renewal_date'],
date_3_weeks_in_future )

```

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='12345')
    valid_date_in_future = datetime.date.today() + datetime.timedelta(weeks=2)
    resp = self.client.post(reverse('renew-book-librarian', kwargs=
{'pk':self.test_bookinstance1.pk,}), {'renewal_date':valid_date_in_future} )
    self.assertRedirects(resp, reverse('all-borrowed') )

```

**Advertencia:** 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 /).

```
resp = self.client.post(reverse('renew-book-librarian', kwargs=
{'pk':self.test_bookinstance1.pk,}), {'renewal_date':valid_date_in_future},
follow=True)
self.assertRedirects(resp, '/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='12345')
    date_in_past = datetime.date.today() - datetime.timedelta(weeks=1)
    resp = self.client.post(reverse('renew-book-librarian', kwargs=
{'pk':self.test_bookinstance1.pk,}), {'renewal_date':date_in_past} )
    self.assertEqual( resp.status_code,200)
    self.assertFormError(resp, 'form', 'renewal_date', 'Invalid date - renewal in past')

def test_form_invalid_renewal_date_future(self):
    login = self.client.login(username='testuser2', password='12345')
    invalid_date_in_future = datetime.date.today() + datetime.timedelta(weeks=5)
    resp = self.client.post(reverse('renew-book-librarian', kwargs=
{'pk':self.test_bookinstance1.pk,}), {'renewal_date':invalid_date_in_future} )
    self.assertEqual( resp.status_code,200)
    self.assertFormError(resp, '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` 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](#) : 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 \(en-US\)](#) 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).

## Reto para mi mismo

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.

## Resumen

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.

## Mirar tambien

- [Writing and running tests](#) (Django docs)
- [Writing your first Django app, part 5 > Introducing automated testing](#) (Django docs)
- [Testing tools reference](#) (Django docs)
- [Advanced testing topics](#) (Django docs)
- [A Guide to Testing in Django](#) (Toast Driven Blog, 2011)
- [Workshop: Test-Driven Web Development with Django](#) (San Diego Python, 2014)
- [Testing in Django \(Part 1\) - Best Practices and Examples](#) (RealPython, 2013)

This page was last modified on 9 mar 2023 by [MDN contributors](#).