Entrega 6

Testing

Profesor:

Oscar Eduardo Alvarez Rodrigue

Fecha: 16 / 11 / 2025

Integrantes:

Cristian David Arcia Quintero

Juan Jose Sanchez Mora

Daniel Mauricio Samaca Gonzalez

Jose Alejandro Solano Garces

Bogotá DC.

2025

En esta entrega se presentan los avances relacionados con la implementación y validación de diversas funcionalidades de la aplicación **My Building App**. Para ello, se desarrollaron y ejecutaron múltiples **tests unitarios** que permiten verificar el correcto funcionamiento de los módulos clave del sistema.

A continuación, se explican los tests realizados, la función testeada y los casos identificados en el test:

Tests de funcionalidades de login

A continuación, se muestran algunas funcionalidades necesarias para la implementación de un sistema de login, el sistema de login ya tiene todos sus templates y se está puliendo su lógica para el momento de esta entrega, debido a que algunas funcionalidades de este sistema, solo se pueden testear con test de integración, a continuación se presentan algunas de las funcionalidades que se pueden testear con test unitarios, los test a continuación se hicieron con la librería unitest, no todos lo test de el proyecto se hicieron con esta herramienta.

Antes de comenzar se indica que los test están en el archivo tests.py de la carpeta core del proyecto, y las funciones testeadas están en el archivo services de la carpeta services en el core del proyecto.

1. **es_email_valido:**

    a. **Funcionalidad:** El propósito de esta función es validar que la estructura de una cadena de caracteres, para identificar si es un email válido, sólo se centra en la estructura no válida nombres de dominio válidos, ni extensiones válidas.

    b. **Casos limites:** Para que la función diga que la cadena pasada es un correo válido debe tener un nombre de usuario, un arroba (@), un nombre de dominio (gmail o hotmail), un punto (.) y una extensión (com, co, etc), teniendo en cuenta lo anterior se presentan los siguientes casos límites:

        - El primer caso sería una cadena sin arroba, como “usuario”, como no tiene dominio ni extensión, la función debería devolver una negativa.

        - El segundo caso sería una cadena con usuario y dominio, pero sin extensión, como “usuario@dominio”, debido a su parte faltante, la función debería devolver una negativa.

        - Aparte, también se espera que si el dominio y la extensión están vacíos, es decir, cadenas como las siguientes: “usuario@” o “usuario@dominio.”, la función debería devolver una negativa.

    c. **Caso normal:** El caso normal es cuando se detecta un email válido, es decir que una cadena sea parecida a lo siguiente “usuario@dominio.extencion”, y

que cada uno de estos componentes tenga contenido, al detectar este caso la función debería una aprobación.

2. **es_contraseña_valida:**

    a. **Funcionalidad:** El propósito de esta función es validar que una contraseña cumpla con requisitos mínimos de seguridad, como una longitud mayor o igual a 8, que contenga al menos una mayúscula, una minúscula y un número, y por último que no contenga un espacio.

    b. **Casos limites:** Para que la función diga que la cadena pasada es una contraseña válida se deben cumplir los requisitos anteriormente expuestos, teniendo en cuenta esto, se presenta los siguientes casos límites:

        - El primer caso sería una cadena que no sea de mínimo 8 caracteres, como “short”, en este caso la función debe devolver una negativa.

        - El segundo caso sería una cadena sin mayúsculas, como “minúscula”, en este caso la función debe devolver una negativa.

        - El tercer caso sería una cadena sin minúsculas, como “MAYÚSCULA”, en este caso la función debe devolver una negativa.

        - El cuarto caso sería una cadena sin números, como “Contraseña”, en este caso la función debe devolver una negativa.

        - El último caso sería una cadena con espacio, como “Buena Contraseña 1”, en este caso la función debe devolver una negativa.

    c. **Caso normal:** El caso normal es cuando se detecta una contraseña válida, que cumpla con todos los requisitos, como lo sería “BuenaContraseña1”, en este caso la función debería devolver una aprobación.

3. **crear_codigo_de_autenticacion:**

    a. **Funcionalidad:** El propósito de esta función es generar un código de autenticación aleatorio para el proceso de recuperar contraseña, el código se genera respetando ciertos parámetros, como lo son su longitud y el número de letras, y en su configuración base produce códigos de 6 dígitos con 4 letras.

    b. **Casos bordes:** Para testear esta funciona se uso una funcion llamada validar código, que recibe la longitud y la proporción deseada y analiza si el código generado por la funcion cumple con lo esperado, teniendo en cuanta esto, se presentan los siguientes casos límites:

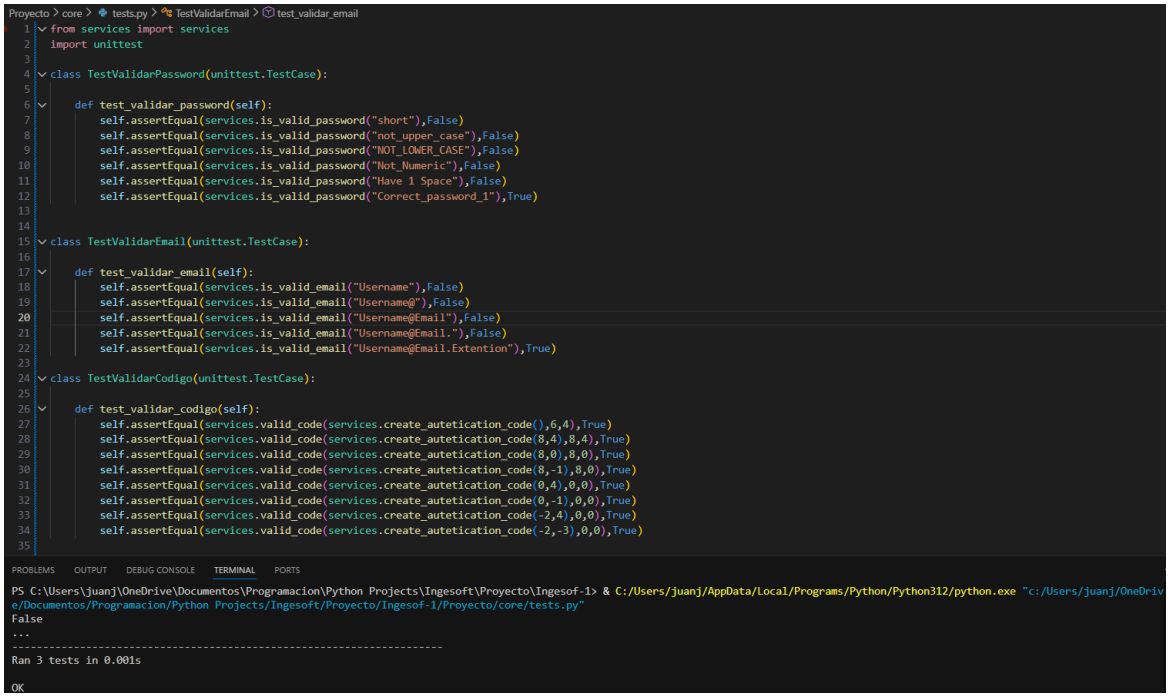
- El primer caso sería cuando se le pasa a la función un número de letras esperado mayor a la longitud pedida, en este caso se espera que la función retorna una cadena de la longitud pedida de solo letras.

- El segundo caso sería cuando se le pasa a la función un número negativo de letras esperado, en este caso la función debería devolver una cadena de la longitud esperada de solo números.

- El tercer caso sería cuando le pasas una longitud negativa a la función, en este caso independientemente de la cantidad de letras esperada, se espera que la función devuelva una cadena vacía.

c. **Caso normal:** El caso normal es cuando se le pasa a la función una longitud deseada y una cantidad de letras esperada, ambas positivas y la primera mayor a la segunda, en este caso normal se debería recibir una cadena que cumpla los parámetros especificados.

Por último, como prueba de la realización de los tese, a continuación se mostrará una imagen de la ejecución de los tests:



Tests de la app de reservas

En el progreso actual del proyecto se tienen alrededor de 3 templates desarrollados para la app de reservas, estas vistas ya cuentan con la respectiva lógica de negocio y los diferentes diseños acorde a los modelados en los mockups. Evitando hacer test de integración para centrarnos únicamente en los test unitarios, fue necesario dejar de lado tanto las funciones que retornar acciones de redirect o render, como las que gestionan y se comunican directamente con la base de datos.

Cada una de las pruebas se implementaron con la herramienta de **TestCase** la cual necesita ser importada desde **django.test**. Las funciones que cumplen las condiciones anteriores para ser testeables y que gestionan parte de la lógica de negocio enfocada al sistema de reservas son las siguientes:

1. **horarios_posibles_hoy:**

    a. **Funcionalidad:** El propósito de esta función consiste en que calculando la fecha y hora del instante actual se generen los “horarios” en los que se puede ejecutar una reserva para el dia de hoy, por ejemplo, si el usuario está usando el sistema a la una de la tarde, este último no debe mostrar cómo horarios para una posible reserva, los bloques de las 6, 8, 10 y 12 de la mañana del presente dia.

    b. **Casos límites**: Para esta función dependen únicamente de la hora actual

        - Son las 5 am: Para este punto se deberían crear todos los bloques horarios que corresponden al presente dia (bloques de dos horas desde las 6 am - 6 pm), por lo que al final se obtendrá una serie de tuplas que serán el resultado de la iteración de todos los bloques horarios posibles (7) con la cantidad de zonas comunes posibles.

        - Son las 7 pm: La función no debería retornar alguna tupla ya que no hay bloques horarios permitidos para el presente día.

    c. Caso normal

        - Son las 12 m: Las tuplas creadas solo deben contener a los bloques horarios desde las 2 pm - 6 pm

2. **construir_reservas_disponibles:**

    a. **Funcionalidad:** Esta función compara todos los posibles horarios para ejecutar una reservación con los horarios que ya se encuentran reservados, esto con el fin de retornar una lista de diccionarios con los horarios verdaderamente disponibles desde el momento actual hasta los siguientes 6 días

    b. **Casos limites**:

        - Todos los horarios ocupados: Al contrastar todos los horarios posibles para ejecutar una reservación con los horarios que ya están reservados y ver observar que ambos conjuntos contienen lo mismo, la funcion no deberia retornar una lista vacia

        - Todos los horarios disponibles: Al igual que en la función anterior primero se contrastan los horarios posibles con los horarios ya reservados, pero en este caso no coincide ninguno (no existen

horarios reservados por lo cual todos los horarios posibles también son horarios disponibles). La función retorna una lista de diccionarios con todos los horarios

c. **Caso normal**

- Algunos horarios ocupados: Al hacer la misma comparación que se mencionó anteriormente entre los horarios posibles y los horarios reservados, la función retorna los horarios que se son disjuntos entre ambos conjuntos (los que están en posibles pero no están en reservados)

3. **obtener_campos**

a. **Funcionalidad:** A través de una petición http con el método POST, el usuario registra ciertos campos fundamentales para formalizar una reserva en la BD, por lo que es necesario validar el formato de cada uno de estos campos ingresados.

b. **Casos limites**:

- Fecha inválida: Al ingresar una fecha inválida sería imposible gestionar correctamente una reserva, es por esto que la función retorna NULL además de refrescar la página y notificar al usuario a través de un mensaje.

- Hora inválida: El mecanismo es el mismo que para el caso anterior

- Zona común invalida: Teniendo una zona común invalida no es posible asociar una reserva con un id_zona_comun, lo que degrada la calidad del sistema de reservas y permitiría la creación de reservas al aire que no se verían reflejadas en la vista de “horarios disponibles”, es por esto que la función también retorna NULL y lanza un mensaje de error

c. **Caso normal**

- Formato válido: cuando se obtiene un parseo correcto de cada uno de los campos, se guarda la reserva en la base de datos y se actualiza la página

Tests de funcionalidad app foro

Los siguientes tests se realizaron en Testcase con base en dos templates del módulo de foro; el template principal forum_home, que es la pantalla principal para cualquier usuario, y que contiene una separacion por categorías de todas las publicaciones. El segundo template admin_approval es la pantalla de administrador, en la cual se aprueban o rechazan las solicitudes de publicación pendientes por aprobación.

1. **setUp(self)**

- Propósito: Su función principal es preparar el entorno de la base de datos con datos de prueba consistentes para cada prueba.
- Operaciones:
    - Crea un Usuario de prueba (self.user) que servirá como autor de las publicaciones.
    - Crea varias Publicacion con diferentes estados de visibilidad y categoria para ser utilizadas en las pruebas subsiguientes:
        - post1, post2: Visibles (visibilidad=1).
        - post3, post4: Pendientes de aprobación (visibilidad=None).

2. **test_main_view_displays_visible_posts(self)**

- Propósito: Verificar que la vista principal del foro (reverse('forum')) solo muestra las publicaciones que han sido aprobadas (es decir, visibilidad=1). Las publicaciones pendientes o rechazadas no deben ser visibles para el usuario común.
- Casos Normales:
    - Asegura que Test Post 1 y Test Post 2 (visibles) se encuentran en la respuesta HTML.
- Casos Límite:
    - Asegura que Pending Post y Post to be Rejected (pendientes) *no* se encuentran en la respuesta HTML. Esto confirma la lógica de filtrado de visibilidad.
    - Verifica que el código de estado de la respuesta HTTP es 200 (OK).

3. **test_filter_by_category_view(self)**

- Propósito: Validar que la vista de filtrado por categoría (reverse('filter_by_category', args=['anuncios'])) funciona correctamente, mostrando solo las publicaciones que pertenecen a la categoría especificada.
- Casos Normales:
    - Filtra por la categoría 'anuncios' y verifica que Test Post 1 (de 'anuncios') está presente.
- Casos Límite:
    - Asegura que Test Post 2 (de 'eventos') *no* está presente, confirmando que el filtro excluye otras categorías.
    - Asegura que Pending Post (pendiente) *no* está presente, lo que implica que el filtro de categoría también debería respetar la visibilidad
    - Verifica que el código de estado de la respuesta HTTP es 200 (OK).

4. **test_create_post_view(self)**

- Propósito: Probar la funcionalidad de creación de nuevas publicaciones a través de una solicitud POST a la vista create_post.
- Casos Normales:
    - Envía datos válidos para una nueva publicación.
    - Verifica que la respuesta es una redirección (código 302), lo que indica una operación exitosa que generalmente redirige al usuario a otra página.
    - Confirma que una publicación con el título 'New Test Post' existe en la base de datos después de la operación.
    - Verifica que la cantidad de publicaciones en la base de datos ha aumentado en uno.
    - Verifica que la visibilidad de la nueva publicación es None por defecto, indicando que está pendiente de aprobación.
- Casos Límite/Consideraciones:
    - Validación de Formulario: Casos de datos inválidos (por ejemplo, título vacío, categoría inexistente).

5. **test_admin_approval_view(self)**

- Propósito: Asegurar que la vista diseñada para la aprobación de administradores (reverse('admin_approval')) solo muestra las publicaciones que están pendientes de revisión (visibilidad=None).
- Casos Normales:
    - Verifica que Pending Post y Post to be Rejected (pendientes) se encuentran en la respuesta HTML.
- Casos Límite:
    - Asegura que Test Post 1 (ya visible) *no* se encuentra en la respuesta HTML, confirmando que la vista excluye publicaciones ya gestionadas.
    - Verifica que el código de estado de la respuesta HTTP es 200 (OK).
- Consideraciones:
    - Permisos de Administrador: Este test no simula la autenticación como administrador.

6. **test_accept_post_view(self)**

- Propósito: Verificar que la vista accept_post (reverse('accept_post', args=[self.post3.id_publicacion])) cambia correctamente el estado de visibilidad de una publicación pendiente a 1 (visible).
- Casos Normales:
    - Toma post3 (que inicialmente tiene visibilidad=None).
    - Realiza una solicitud GET a la vista de aceptación.
    - Verifica una redirección (302).
    - Crucial: Usa self.post3.refresh_from_db() para recargar el objeto post3 con su estado más reciente de la base de datos. Sin esto, el objeto post3 en memoria seguiría teniendo visibilidad=None.
    - Afirma que la visibilidad del post3 ahora es 1..

7. **test_reject_post_view(self)**

- Propósito: Verificar que la vista reject_post (reverse('reject_post', args=[self.post4.id_publicacion])) cambia correctamente el estado de visibilidad de una publicación pendiente a 2 (rechazada).
- Casos Normales:
    - Toma post4 (que inicialmente tiene visibilidad=None).
    - Realiza una solicitud GET a la vista de rechazo.
    - Verifica una redirección (302).
    - Crucial: Usa self.post4.refresh_from_db().
    - Afirma que la visibilidad del post4 ahora es 2.

Test de otras funcionalidades de la aplicación
A continuación se testeando funciones que se usan en diferentes secciones de la aplicación y puede que en más de una, se sigue con la metodología anterior de explicar la funcionalidad de la función y sus casos testeados:

1. `solicitar_contraseña`:

a. Funcionalidad:
El propósito de esta función es validar que un usuario ingrese dos contraseñas idénticas. Si ambas cadenas coinciden se acepta la contraseña, mientras que si son diferentes, la función solicita el reingreso hasta que la coincidencia sea correcta.

b. Casos límite:

- El primer caso límite es cuando se ingresan dos contraseñas distintas, por ejemplo: `"abc"` y `"xyz"`. En este caso la función debe solicitar nuevamente los valores.

- El segundo caso límite es cuando se requiere más de un intento exitoso, por ejemplo: primero dos contraseñas distintas y luego dos iguales. El test debe confirmar que la función retorna la correcta en el segundo intento.

c. Caso normal:
El caso normal es cuando ambas contraseñas coinciden en el primer intento, por ejemplo: `"password123"` y `"password123"`. En este caso la función debe retornar la contraseña sin requerir más solicitudes.

2. `validar_asunto`:

a. Funcionalidad:
Determina si un asunto ingresado por el usuario es válido, manteniendo un formato apropiado. Un asunto válido debe contener únicamente caracteres permitidos y tener una longitud entre 5 y 60 caracteres.

b. Casos límite:

- Asunto demasiado corto, por ejemplo `"ok"` → la función debe devolver una negativa.
- Asunto demasiado largo (más de 60 caracteres) → la función debe devolver una negativa.
- Asunto con caracteres inválidos, por ejemplo `"Error $$ sistema"` → la función debe devolver una negativa.

c. Caso normal:
El caso normal es cuando se ingresa un asunto de longitud adecuada y con caracteres permitidos, por ejemplo `"Solicitud de mantenimiento"`. En este caso la función debe devolver una aprobación.

3. `validar_descripcion`:

a. Funcionalidad:
Evalúa que la descripción escrita por el usuario cumpla un tamaño mínimo y máximo adecuado para asegurar contenido significativo sin desbordar la información.
Una descripción válida debe tener entre 10 y 500 caracteres.

b. Casos límite:

- Descripción muy corta, por ejemplo `"muy corto"` → la función debe devolver una negativa.
- Descripción excesivamente larga (más de 500 caracteres) → la función debe devolver una negativa.

- Descripción con caracteres inválidos o inesperados según pruebas → la función debe devolver una negativa.

c. Caso normal:

El caso normal se presenta cuando la descripción contiene información completa y suficiente, por ejemplo:

`"Necesito reparación de la puerta del salón comunal."`

La función debería retornar una aprobación.