UNIVERSIDAD NACIONAL DE COLOMBIA
FACULTAD DE INGENIERÍA
DEPARTAMENTO DE INGENIERÍA DE SISTEMAS E INDUSTRIAL

Entrega 06

Fecha: 16/11/2025

Joan Camilo Betancourt Gonzalez (jobetancourtg@unal.edu.co)
Nicolás Alejandro Diosa Benavides (ndiosab@unal.edu.co)
Jonathan Felipe López Nuñez (jolopezn@unal.edu.co)
Raúl Felipe Rodríguez Hernández (rrodriguezhe@unal.edu.co)

Pruebas unitarias Joan Camilo Betancourt González:

Los 3 tests unitarios que realicé son de la clase Auth, la cual se utiliza en negocio para la parte de registro y login del usuario, en este caso las funciones a testear son:

- Incorrect password
- Is valid password policy
- Password match

Se hace uso de la librería unittest de python.

Test: is_valid_password_policy

En este test se evalúa si la contraseña cumple con la política de Parchate, 8 caracteres, debe tener un símbolo especial y un número. Si no se cumplen las tres políticas, la función debe retornar un valor false, como se muestra a continuación en el test:

Python

```
'''
Hace el test de la función is_valid_password de la clase
UserValidator en el archivo auth.py



En este test existen los siguientes casos:
```

```
Caso 1: La contraseña tiene una longitud mayor o igual a 8
simbolos
        y tiene al menos un número y tiene al menos un simbolo
especial (true)

Caso 2: La contraseña no cumple con las 3 condiciones (false):
        - No tiene una longitud mayor o igual a 8 simbolos
        - No tiene al menos un número
        - No tiene al menos un simbolo especial

    En este caso existe el caso limite: la contraseña tiene
exactamente 8 simbolos,
    la contraseña tiene exactamente un número, la contraseña
tiene exactamente un simbolo especial.
'''


import unittest
from core.Negocio.auth import UserValidator

class Test_is_valid_password_policy(unittest.TestCase):

    def setUp(self):
        self.validator = UserValidator()

    def test_is_valid_password_policy(self):
        casos = [
            ("", False),                          # Cadena vacía
(0 simbolos)
            ("corto1*", False),                   # Contraseña con
longitud menor a 8 (7 simbolos)
            ("*contrasenasinnumero", False),      # No tiene
digitos
```

```
            ("contrasenasinsimbolo12", False),    # No tiene
símbolos
            ("contraseñasinsimbolo12", True),     # Es la misma
contraseña de arriba pero ñ cuenta como simbolo especial
            ("Valida8!", True),                   # Contraseña con
longitud 8, con número y símbolo (Limite)
            ("12345678*", True),                  # Contraseña con
solo digitos y simbolo
            ("@@@2@@@@", True),                   # Contraseña con
solo simbolos y digito
        ]

        for entrada, esperado in casos:
            with self.subTest(entrada=entrada):

self.assertEqual(self.validator.is_valid_password_policy(entrada)
, esperado)


if __name__ == '__main__':
    unittest.main()
```

Resultado:

```
                                                                                python manage.py test core.tests.test_is_valid_password_poli
cyILED (errors=1)
>> nv) PS C:\Users\milob\Documents\SQL Server Management Studio\Los-sin-P.A.P.A\Proyecto\Internal>
.env cargado desde: C:\Users\milob\Documents\SQL Server Management Studio\Los-sin-P.A.P.A\Proyecto\Internal\.env
Found 1 test(s).
System check identified some issues:

WARNINGS:
core.Chat.id_actividad: (fields.W342) Setting unique=True on a ForeignKey has the same effect as using a OneToOneField.
        HINT: ForeignKey(unique=True) is usually better served by a OneToOneField.
core.ParticipanteActividad.id_actividad: (fields.W342) Setting unique=True on a ForeignKey has the same effect as using a OneToOneField.
        HINT: ForeignKey(unique=True) is usually better served by a OneToOneField.

System check identified 2 issues (0 silenced).
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

Test: password_match

Este test permite verificar que las contraseñas al registrarse sean las mismas, solo existen dos casos, o ambas son iguales o son diferentes como se muestra en el test:

Python

```python
'''
Hace el test de la función password_match de la clase
UserValidator en el archivo auth.py


En este test existen dos casos posibles:


Caso 1: Las contraseñas coinciden (true)
Caso 2: Las contraseñas no coinciden (false)


'''


import unittest
from core.Negocio.auth import UserValidator


class Test_password_match(unittest.TestCase):
    def setUp(self):
        self.validator = UserValidator()

    def test_passwords_match(self):
        casos = [
            ("abc123", "abc123", True),                    # Son
iguales
            ("", "", True),                                #
Ambas vacías (Iguales)
            ("Nosequeponer", "yasupequeponer", False),     #
Diferentes
```

```
            ("abc123", "ABC123", False),                    #
Mayúsculas las hacen diferentes
            ("password", None, False),                      # Una
es None (Diferentes)
        ]

        for pass1, pass2, esperado in casos:
            with self.subTest(p1=pass1, p2=pass2):

self.assertEqual(self.validator.passwords_match(pass1, pass2),
esperado)



if __name__ == '__main__':
    unittest.main()
```

Resultado:

```
(venv) PS C:\Users\milob\Documents\SQL Server Management Studio\Los-sin-P.A.P.A\Proyecto\Internal> python manage.py test core.tests.test_password_match
>>
core.Chat.id_actividad: (fields.W342) Setting unique=True on a ForeignKey has the same effect as using a OneToOneField.
        HINT: ForeignKey(unique=True) is usually better served by a OneToOneField.
core.ParticipanteActividad.id_actividad: (fields.W342) Setting unique=True on a ForeignKey has the same effect as using a OneToOneField.
        HINT: ForeignKey(unique=True) is usually better served by a OneToOneField.

System check identified 2 issues (0 silenced).
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

Test: incorrect_password

Para este test, solo hay dos casos de login, o la contraseña ingresada corresponde a la del usuario, o la contraseña ingresada no corresponde, primero se crea un usuario y con los casos se comprueba si la contraseña dada es igual a la de la base de datos, como se muestra en el test:

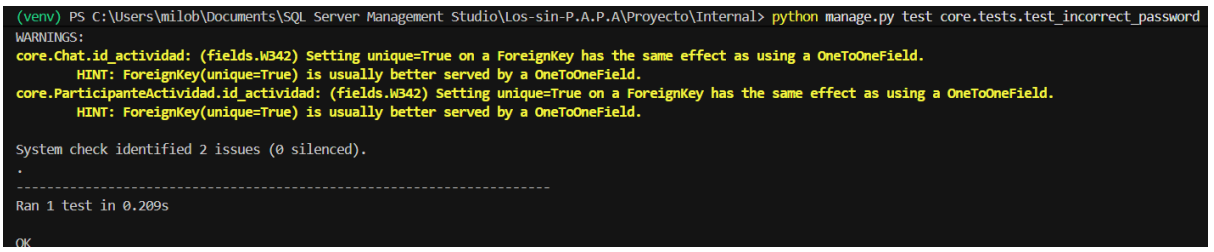
Python

```
'''
Hace el test de la función incorrect_password de la clase
UserValidator en el archivo auth.py
```

```
En este test únicamente existen dos casos posibles:

Caso 1: La contraseña coincide y está asignada para el usuario
(false)
Caso 2: La contraseña no coincide para el usuario (true)

'''

from core.Negocio.auth import UserValidator
from core.models import Usuario
import unittest

class Test_incorrect_password(unittest.TestCase):
    def test_incorrect_password(self):
        # Los datos
        casos = [
            ("contraseña_asociada_a_usuario", False),
            ("contraseña_no_asociada_a_usuario", True)
        ]
        userV = UserValidator()

        for entrada, esperado in casos:
            # Contexto

userV.db.create_usuario("userTestincorrect_password","emailTestin
correct_password","contraseña_asociada_a_usuario",None,None,None)

            with self.subTest(entrada=entrada):
                # Lo que se espera

self.assertEqual(userV.incorrect_password("userTestincorrect_pass
word",entrada), esperado)
```

```
userV.db.delete(Usuario,"nombre_usuario","userTestincorrect_passw
ord")


if __name__ == '__main__':
    unittest.main()
```

Resultado:



Pruebas unitarias Nicolás Alejandro Diosa Benavides:

Estos tres test unitarios evalúan las funciones alojadas en auth.py más exactamente en la clase UserValidator, además se hacen con la librería unittest expuesta por el profesor en clase, en los códigos se encuentran los casos posibles claramente expuestos, desde los casos válidos hasta los inválidos:

1. username_available
2. email_available
3. is_valid_email

Así pues detalladamente está:

1. username_available: En este test sólo se evalúa o examina si existe ya el nombre de usuario por lo que solo existen dos casos posibles, que exista o que no, para hacer la prueba insertamos un usuario de prueba para poder efectuar el test y se borrará al final de la ejecución; este mecanismo es utilizado también para el test de email_available:

Python

```
'''
Hace el test de la función username_available de la clase
   UserValidator en el archivo auth.py
```

```
En este test únicamente existen dos casos posibles:

Caso 1: El nombre de usuario está disponible para ser
   registrado (True)
Caso 2: El nombre de usuario no está disponible para ser
   registrado (False)

'''

from core.Negocio.auth import UserValidator
from core.models import Usuario
import unittest

class Test_username_available(unittest.TestCase):
    def test_username_available(self):
        # Los datos
        casos = [
            ("nombre disponible", True),
            ("nombre ya ocupado", False)
        ]
        userV = UserValidator()

        for entrada, esperado in casos:
            # Contexto

   userV.db.create_usuario("usuarioTestUserAvaibable","nombre
   ya ocupado","contraseñaTestUserAvaibable",None,None,None)

            with self.subTest(entrada=entrada):
                # Lo que se espera

   self.assertEqual(userV.email_available(entrada), esperado)
```

```
    userV.db.delete(Usuario,"nombre_usuario","usuarioTestUserAva
    ibable")


if __name__ == '__main__':
    unittest.main()
```

La función evaluada es la siguiente:

```
    def username_available(self, username):
        try:
            self.db.get_usuario_by_nombre_usuario(username)
            return False
        except ObjectDoesNotExist:
            return True
```

El resultado del test arroja lo siguiente:

```
(venv) PS C:\Users\Usuario\Documents\UNAL\Matrícula 2025-II\Ingeniería de software I\Los-sin-P.A.P.A\Proyecto\Internal> python manage.py test core.tests.test_username_available
.env cargado desde: C:\Users\Usuario\Documents\UNAL\Matrícula 2025-II\Ingeniería de software I\Los-sin-P.A.P.A\Proyecto\Internal\.env
Found 1 test(s).
System check identified some issues:

WARNINGS:
core.Chat.id_actividad: (fields.W342) Setting unique=True on a ForeignKey has the same effect as using a OneToOneField.
        HINT: ForeignKey(unique=True) is usually better served by a OneToOneField.
core.ParticipanteActividad.id_actividad: (fields.W342) Setting unique=True on a ForeignKey has the same effect as using a OneToOneField.
        HINT: ForeignKey(unique=True) is usually better served by a OneToOneField.

System check identified 2 issues (0 silenced).
.
----------------------------------------------------------------------
Ran 1 test in 0.054s

OK
```

La conclusión es que la función se comporta de la manera esperada.

2. email_available: En este test como en el anterior se hace una evaluación de existencia del email, si existe entonces la función deberá arrojar el booleano true, y si no entonces false, el test realizado es muy parecido al anterior, usando incluso el mismo mecanismo de creación de un usuario para validar el correcto funcionamiento:

Python

```
'''
Hace el test de la función email_available de la clase
UserValidator en el archivo auth.py

En este test únicamente existen dos casos posibles:
```

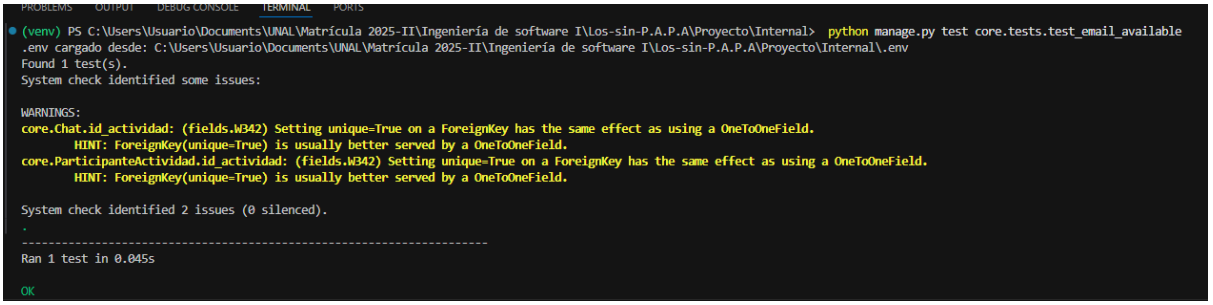
```
Caso 1: El email está disponible para ser registrado (True)
Caso 2: El email no está disponible para ser registrado (False)

'''

from core.Negocio.auth import UserValidator
from core.models import Usuario
import unittest

class Test_email_available(unittest.TestCase):
    def test_email_available(self):
        # Los datos
        casos = [
            ("email disponible", True),
            ("email ya ocupado", False)
        ]
        userV = UserValidator()

        for entrada, esperado in casos:
            # Contexto

userV.db.create_usuario("usuarioTestEmailAvaibable","email ya
ocupado","contraseñaTestEmailAvaibable",None,None,None)

            with self.subTest(entrada=entrada):
                # Lo que se espera
                self.assertEqual(userV.email_available(entrada),
esperado)

userV.db.delete(Usuario,"nombre_usuario","usuarioTestEmailAvaibab
le")


if __name__ == '__main__':
```

```
    unittest.main()
```

La función evaluada es:

```
    def email_available(self, email):
        try:
            self.db.get_usuario_by_email(email)
            return False
        except ObjectDoesNotExist:
            return True
```

El resultado de la ejecución de este test fue el siguiente:



Por lo que se intuye que el funcionamiento del código de email_available es correcto.

3. is_valid_email: Este test es un poco más complejo que los anteriores puesto que cumple con una funcionalidad un poco más significativa, la función que se evalúa en este test se encarga de corroborar la validez de un email ingresado a la hora del registro por lo que es necesaria una expresión regex para hacer las validaciones de formato de la dirección, se toma como dirección válida aquella que tiene un nombre, un arroba, un dominio, un punto que separa a la segunda parte del dominio (.com, .co etc) y por último esa última parte del dominio, un ejemplo válido sería:

   emailvalido@dominio.segundaparteldominio

   Sin esta estructura no puede ser considerado un email válido, adicionalmente, el email no puede tener puntos seguidos en la parte del dominio puesto que no existe ningún dominio con este estilo, así pues el test queda de la siguiente manera:

Python

```
'''
Hace el test de la función is_valid_email de la clase
UserValidator en el archivo auth.py

En este test existen varios casos posibles:
```

```
Caso válido:

1. Que el email cumpla con el nombre, el arroba, el dominio, y el
sufijo con su punto respectivo

Casos inválidos:
1. Que la primera parte del nombre esté vacía
2. Que no exista el arroba
3. Que no exista el dominio
4. Que no exista el punto que separa el dominio del .com o .co
5. Que no exista la última parte referida al .co y ese tipo de
sufijos
6. Que tenga puntos seguidos en la parte del dominio (@hola..com)

'''

from core.Negocio.auth import UserValidator
import unittest

class Test_is_valid_email(unittest.TestCase):
    def test_is_valid_email(self):
        # Los datos
        casos = [
            ("@notienenombre.co", False),
            ("emailsinarroba.co", False),
            ("emailsindominio@.co", False),
            ("emailsinpuntoseparador@dominioco", False),
            ("emailsinpartefinaldominio@dominio.", False),
            ("dominioconpuntosseguidos@dominio..com", False),
            ("emailcorrecto@email.co", True),
            ("emailcorrecto@unal.edu.co", True)
        ]
        userV = UserValidator()

        for entrada, esperado in casos:
```

```
            # Contexto
            with self.subTest(entrada=entrada):
                # Lo que se espera
                self.assertEqual(userV.is_valid_email(entrada),
esperado)

if __name__ == '__main__':
    unittest.main()
```

La función que es evaluada es la siguiente:

```
def is_valid_email(self, email):
    patron_email = r'^[a-zA-Z0-9._%+-]+@([a-zA-Z0-9-]+\.)+[a-zA-Z]{2,}$'
    if re.fullmatch(patron_email, email):
        return True
    else:
        return False
```

La expresión regex (que se ve bastante horrible) da los parámetros de cada una de las porciones resaltadas anteriormente y la función fullmatch() permite evaluar la cadena que llegue especificando si cumple o no con esos parámetros. Los resultados obtenidos en el test son los siguientes:

```
(venv) PS C:\Users\Usuario\Documents\UNAL\Matrícula 2025-II\Ingeniería de software I\Los-sin-P.A.P.A\Proyecto\Internal>  python manage.py test core.tests.test_is_valid_email
.env cargado desde: C:\Users\Usuario\Documents\UNAL\Matrícula 2025-II\Ingeniería de software I\Los-sin-P.A.P.A\Proyecto\Internal\.env
Found 1 test(s).
System check identified some issues:

WARNINGS:
core.Chat.id_actividad: (fields.W342) Setting unique=True on a ForeignKey has the same effect as using a OneToOneField.
        HINT: ForeignKey(unique=True) is usually better served by a OneToOneField.
core.ParticipanteActividad.id_actividad: (fields.W342) Setting unique=True on a ForeignKey has the same effect as using a OneToOneField.
        HINT: ForeignKey(unique=True) is usually better served by a OneToOneField.

System check identified 2 issues (0 silenced).
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

De este modo es posible decir que también ésta función cumple con su cometido.

Pruebas unitarias Jonathan Felipe Núñez:

Los test que que se realizaron están relacionados con la funcionalidad de mostrar actividades y agregar tareas a un evento/materia, siendo estos test, estos test están implementados con pytest:

1. actividad_disponible
2. descripcion_valida
3. estado_valido
4. fecha_valida
5. icalendar_valido

6. prioridad_valida

El test 1 está relacionado con una funcionalidad ya desarrollada, pero los demás test están relacionados con funcionalidades pendientes por desarrollar.

1. actividad_disponible: Verifica si una actividad está disponible, es decir si aún no ha sucedido

Python

```
"""
TEST: Disponibilidad de actividad (actividad_disponible)

Objetivo:
---------
Validar la función `actividad_disponible` en
`core.Negocio.actividades`.

Reglas evaluadas:
-----------------
1. `None` como entrada → no disponible (False)
2. Objeto sin `fecha_hora_inicio` → no disponible (False)
3. `fecha_hora_inicio` en el pasado → no disponible (False)
4. `fecha_hora_inicio` en el presente → no disponible (False)
5. `fecha_hora_inicio` en el futuro → disponible (True)

Casos límite probados:
----------------------
- entrada `None`
- objeto con `fecha_hora_inicio = None`
- fecha pasada, actual y futura

Ejecutar con: `pytest
.\Proyecto\Internal\core\tests\test_actividades.py`
"""

import pytest
from django.utils import timezone
from datetime import timedelta
```

```python
from unittest.mock import MagicMock

from core.Negocio.actividades import actividad_disponible


# Dummy mínimo para simular una entidad Actividad con el atributo
esperado
class DummyActividad:
    def __init__(self, fecha_hora_inicio=MagicMock()):
        self.fecha_hora_inicio = fecha_hora_inicio


@pytest.mark.django_db
def test_actividad_disponible():
    ahora = timezone.now()

    casos = [
        ("actividad_none", None, True),
        ("sin_fecha", DummyActividad(None), False),
        ("fecha_pasada", DummyActividad(ahora -
timedelta(hours=1)), False),
        ("fecha_actual", DummyActividad(ahora), False),
        ("fecha_futura", DummyActividad(ahora +
timedelta(hours=1)), True),
    ]

    for nombre_caso, entrada, esperado in casos:
        assert actividad_disponible(entrada) == esperado, f"Falló
caso: {nombre_caso}"
```

2. descripcion_valida: Verifica que la descripción de una tarea esté entre el rango de 0 y 280 caracteres permitidos

Python

```python
"""
TEST: descripcion_valida

Objetivo:
---------
Probar la validación de la descripción de una tarea: debe aceptar
textos de longitud entre 0 y 280 caracteres inclusive.

Casos límite:
-------------
- cadena vacía (`""`) → válido
- 1 carácter (`"a"`) → válido
- límite superior (`"a" * 280`) → válido
- justo por encima del límite (`"a" * 281`) → inválido
- tipo no-string (`123`) → inválido
- `None` → inválido
"""

import pytest

from core.Negocio.tareas import descripcion_valida


@pytest.mark.parametrize(
    "texto, esperado",
    [
        ("", True),
        ("a", True),
        ("a" * 280, True),
        ("a" * 281, False),
        (123, False),
        (None, False),
    ],
)
def test_descripcion_valida(texto, esperado):
```

```
    assert descripcion_valida(texto) == esperado
```

3. estado_valido: Verifica que una tarea tenga un estado válido, definido entre “Por realizar”, “Realizando” y “Realizada”.

Python

```
"""
TEST: estado_valido

Objetivo:
---------
Validar que el estado de una tarea sea uno de los valores
permitidos:
`Por realizar`, `Realizando`, `Realizada`.

Casos límite:
-------------
- valores permitidos en caso correcto → válido
- mismo texto en distinto case (`por realizar`) → inválido
- valores en otro idioma (`Completed`) → inválido
- `None` → inválido
"""

import pytest

from core.Negocio.tareas import estado_valido


@pytest.mark.parametrize(
    "estado, esperado",
    [
        ("Por realizar", True),
```

```
        ("Realizando", True),
        ("Realizada", True),
        ("por realizar", False),
        ("Completed", False),
        (None, False),
    ],
)
def test_estado_valido(estado, esperado):
    assert estado_valido(estado) == esperado
```

4. fecha_valida: Valida que una tarea tenga una fecha válida, es decir, en el futuro.

Python

```
"""
TEST: fecha_valida

Objetivo:
---------
Probar la validación de la fecha de una tarea: la fecha debe
estar en el futuro.

Casos límite:
-------------
- fecha en el pasado (`timezone.now() - timedelta(minutes=1)`) →
inválido
- fecha exactamente ahora (`timezone.now()`) → inválido
- fecha inmediatamente en el futuro (`timezone.now() +
timedelta(minutes=1)`) → válido
- fecha en formato ISO en el futuro → válido
- texto no parseable (`"not-a-date"`) → inválido
- `None` → inválido
"""
```

```python
import pytest
from django.utils import timezone
from datetime import timedelta

from core.Negocio.tareas import fecha_valida


@pytest.mark.parametrize(
    "valor, esperado",
    [
        (timezone.now() - timedelta(minutes=1), False),
        (timezone.now(), False),
        (timezone.now() + timedelta(minutes=1), True),
        ((timezone.now() + timedelta(days=1)).isoformat(), True),
        ("not-a-date", False),
        (None, False),
    ],
)
def test_fecha_valida(valor, esperado):
    assert fecha_valida(valor) == esperado
```

5. icalendar_valido: Valida que la frecuencia de una tarea tenga un formato de iCalendar valido

Python

```
"""
TEST: icalendar_valido

Objetivo:
---------
```

```
Verificar que un texto tenga el formato mínimo esperado de
iCalendar
conteniendo `BEGIN:VCALENDAR` y `END:VCALENDAR`.

Casos límite:
-------------
- calendario completo con evento → válido
- calendario vacío (`BEGIN:VCALENDAR` / `END:VCALENDAR`) → válido
- solo evento (`BEGIN:VEVENT` / `END:VEVENT`) → inválido
- texto aleatorio → inválido
- `None` → inválido
"""

import pytest

from core.Negocio.tareas import icalendar_valido


@pytest.mark.parametrize(
    "texto, esperado",
    [

("BEGIN:VCALENDAR\nBEGIN:VEVENT\nEND:VEVENT\nEND:VCALENDAR",
True),
        ("BEGIN:VCALENDAR\nEND:VCALENDAR", True),
        ("BEGIN:VEVENT\nEND:VEVENT", False),
        ("random text", False),
        (None, False),
    ],
)
def test_icalendar_valido(texto, esperado):
    assert icalendar_valido(texto) == esperado
```

6. prioridad_valida:Verifica que la prioridad de una tarea sea 1, 2 o 3

Python

```python
"""
TEST: prioridad_valida

Objetivo:
---------
Comprobar que la función `prioridad_valida` acepte únicamente los
números `1`, `2` o `3` como prioridad.

Casos límite:
-------------
- `1`, `2`, `3` → válidos
- `0`, `4`, `-1` → inválidos
- valores no enteros (`"1"`, `1.0`) → inválidos
- `None` → inválido
"""

import pytest

from core.Negocio.tareas import prioridad_valida


@pytest.mark.parametrize(
    "valor, esperado",
    [
        (1, True),
        (2, True),
        (3, True),
        (0, False),
        (4, False),
        (-1, False),
        ("1", False),
        (1.0, False),
        (None, False),
    ],
```

```
)
def test_prioridad_valida(valor, esperado):
    assert prioridad_valida(valor) == esperado
```

Pruebas unitarias Raúl Felipe Rodríguez Hernández:

Los siguientes 3 tests unitarios están hechos para validar la lógica correspondiente a la creación de actividades dentro del modelo ActividadService.

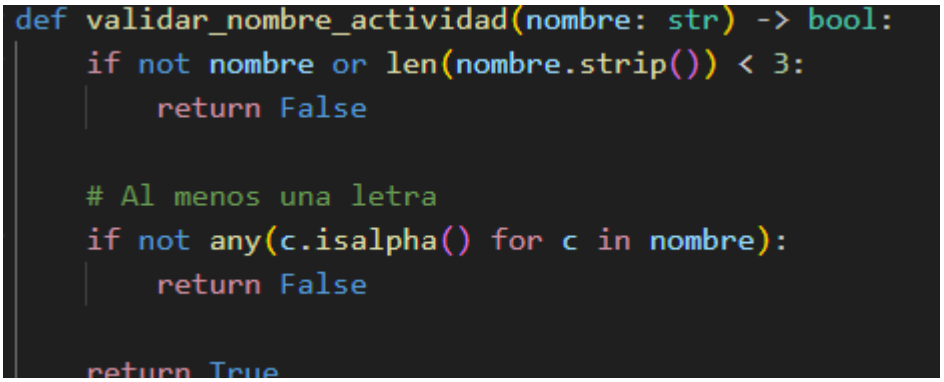
Estos tests fueron desarrollados utilizando pytest, junto con pytest-django para habilitar la ejecución dentro del proyecto Django.

Las pruebas evalúan la funcionalidad principal de la capa de negocio, asegurando que solo se creen actividades con datos válidos y que los errores se detecten correctamente. Los tests implementados son: validar nombre de la actividad, validar número de cupos y validar fechas de inicio y fin.

En todos los casos se utilizó unittest.mock.patch para reemplazar DB_Manager, evitando el acceso real a la base de datos y enfocando las pruebas únicamente en la lógica de negocio.

El primer test es el de la validación del nombre de la actividad

Este test evalúa la función validar nombre actividad y su integración dentro de crear actividad.



Función validar_nombre_actividad

Python

```python
class ActividadService:
    """Capa de negocio para manejar la lógica de actividades."""
    def crear_actividad(self, request, datos, foto=None):
        """Crea una nueva actividad validando los datos antes de guardarla."""
        db = DB_Manager()
        # Buscar el usuario (creador)
        try:
            creador = db.get_usuario_by_nombre_usuario(request.session['username'])
        except db.get_usuario_by_nombre_usuario(request.session['username']).DoesNotExist:
            raise ValueError("Usuario no encontrado en la base de datos.")

        # Validaciones
        nombre = datos.get('nombre_actividad', '')

        if not validar_nombre_actividad(nombre):
            raise ValueError("El nombre de la actividad no es válido.")

        fecha_inicio_raw = datos.get('fecha_hora_inicio')
        fecha_fin_raw = datos.get('fecha_hora_fin')

        if not fecha_inicio_raw:
```

```
            raise ValueError("La fecha de inicio es
obligatoria.")

        if not fecha_fin_raw:
            raise ValueError("La fecha de fin es
obligatoria.")

        try:
            fecha_inicio =
datetime.datetime.fromisoformat(fecha_inicio_raw)
            fecha_fin =
datetime.datetime.fromisoformat(fecha_fin_raw)
        except Exception:
            raise ValueError("Las fechas deben estar en
formato ISO YYYY-MM-DDTHH:MM.")

        if fecha_fin <= fecha_inicio:
            raise ValueError("La fecha de fin debe ser
posterior a la fecha de inicio.")

        cupos_raw = datos.get('cupos')

        if not validar_cupos(cupos_raw):
            raise ValueError("Los cupos deben ser un
entero mayor o igual a 0.")

        cupos = int(cupos_raw)

        # Guardar actividad
        actividad = db.create_actividad(
```

```
            id_creador=creador,
            nombre_actividad=datos['nombre_actividad'],
            descripcion=datos.get('descripcion', ''),
            categoria=datos.get('categoria', ''),
            ubicacion=datos.get('ubicacion', ''),
            lat=0,
            lng=0,
            fecha_hora_inicio=datos['fecha_hora_inicio'],
            fecha_hora_fin=datos.get('fecha_hora_fin'),
            cupos=datos.get('cupos') or 0,
            foto_actividad=foto.name if foto else '',
            estado='Activa',
            fecha_hora_creacion=datetime.datetime.now(),

fecha_hora_actualizacion=datetime.datetime.now(),
        )

        return actividad
```

Función crear_actividad

Los valores probados incluyen nombres vacíos, de un solo carácter, numéricos, correctos con más de 3 caracteres y correctos con espacios. Lo que se espera es que los nombres inválidos generen una excepción y los nombres válidos permitan la creación de de la actividad.

El test utiliza un diccionario de casos con sus valores esperados y recorre cada uno para verificar su comportamiento.

A continuación se presenta el test.

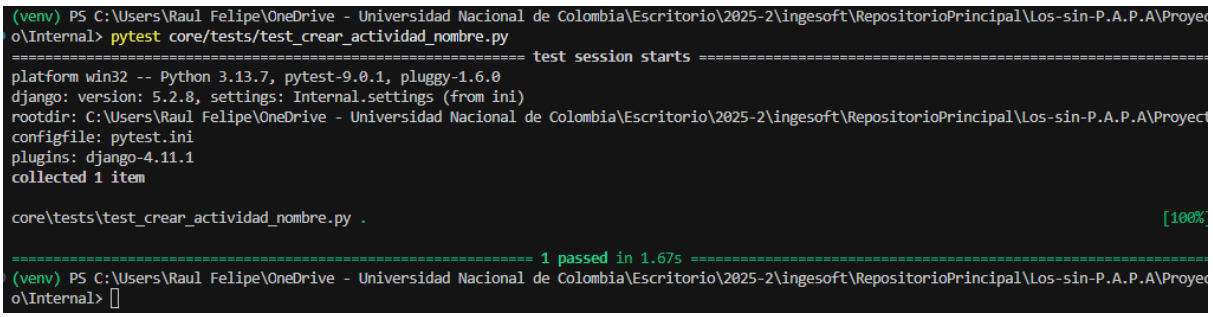
Python

```
"""
TEST: Validación del nombre de actividad en
ActividadService

Objetivo:
---------
Validar que la lógica de negocio para crear actividades
(ActividadService.crear_actividad)
aplique correctamente las reglas sobre el nombre de la
actividad.

Casos límite probados:
----------------------
1. Nombre vacío ("") → debe rechazarlo (no es válido)
2. Nombre muy corto ("A") → debe rechazarlo
3. Nombre numérico ("12345") → debe rechazarlo
4. Nombre válido simple ("Mi actividad") → debe aceptarlo
5. Nombre válido largo ("Actividad deportiva") → debe
aceptarlo

Técnicas empleadas:
-------------------
- Uso de pytest + pytest-django.
- Uso de `unittest.mock.patch` para reemplazar DB_Manager
con un mock
  y evitar interacción real con la base de datos.
- Validación de excepciones cuando el nombre es inválido.
- Validación de que `create_actividad` se llama cuando
los datos son válidos.
```

```
Este test es esencial en el sistema porque garantiza que
la creación
de actividades cumpla con las reglas de negocio antes de
intentar guardar
en la base de datos.
"""

import pytest
from unittest.mock import MagicMock, patch
from django.utils import timezone
from datetime import timedelta

from core.Negocio.actividad_service import
ActividadService


@pytest.mark.django_db
def test_crear_actividad_nombre():

    # Parcheamos DB_Manager dentro del servicio
    with
patch("core.Negocio.actividad_service.DB_Manager") as
MockDB:

        mock_db = MockDB.return_value

        # Mock del usuario obtenido desde la BD
        mock_usuario = MagicMock()
```

```
mock_db.get_usuario_by_nombre_usuario.return_value =
mock_usuario

        # Mock de creación de actividad (para evitar
insert real en BD)
        mock_db.create_actividad.return_value =
MagicMock()

        # Instanciamos el servicio
        service = ActividadService()

        # Mock del request con sesión
        request = MagicMock()
        request.session = {"username": "testuser"}

        ahora = timezone.now()

        # Casos límite
        casos = [
            ("", False),                    # Nombre
vacío → inválido
            ("Mi actividad", True),         # Válido
            ("A", False),                   # Muy corto
→ inválido
            ("12345", False),               # Numérico
→ inválido
            ("Actividad deportiva", True),  # Válido
        ]
```

```
        for nombre_caso, esperado in casos:

            datos = {
                "nombre_actividad": nombre_caso,
                "descripcion": "Algo",
                "categoria": "Ocio",
                "ubicacion": "Canchas",
                "fecha_hora_inicio": (ahora +
timedelta(hours=1)).isoformat(),
                "fecha_hora_fin": (ahora +
timedelta(hours=2)).isoformat(),
                "cupos": 10,
            }

            if not esperado:
                # Si se espera que falle → debe lanzar
excepción
                with pytest.raises(Exception):
                    service.crear_actividad(request,
datos)
            else:
                # Si es válido → NO debe lanzar excepción
                service.crear_actividad(request, datos)
                mock_db.create_actividad.assert_called()
```



En conclusión, este test demuestra que la función válida correctamente las reglas mínimas para aceptar un nombre válido y rechaza casos triviales o incorrectos.

El segundo test es el de validación de los cupos de las actividades.

Este segundo test evalúa la función validar_cupos y cómo ActividadService maneja dicho valor durante la creación de la actividad.

Las reglas verificadas por el test son:

- cupos negativos → inválido
- cupos igual a 0 → válido
- cupos entero positivo → válido
- cupos vacío → inválido
- cupos texto no convertible a entero → inválido

Para evitar la interacción con PostgreSQL, se parchea DB_Manager y se simula a un usuario que crea la actividad y la llamada a create_actividad.

```python
def validar_cupos(cupos_raw):
    """Valida el número de cupos."""
    if cupos_raw in ["", None]:
        return False

    try:
        cupos = int(cupos_raw)
    except ValueError:
        return False

    if cupos < 0:
        return False

    return True
```

Función validar_cupos

Python

```
"""
TEST: Validación del campo 'cupos' en
ActividadService.crear_actividad

Objetivo:
---------
Validar que la capa de negocio rechace cupos inválidos y
acepte cupos válidos.

Reglas evaluadas:
-----------------
1. cupos < 0 → inválido
2. cupos = 0 → válido (permite actividades sin cupos)
3. cupos > 0 → válido
4. cupos no numérico → inválido
5. cupos vacío → inválido
```

```
Casos límite probados:
----------------------
- -5      → inválido
- 0       → válido
- 10      → válido
- ""      → inválido
- "abc"   → inválido

Técnicas de testeo:
-------------------
- pytest + pytest-django
- uso de unittest.mock.patch para reemplazar DB_Manager y
evitar acceso a la BD
- validación con pytest.raises en casos inválidos
"""

import pytest
from unittest.mock import patch, MagicMock
from django.utils import timezone
from datetime import timedelta

from core.Negocio.actividad_service import
ActividadService



@pytest.mark.django_db
def test_crear_actividad_cupos():

    # Parcheamos DB_Manager para evitar consultas reales
```

```
    with
patch("core.Negocio.actividad_service.DB_Manager") as
MockDB:

        mock_db = MockDB.return_value

        # Mock del usuario retornado por la base de datos
        mock_usuario = MagicMock()

mock_db.get_usuario_by_nombre_usuario.return_value =
mock_usuario

        # Mock de creación de actividad
        mock_db.create_actividad.return_value =
MagicMock()

        service = ActividadService()
        request = MagicMock()
        request.session = {"username": "testuser"}

        ahora = timezone.now()

        casos = [
            (-5, False),
            (0, True),
            (10, True),
            ("", False),
            ("abc", False),
        ]
```

```
        for cupos_caso, esperado in casos:

            datos = {
                "nombre_actividad": "Prueba Cupos",
                "descripcion": "Test",
                "categoria": "Ensayo",
                "ubicacion": "Salón",
                "fecha_hora_inicio": (ahora +
timedelta(hours=1)).isoformat(),
                "fecha_hora_fin": (ahora +
timedelta(hours=2)).isoformat(),
                "cupos": cupos_caso,
            }

            if not esperado:
                # Si es caso inválido → debe lanzar
excepción
                with pytest.raises(ValueError):
                    service.crear_actividad(request,
datos)

            else:
                # Si es válido → no debe lanzar error
                service.crear_actividad(request, datos)
                mock_db.create_actividad.assert_called()
```

```
o\Internal> pytest core/tests/test_crear_actividad_cupos.py
============================= test session starts =============================
platform win32 -- Python 3.13.7, pytest-9.0.1, pluggy-1.6.0
django: version: 5.2.8, settings: Internal.settings (from ini)
rootdir: C:\Users\Raul Felipe\OneDrive - Universidad Nacional de Colombia\Escritorio\2025-2\ingesoft\RepositorioPrincipal\Los-sin-P.A.P.A\Proyecto
configfile: pytest.ini
plugins: django-4.11.1
collected 1 item

core\tests\test_crear_actividad_cupos.py .                               [100%]

============================== 1 passed in 1.60s ==============================
(venv) PS C:\Users\Raul Felipe\OneDrive - Universidad Nacional de Colombia\Escritorio\2025-2\ingesoft\RepositorioPrincipal\Los-sin-P.A.P.A\Proyect
o\Internal>
```

Gracias a estos casos se confirma que el servicio rechaza valores no numéricos o negativos y acepta 0 o enteros positivos, cumpliendo con la lógica esperada.

El tercer test corresponde a la validación de fecha de inicio y fin de las actividades.

Este test valida que las fechas cumplan con los criterios establecidos: La actividad debe tener fecha de inicio, la fecha de fin debe existir y la fecha de fin debe ser posterior a la fecha de inicio. El archivo de test genera distintas combinaciones usando timezone.now() y timedelta:

Casos evaluados:

- fecha_inicio vacía → inválido
- fecha_fin vacía → inválido
- fecha_fin = fecha_inicio → inválido
- fecha_fin anterior a fecha_inicio → inválido
- fecha_fin posterior a fecha_inicio → válido

Como en los tests anteriores, DB_Manager es reemplazado con mocks, aislando la lógica de negocio.

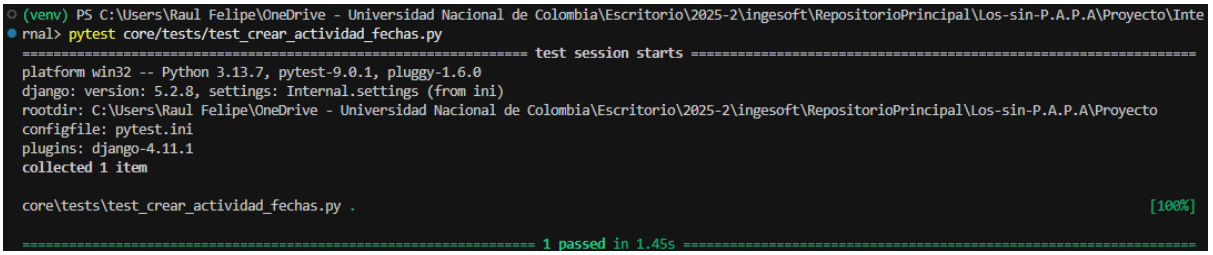
```python
fecha_inicio_raw = datos.get('fecha_hora_inicio')
fecha_fin_raw = datos.get('fecha_hora_fin')

if not fecha_inicio_raw:
    raise ValueError("La fecha de inicio es obligatoria.")

if not fecha_fin_raw:
    raise ValueError("La fecha de fin es obligatoria.")

try:
    fecha_inicio = datetime.datetime.fromisoformat(fecha_inicio_raw)
    fecha_fin = datetime.datetime.fromisoformat(fecha_fin_raw)
except Exception:
    raise ValueError("Las fechas deben estar en formato ISO YYYY-MM-DDTHH:MM.")

if fecha_fin <= fecha_inicio:
    raise ValueError("La fecha de fin debe ser posterior a la fecha de inicio.")
```

Python

```python
"""
TEST: Validación de fechas en
ActividadService.crear_actividad

Reglas probadas:
----------------
1. La fecha de inicio es obligatoria.
2. La fecha de fin es obligatoria.
3. La fecha de fin debe ser mayor que la fecha de inicio.

Casos límite:
-------------
- falta fecha inicio      → inválido
- falta fecha fin         → inválido
- fecha fin < inicio      → inválido
- fecha fin = inicio      → inválido
- rango válido            → válido
"""

import pytest
from unittest.mock import patch, MagicMock
from django.utils import timezone
from datetime import timedelta

from core.Negocio.actividad_service import
ActividadService



@pytest.mark.django_db
```

```
def test_crear_actividad_fechas():

    with
patch("core.Negocio.actividad_service.DB_Manager") as
MockDB:

        mock_db = MockDB.return_value

        # Mock usuario creador
        mock_usuario = MagicMock()

mock_db.get_usuario_by_nombre_usuario.return_value =
mock_usuario

        # Mock creación de actividad
        mock_db.create_actividad.return_value =
MagicMock()

        service = ActividadService()

        request = MagicMock()
        request.session = {"username": "testuser"}

        ahora = timezone.now()

        casos = [
            (None, ahora.isoformat(), False),    # Falta
inicio
            (ahora.isoformat(), None, False),    # Falta
fin
```

```
            (ahora.isoformat(), (ahora -
timedelta(hours=1)).isoformat(), False), # Fin antes de
inicio
            (ahora.isoformat(), ahora.isoformat(),
False), # Fin = inicio
            (ahora.isoformat(), (ahora +
timedelta(hours=1)).isoformat(), True),  # Correcto
        ]

        for fecha_inicio, fecha_fin, valido in casos:

            datos = {
                "nombre_actividad": "Test Fechas",
                "descripcion": "Test",
                "categoria": "Ensayo",
                "ubicacion": "Salón",
                "fecha_hora_inicio": fecha_inicio,
                "fecha_hora_fin": fecha_fin,
                "cupos": 5,
            }

            if not valido:
                with pytest.raises(Exception):
                    service.crear_actividad(request,
datos)
            else:
                service.crear_actividad(request, datos)
                mock_db.create_actividad.assert_called()
```



Este test garantiza que la lógica temporal del sistema sea estricta y que ninguna actividad pueda registrarse con fechas inconsistentes.