INGENIERÍA | DEPARTAMENTO
UNIVERSIDAD DE LOS ANDES | DE COMPUTACIÓN
MÉRIDA VENEZUELA | ESCUELA DE SISTEMAS

Laboratorio de Ingeniería del Software
Práctica Nro. 8
Pruebas Automatizadas

Prof. Solazver Solé
Prep. Yenifer Ramírez
Semestre B-2018

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. Una forma de mitigar estos problemas es escribir tests automatizados, que pueden ser ejecutados de manera fácil y fiable cada vez que hagamos un cambio. Además, las pruebas automatizadas pueden actuar como el primer üsuario", lo que nos obliga a ser rigurosos a la hora de definir y documentar cómo debe comportarse nuestro sitio web. A menudo son base para *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, es por ello que este laboratorio muestra cómo automatizar las pruebas de nuestro sitio web, usando el framework de pruebas de Django.

1. Tipos de Pruebas

Existen numerosos tipos, niveles y clasificaciones de pruebas y enfoques de prueba. Las pruebas automatizadas más importantes son:

1.1. Pruebas unitarias

Verifican el comportamiento funcional de los componentes individuales, a menudo a nivel de clase y función.

1.2. Pruebas de regresión

Son pruebas que reproducen errores históricos. Inicialmente, cada prueba se ejecuta para verificar que el error se ha solucionado y, a continuación, se vuelve a ejecutar para asegurarse de que no se haya vuelto a introducir luego de los cambios posteriores en el código.

1.3. Pruebas de integración

Verifican cómo funcionan las agrupaciones de componentes cuando se usan juntos. Las pruebas de integración son conscientes de las interacciones requeridas entre los componentes, pero no necesariamente de las operaciones internas de cada componente. Pueden cubrir agrupaciones simples de componentes a través de todo el sitio web.
Nota: Otros tipos comunes de pruebas incluyen caja negra, caja blanca, manual, automatizado, canario, humo, conformidad, aceptación, funcional, sistema, rendimiento, carga y pruebas de estrés.

2. ¿Qué aporta Django a las pruebas?

Django proporciona un framework de prueba con una pequeña jerarquía de clases que se basan en la biblioteca de pruebas unitarias estándar de Python. A pesar del nombre, este marco de prueba es adecuado tanto para pruebas unitarias como de integración. El framework de trabajo de Django agrega API de métodos y herramientas para ayudar a probar el comportamiento web y específico de Django. Estos nos permiten simular solicitudes, insertar datos de prueba e inspeccionar el resultado de nuestra aplicación. Django también proporciona una API (`LiveServerTestCase`) y herramientas para usar diferentes framework de prueba, por ejemplo, puede integrarse con el popular framework de *Selenium* para simular que un usuario interactúa con un navegador en vivo.
Para escribir una prueba que derive de cualquiera de las clases base de pruebas de Django utilizamos `SimpleTestCase, TransactionTestCase, TestCase, LiveServerTestCase` y luego escribimos métodos separados para verificar que la funcionalidad específica funciona como se espera (las pruebas usan métodos de prueba para verificar que las expresiones dan como resultado valores Verdaderos o Falsos, o que dos valores son iguales, etc.) Al iniciar una ejecución de prueba, el framework ejecuta los métodos de prueba elegidos en sus clases derivadas. Los métodos de prueba se ejecutan de forma independiente, con una configuración común y/o un comportamiento de desmontaje definidos en la clase, como se muestra:

```
1 class YourTestClass(TestCase):
2
3     def setUp(self):
4     """
5     La configuracion  se ejecuta antes de cada metodo
```

```
6       de prueba.
7       """
8           pass
9
10      def tearDown(self):
11      """
12      Limpia la ejecucion despues de cada metodo
13      de prueba.
14      """
15          pass
16
17      def test_something_that_will_pass(self):
18          self.assertFalse(False)
19
20      def test_something_that_will_fail(self):
21          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 ejecutar sus pruebas y ejecuta cada función de prueba en su propia transacción. La clase también posee un Cliente de prueba que puede usar para simular a un usuario que interactúa con el código en el nivel de visualización, pero puede resultar 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 estemos familiarizado con lo que podemos hacer con esta clase, es posible que deseemos reemplazar algunas de sus pruebas con las clases de pruebas más simples disponibles.

3. ¿Qué debemos probar?

Debemos probar todos los aspectos de nuestro propio código, pero no las bibliotecas o las funciones proporcionadas como parte de Python o Django.
Por ejemplo, consideremos el modelo de `Author` definido por ustedes en el laboratorio anterioir. No es necesario probar explícitamente que el primer nombre y el último nombre se hayan almacenado 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 necesitamos probar que `date_of_birth` ha sido validado para ser un campo de fecha, porque esto es algo implementado en Django.
Sin embargo, si podemos verificar el texto utilizado para las etiquetas (Nombre, Apellido, Fecha de nacimiento, Fallecido) y el tamaño del campo asignado para los textos, ya que son parte de nuestro diseño y algo que podría ser roto y cambiado en el futuro.
Del mismo modo, debemos verificar que los métodos personalizados `get_absolute_url()` y `__str__()` se comporten según sea necesario porque son nuestro código y lógica. Si definimos el método `get_absolute_url()` podemos confiar en que el

método `reverse()` de Django se ha implementado correctamente, por lo que lo que está probando es que la vista asociada se ha definido realmente.
Nota: si somos astutos podemos notar que también es posible querer limitar la fecha de nacimiento a los valores razonables y verificar que la muerte se produce después del nacimiento. En Django, esta restricción se agregaría a sus clases de formulario (aunque puede definir validadores para los campos que parecen ser usados solo en el nivel del formulario, no en el nivel del modelo).

4. Estructura de la Pruebas

Django usa el descubrimiento de prueba incorporado del módulo `unittest`, que descubrirá las pruebas en el directorio de trabajo actual en cualquier archivo nombrado con el patrón **test*.py**. Siempre que nombremos los archivos adecuadamente, podemos usar la estructura que deseamos. Es recomendables que creemos un módulo para nuestro código de prueba y que tenga archivos separados para los modelos, vistas, formularios y cualquier otro tipo de código que necesitemos probar. Podemos crear la siguiente estructura para nuestras pruebas: Podemos agregar en el archivo **test_models.py** una clase de caso de

```
catalog/
└── tests
    ├── __init__.py
    ├── test_forms.py
    ├── test_models.py
    └── test_views.py
```

prueba derivando de TestCase:

```python
from django.test import TestCase

class YourTestClass(TestCase):

    @classmethod
    def setUpTestData(cls):
        print("setUpTestData: se ejecuta una vez para
        configurar datos no modificados para todos los
        metodos de clase.")
        pass

    def setUp(self):
        print("setUp: se ejecuta una vez para cada
        metodo de prueba para configurar datos limpios.")
        pass

    def test_false_is_false(self):
        print("Metodo: test_false_is_false.")
```

```
19          self.assertFalse(False)
20
21      def test_false_is_true(self):
22          print("Metodo: test_false_is_true.")
23          self.assertTrue(False)
24
25      def test_one_plus_one_equals_two(self):
26          print("Metodo: test_one_plus_one_equals_two.")
27          self.assertEqual(1 + 1, 2)
```

Esta clase define dos métodos que podemos usar para la configuración previa a la prueba (por ejemplo, para crear cualquier modelo u otro objeto que necesitáramos para la prueba):

- **`setUpTestData()`:** se llama una vez al comienzo de la ejecución de prueba para la configuración de la clase-nivel. Usaríamos esto para crear objetos que no se modificarán o modificarán 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 la base de datos, ya que la clase base `TestCase` se encarga de la eliminación de la base de datos por nosotros.

Además tenemos varios métodos de prueba, que utilizan funciones `Assert` para comprobar si las condiciones son verdaderas, falsas o iguales (`AssertTrue`, `AssertFalse`, `AssertEqual` [1]). Si la condición no se evalúa como se esperaba, la prueba fallará e informará el error a su consola.

5. Ejecutar los Test

La forma más fácil de ejecutar todas las pruebas es usar el comando:

```
(eV)usuario@pc:tests$ python manage.py test
```

Esto descubrirá todos los archivos nombrados con la prueba de patrón ***.py** bajo el directorio actual y ejecutará todas las pruebas definidas usando las clases base apropiadas. Por defecto, las pruebas informarán individualmente solo sobre las fallas de la prueba, seguidas de un resumen de la prueba.

5.1. Más información en un Test

Si deseamos obtener más información sobre la ejecución de la prueba, podemos cambiar la *verbosidad*. Por ejemplo, para enumerar los éxitos de la prueba, así

[1] Aserciones estándar https://docs.python.org/3/library/unittest.html

como los fallos (y un montón de información sobre cómo se configura la base de datos de prueba), podemos establecer la *verbosidad* en "2":

```
(eV)usuario@pc:tests$ python manage.py test -verbosity 2
```

5.2. Ejecutar un Test específico

Si deseamos ejecutar un subconjunto de nuestras pruebas, podemos hacerlo especificando la ruta completa del paquete, módulos, subclases o métodos de `TestCase`:

```
(eV)usuario@pc:tests$ python manage.py test catalog.tests
```

```
(eV)usuario@pc:tests$ python manage.py test catalog.tests.test_models
```

```
(eV)usuario@pc:tests$ ... test catalog.tests.test_models.YourTestClass
```

```
(eV)usuario@pc:tests$ ... catalog.tests.test_models.YourTestClass.test
```

6. Tests en Proyecto Biblioteca

6.1. Modelos

Como se mencionó anteriormente, deberíamos probar cualquier cosa que sea parte de nuestro diseño o que esté definida por el código que hemos escrito. Por ejemplo, considere el modelo de `author` creado en el laboratorio anterior, aquí deberíamos probar las etiquetas para todos los campos, porque aunque no hemos especificado explícitamente la mayoría de ellos, debemos tener 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 garantizar que se implementó según lo que planeamos.
Abrimos nuestro archivo **/catalog/tests/test_models.py**, y reemplazamos cualquier código existente con el siguiente código de prueba para el modelo de Autor.
Aquí veremos que primero importamos `TestCase` y derivamos nuestra clase de prueba (`AuthorModelTest`), usando un nombre descriptivo para que podamos identificar fácilmente las pruebas que fallan en el resultado de la prueba. Luego llamamos a `setUpTestData()` para crear un objeto de autor que usaremos pero que no modificaremos en ninguna de las pruebas.

```
1 from django.test import TestCase
2 from catalog.models import Author
3
4 class AuthorModelTest(TestCase):
```

```
5
6       @classmethod
7       def setUpTestData(cls):
8           """
9           Configuramos objetos no modificados utilizados
10          por todos los metodos de prueba.
11          """
12          Author.objects.create(first_name='Big',
13                                last_name='Bob')
14
15      def test_first_name_label(self):
16          author=Author.objects.get(id=1)
17          field_label = author._meta.get_field
18                        ('first_name').verbose_name
19          self.assertEquals(field_label,'first name')
20
21      def test_date_of_death_label(self):
22          author=Author.objects.get(id=1)
23          field_label = author._meta.get_field
24                        ('date_of_death').verbose_name
25          self.assertEquals(field_label,'died')
26
27      def test_first_name_max_length(self):
28          author=Author.objects.get(id=1)
29          max_length = author._meta.get_field
30                       ('first_name').max_length
31          self.assertEquals(max_length,100)
32
33      def test_object_name_is_last_name_comma_first_name
34          (self):
35          author=Author.objects.get(id=1)
36          expected_object_name = '%s, %s' %
37                                 (author.last_name,
38                                  author.first_name)
39          self.assertEquals(expected_object_name,
40                            str(author))
41
42      def test_get_absolute_url(self):
43          author=Author.objects.get(id=1)
44          """
45          Esto tambien fallara si el urlconf no esta
46          definido.
47          """
48          self.assertEquals(author.get_absolute_url(),
49                            '/catalog/author/1')
```

Las pruebas de los campos comprueban que los valores de las etiquetas de los campos (`verbose_name`) y que el tamaño de los campos de caracteres son los esperados.
También necesitamos probar nuestros métodos personalizados. Básicamente, estos solo comprueban que el nombre del objeto se construyó como esperábamos con el formato .ªpellido", "Nombre", y que la URL que obtenemos para un elemento de Autor es como esperaríamos. Ahora sintámonos libres de crear nuestras propias pruebas para nuestros otros modelos.

6.2. Formularios

La filosofía para probar nuestros formularios es la misma que para probar sus modelos; debe probar todo lo que hayamos codificado o nuestro diseño especifique, pero no el comportamiento del framework subyacente y otras bibliotecas de terceros. Para efectos de este laboratorio crearemos un formulario para la renovación de entrega de un libro:

```
class RenewBookForm(forms.Form):
    """
    Formulario para que un bibliotecario renueve
    los libros.
    """
    renewal_date = forms.DateField(help_text="Ingrese
                                   una fecha entre hoy
                                   y 4 semanas (por
                                   defecto 3).")

    def clean_renewal_date(self):
        data = self.cleaned_data['renewal_date']

        #La fecha de verificacion es antes de hoy
        if data < datetime.date.today():
            raise ValidationError(_('Fecha invalida'))
        #La fecha de verificacion esta dentro del rango
        if data > datetime.date.today() +
                  datetime.timedelta(weeks=4):
            raise ValidationError(_('Fecha invalida-
                                     renovacion con
                                     mas de 4 semanas
                                     de anticipacion'))

        # Devolvemos siempre data limpia.
        return data
```

Ahora abrimos nuestro archivo **/catalog/tests/test_forms.py** y reemplazamos cualquier código existente con el siguiente código de prueba para el formu-

lario `RenewBookForm`. Comenzamos importando nuestro formulario y algunas bibliotecas de Python y Django para ayudar a probar la funcionalidad relacionada con el tiempo. Luego declaramos nuestra clase de prueba de formulario de la misma manera que lo hicimos para los modelos, usando un nombre descriptivo para nuestra clase de prueba derivada de `TestCase`.

```
from django.test import TestCase
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,
                         'Ingrese una fecha entre hoy y
                         4 semana (por defeacto 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())
```

Las dos primeras funciones prueban que la etiqueta del campo y el texto de ayuda son los esperados. Tenemos que acceder al campo utilizando el diccionario de campos (por ejemplo, `form.fields ['renewal_date']`). Tenga en cuenta que también tenemos que probar si el valor de la etiqueta es `None`, porque aunque Django representará la etiqueta correcta, no devuelve `None` si el valor no está establecido explícitamente.
El resto de las funciones comprueban que el formulario es válido para las fechas de renovación justo dentro del rango aceptable y no válido para valores fuera del rango. Observe cómo construimos valores de fecha de prueba alrededor de nuestra fecha actual (`datetime.date.today()`) usando `datetime.timedelta()` (en este caso, especificando un número de días o semanas). Luego simplemente creamos el formulario, pasamos nuestros datos y probamos si es válido.
Al igual en modelos con estas indicaciones podemos comenzar a probar, ejecutemos las pruebas y confirmemos que el código pase.

6.3. Vistas

Para validar nuestro comportamiento de vista usamos el cliente de prueba de Django. Esta clase actúa como un navegador web ficticio que podemos usar para simular las solicitudes `GET` y `POST` en una URL y observar la respuesta. Podemos ver casi todo sobre la respuesta, desde `HTTP` de bajo nivel (encabezados de resultados y códigos de estado) hasta la plantilla que estamos usando para representar el `HTML` y los datos de contexto que le estamos transmitiendo. También podemos ver la cadena de redirecciones (si las hay) y consultar la URL y el código de estado en cada paso. Esto nos permite verificar que cada vista está haciendo lo que se espera.
Durante las actividades sugeridas se le indico que creará una vista, si no la realizó, tome este código para comenzar con las pruebas. Una vista simple, que proporciona una lista de todos los autores es suficiente. Esta se debe mostrar en la URL **catalog/authors/**.

```
class AuthorListView(generic.ListView):
    model = Author
    paginate_by = 10
```

Como esta es una vista de lista genérica, casi todo está hecho para nosotros por Django. Podríamos decirse que si confiamos en Django, entonces lo único que necesitamos probar es que se pueda acceder a la vista en la URL correcta y se pueda acceder utilizando su nombre. Sin embargo, si estamos utilizando un proceso de desarrollo basado en pruebas, comenzaremos por escribir pruebas que confirmen que la vista muestra a todos los Autores, paginándolos en grupos de 10.

Abrimos el archivo **/catalog/tests/test_views.py** y reemplazamos cualquier texto existente con el siguiente código de prueba para `AuthorListView`. Como antes, importamos nuestro modelo y algunas clases útiles. En el método `setUpTestData()` configuramos una serie de objetos de `Author` para que podamos probar nuestra paginación.

```
from django.test import TestCase
from catalog.models import Author
from django.urls import reverse

class AuthorListViewTest(TestCase):

    @classmethod
    def setUpTestData(cls):
        #Crea 13 autores por pagina
        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):
```

```
33          resp = self.client.get(reverse('authors'))
34          self.assertEqual(resp.status_code, 200)
35          self.assertTrue('is_paginated' in resp.context)
36          self.assertTrue(resp.context['is_paginated'] ==
37                          True)
38          self.assertTrue(len(resp.context
39                          ['author_list']) == 10)
40
41      def test_lists_all_authors(self):
42          """
43          Va a la segunda pagina y confirma que hay
44          exactamente 3
45          """
46          resp = self.client.get(reverse('authors')+
47                                 '?page=2')
48          self.assertEqual(resp.status_code, 200)
49          self.assertTrue('is_paginated' in resp.context)
50          self.assertTrue(resp.context['is_paginated'] ==
51                          True)
52          self.assertTrue(len(resp.context
53                          ['author_list']) == 3)
```

Todas las pruebas usan el cliente (que pertenece a nuestra clase derivada de `TestCase`) para simular una solicitud `GET` y obtener una respuesta (`resp`). La primera versión verifica una URL específica (nota, solo la ruta específica sin el dominio) mientras que la segunda genera la URL a partir de su nombre en la configuración de la URL.

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

Una vez que tenemos la respuesta, le preguntamos por el estado de su código, la plantilla utilizada, si la respuesta está paginada o no, la cantidad de elementos devueltos y la cantidad total de elementos.
La variable más interesante que hemos demostrado hasta el momento es `resp.context`, que es la variable de contexto que la vista pasa a la plantilla. Esto es increíblemente útil para las pruebas, ya que nos permite confirmar que nuestra plantilla está obteniendo todos los datos que necesita. En otras palabras, podemos verificar que estamos usando la plantilla prevista y qué datos está obteniendo la plantilla, lo que ayuda en gran medida a verificar que cualquier problema de representación se deba únicamente a la plantilla.

Actividades Sugeridas

- Codifique por lo menos una prueba de un modelo, vista y formulario, de los ejemplos realizados en laboratorios anteriores.
- Investigue como probar una vista con formulario y realice las pruebas con lo desarrollado hasta el momento.