Pruebas de Atomicidad para Create y Update

Para probar la atomicidad de las operaciones CREATE y UPDATE, puedes implementar pruebas que verifiquen que todas las operaciones dentro de una transacción se completen con éxito o se reviertan por completo en caso de error.

1. Pruebas para CREATE (Creación)

a) Prueba de creación exitosa (transacción completa)

python

import pytest

from unittest.mock import AsyncMock, patch

from sqlalchemy.ext.asyncio import AsyncSession

from fastapi import HTTPException

from uuid import UUID

@pytest.mark.asyncio

async def test\_create\_payment\_term\_success():

"""Prueba que la creación se complete exitosamente con todas las operaciones"""

mock\_db = AsyncMock(spec=AsyncSession)

mock\_commit = AsyncMock()

mock\_db.commit = mock\_commit

*# Configurar mocks para una creación exitosa*

with patch('app.crud.payment\_term.validate\_payment\_term', AsyncMock(return\_value=None)):

with patch('app.crud.payment\_term.sanitize\_input', side\_effect=lambda x: x):

with patch('app.models.payment\_term.PaymentTerm') as mock\_model:

mock\_instance = AsyncMock()

mock\_model.return\_value = mock\_instance

with patch('app.crud.payment\_term.log\_action', AsyncMock(return\_value="mock\_log")):

with patch('app.crud.payment\_term.get\_audit\_level', AsyncMock(return\_value=2)):

*# Ejecutar la función*

from app.crud import payment\_term

result, log = await payment\_term.create\_payment\_term(

mock\_db,

{"code": "TEST", "name": "Test Term", "active": True},

UUID("12345678-1234-5678-1234-567812345678")

)

*# Verificar que se llamó a flush pero no a commit (la transacción la maneja el endpoint)*

assert mock\_db.flush.called

assert not mock\_db.commit.called *# El commit lo hace el endpoint*

assert log == "mock\_log"

b) Prueba de creación con error (transacción revertida)

python

@pytest.mark.asyncio

async def test\_create\_payment\_term\_validation\_failure():

"""Prueba que se revierte la transacción cuando falla la validación"""

mock\_db = AsyncMock(spec=AsyncSession)

mock\_rollback = AsyncMock()

mock\_db.rollback = mock\_rollback

*# Configurar validación para que falle*

with patch('app.crud.payment\_term.validate\_payment\_term',

AsyncMock(side\_effect=HTTPException(status\_code=400, detail="Validation error"))):

*# Verificar que se lanza la excepción*

from app.crud import payment\_term

with pytest.raises(HTTPException) as exc\_info:

await payment\_term.create\_payment\_term(

mock\_db,

{"code": "INVALID", "name": "Test Term"},

UUID("12345678-1234-5678-1234-567812345678")

)

*# Verificar que se hizo rollback*

assert exc\_info.value.status\_code == 400

c) Prueba de error en base de datos

python

@pytest.mark.asyncio

async def test\_create\_payment\_term\_database\_error():

"""Prueba que se revierte la transacción cuando hay error de base de datos"""

mock\_db = AsyncMock(spec=AsyncSession)

mock\_db.flush = AsyncMock(side\_effect=IntegrityError("Error", "Error", "Error"))

mock\_rollback = AsyncMock()

mock\_db.rollback = mock\_rollback

*# Configurar mocks*

with patch('app.crud.payment\_term.validate\_payment\_term', AsyncMock(return\_value=None)):

with patch('app.crud.payment\_term.sanitize\_input', side\_effect=lambda x: x):

*# Verificar que se lanza la excepción*

from app.crud import payment\_term

with pytest.raises(HTTPException) as exc\_info:

await payment\_term.create\_payment\_term(

mock\_db,

{"code": "TEST", "name": "Test Term"},

UUID("12345678-1234-5678-1234-567812345678")

)

*# Verificar que se hizo rollback y que se lanza la excepción correcta*

assert exc\_info.value.status\_code == 409

2. Pruebas para UPDATE (Actualización)

a) Prueba de actualización exitosa

python

@pytest.mark.asyncio

async def test\_update\_payment\_term\_success():

"""Prueba que la actualización se complete exitosamente"""

mock\_db = AsyncMock(spec=AsyncSession)

mock\_commit = AsyncMock()

mock\_db.commit = mock\_commit

*# Mock de un payment term existente*

mock\_payment\_term = AsyncMock()

mock\_payment\_term.id = UUID("12345678-1234-5678-1234-567812345678")

mock\_payment\_term.code = "OLD\_CODE"

mock\_payment\_term.name = "Old Name"

*# Configurar mocks*

with patch('app.crud.payment\_term.validate\_payment\_term', AsyncMock(return\_value=None)):

with patch('app.crud.payment\_term.sanitize\_input', side\_effect=lambda x: x):

with patch('app.crud.payment\_term.get\_payment\_term\_by\_id',

AsyncMock(return\_value=mock\_payment\_term)):

with patch('app.crud.payment\_term.log\_action', AsyncMock(return\_value="mock\_log")):

with patch('app.crud.payment\_term.get\_audit\_level', AsyncMock(return\_value=2)):

*# Ejecutar la función de actualización*

from app.crud import payment\_term

result, log = await payment\_term.update\_payment\_term(

mock\_db,

mock\_payment\_term.id,

{"code": "NEW\_CODE", "name": "New Name", "active": False},

UUID("12345678-1234-5678-1234-567812345678")

)

*# Verificar que se actualizaron los campos*

assert result.code == "NEW\_CODE"

assert result.name == "New Name"

assert result.active == False

assert log == "mock\_log"

b) Prueba de actualización con error

python

@pytest.mark.asyncio

async def test\_update\_payment\_term\_with\_error():

"""Prueba que se revierte la transacción cuando falla la actualización"""

mock\_db = AsyncMock(spec=AsyncSession)

mock\_db.flush = AsyncMock(side\_effect=SQLAlchemyError("DB error"))

mock\_rollback = AsyncMock()

mock\_db.rollback = mock\_rollback

*# Mock de un payment term existente*

mock\_payment\_term = AsyncMock()

mock\_payment\_term.id = UUID("12345678-1234-5678-1234-567812345678")

*# Configurar mocks*

with patch('app.crud.payment\_term.validate\_payment\_term', AsyncMock(return\_value=None)):

with patch('app.crud.payment\_term.sanitize\_input', side\_effect=lambda x: x):

with patch('app.crud.payment\_term.get\_payment\_term\_by\_id',

AsyncMock(return\_value=mock\_payment\_term)):

*# Verificar que se lanza la excepción*

from app.crud import payment\_term

with pytest.raises(HTTPException) as exc\_info:

await payment\_term.update\_payment\_term(

mock\_db,

mock\_payment\_term.id,

{"code": "NEW\_CODE", "name": "New Name"},

UUID("12345678-1234-5678-1234-567812345678")

)

*# Verificar que se hizo rollback y que se lanza la excepción correcta*

assert exc\_info.value.status\_code == 500

3. Pruebas de Integración para el Endpoint

a) Prueba de endpoint exitoso

python

@pytest.mark.asyncio

async def test\_create\_payment\_term\_endpoint\_success():

"""Prueba que el endpoint complete la transacción correctamente"""

from app.main import app

from fastapi.testclient import TestClient

client = TestClient(app)

*# Mock de autenticación y base de datos*

with patch('app.routes.payment\_term.get\_current\_user', AsyncMock(return\_value={"id": "user\_id"})):

with patch('app.routes.payment\_term.get\_async\_db', AsyncMock()):

with patch('app.crud.payment\_term.create\_payment\_term',

AsyncMock(return\_value=(AsyncMock(), "mock\_log"))):

*# Realizar la petición*

response = client.post(

"/payment-terms/",

json={"code": "TEST", "name": "Test Term", "active": True},

headers={"Authorization": "Bearer test\_token"}

)

*# Verificar que la respuesta es exitosa*

assert response.status\_code == 200

b) Prueba de endpoint con error

python

@pytest.mark.asyncio

async def test\_create\_payment\_term\_endpoint\_failure():

"""Prueba que el endpoint revierte la transacción cuando hay error"""

from app.main import app

from fastapi.testclient import TestClient

client = TestClient(app)

*# Mock de autenticación y base de datos*

with patch('app.routes.payment\_term.get\_current\_user', AsyncMock(return\_value={"id": "user\_id"})):

with patch('app.routes.payment\_term.get\_async\_db', AsyncMock()) as mock\_db:

mock\_db.rollback = AsyncMock()

*# Configurar la función para que falle*

with patch('app.crud.payment\_term.create\_payment\_term',

AsyncMock(side\_effect=HTTPException(status\_code=400, detail="Error"))):

*# Realizar la petición*

response = client.post(

"/payment-terms/",

json={"code": "INVALID", "name": "Test Term"},

headers={"Authorization": "Bearer test\_token"}

)

*# Verificar que la respuesta tiene error*

assert response.status\_code == 400

*# Verificar que se llamó a rollback*

assert mock\_db.rollback.called

4. Pruebas de Validación de Atomicidad

a) Prueba de que no se persisten datos parciales

python

@pytest.mark.asyncio

async def test\_atomicity\_no\_partial\_data():

"""Prueba que no se persisten datos cuando falla una operación"""

mock\_db = AsyncMock(spec=AsyncSession)

mock\_db.flush = AsyncMock()

mock\_rollback = AsyncMock()

mock\_db.rollback = mock\_rollback

*# Contador para simular fallo después de la primera operación*

call\_count = 0

def mock\_flush():

nonlocal call\_count

call\_count += 1

if call\_count > 1: *# Fallar en el segundo flush (simulando error en el log)*

raise SQLAlchemyError("Error en segunda operación")

mock\_db.flush = mock\_flush

*# Configurar mocks*

with patch('app.crud.payment\_term.validate\_payment\_term', AsyncMock(return\_value=None)):

with patch('app.crud.payment\_term.sanitize\_input', side\_effect=lambda x: x):

with patch('app.models.payment\_term.PaymentTerm') as mock\_model:

mock\_instance = AsyncMock()

mock\_model.return\_value = mock\_instance

with patch('app.crud.payment\_term.log\_action',

AsyncMock(return\_value=AsyncMock())):

with patch('app.crud.payment\_term.get\_audit\_level',

AsyncMock(return\_value=2)):

*# Verificar que se lanza la excepción*

from app.crud import payment\_term

with pytest.raises(SQLAlchemyError):

await payment\_term.create\_payment\_term(

mock\_db,

{"code": "TEST", "name": "Test Term"},

UUID("12345678-1234-5678-1234-567812345678")

)

*# Verificar que se hizo rollback*

assert mock\_db.rollback.called

5. Configuración de Pruebas

Asegúrate de tener en tu conftest.py o configuración de pruebas:

python

import asyncio

import pytest

from sqlalchemy.ext.asyncio import create\_async\_engine, AsyncSession

from sqlalchemy.orm import sessionmaker

@pytest.fixture

async def async\_db\_session():

"""Session for SQLAlchemy with async support"""

*# Usar base de datos en memoria para pruebas*

engine = create\_async\_engine("sqlite+aiosqlite:///:memory:", echo=False)

async\_session = sessionmaker(

engine, class\_=AsyncSession, expire\_on\_commit=False

)

async with async\_session() as session:

*# Crear tablas*

from app.models.base import Base

async with engine.begin() as conn:

await conn.run\_sync(Base.metadata.create\_all)

yield session

await engine.dispose()

@pytest.fixture(autouse=True)

def setup\_test\_environment():

"""Setup test environment"""

*# Configuración adicional para pruebas*

pass

Consejos para las Pruebas de Atomicidad:

1. **Prueba ambos caminos**: Asegúrate de probar tanto el camino exitoso como los caminos de error.
2. **Verifica el estado de la base de datos**: Después de cada prueba, verifica que los datos estén en el estado esperado.
3. **Usa bases de datos en memoria**: Para pruebas, usa SQLite en memoria para mayor velocidad y aislamiento.
4. **Prueba escenarios edge**: Considera casos como timeouts de conexión, restricciones de unique constraints, y validaciones complejas.
5. **Mockea dependencias externas**: Para aislar las pruebas, mockea servicios externos como APIs de terceros.

Estas pruebas te ayudarán a garantizar que tus operaciones CREATE y UPDATE sean atómicas, es decir, que se completen por completo o se reviertan por completo en caso de error.

# Manual SUPER Detallado de Testing en Postman

## 📋 TABLA DE CONTENIDOS

1. Configuración Inicial
2. Crear Colección
3. Configurar Variables
4. Crear Request
5. Pre-request Script
6. Tests
7. Ejecutar Pruebas
8. Ver Resultados
9. Solución de Problemas

## 🎯 1. Configuración Inicial

### 1.1 Descargar e Instalar Postman

* Ve a <https://www.postman.com/downloads/>
* Descarga la versión para tu sistema operativo
* Instala y abre Postman

### 1.2 Crear Cuenta (Opcional pero Recomendado)

* Click en "Sign Up" en la esquina superior derecha
* Crea una cuenta gratuita para sincronizar tus trabajos

<https://i.imgur.com/5VnW2cL.png>

## 🎯 2. Crear Colección

### 2.1 Click en "Collections" en el menú izquierdo

### 2.2 Click en el botón "+" para crear nueva colección

### 2.3 Nombrar la colección: "API Payment Terms Tests"

### 2.4 Click en "Create"

<https://i.imgur.com/8Xw7R3k.png>

## 🎯 3. Configurar Variables

### 3.1 Click en el ojo 👁️ al lado de "Environment" (arriba a la derecha)

### 3.2 Click en "Add Environment"

### 3.3 Nombrar: "API Testing Environment"

### 3.4 Agregar estas variables:

| **Variable** | **Initial Value** | **Current Value** |
| --- | --- | --- |
| base\_url | http://localhost:8000 | http://localhost:8000 |
| auth\_token | tu\_token\_aqui | tu\_token\_aqui |
| request\_count | 0 | 0 |
| last\_created\_id | `` | `` |

### 3.5 Click en "Add"

<https://i.imgur.com/9jK7W3c.png>

### 3.6 Seleccionar el environment creado en el dropdown

## 🎯 4. Crear Request

### 4.1 Click en la colección "API Payment Terms Tests"

### 4.2 Click en "Add a request"

### 4.3 Configurar:

* **Method**: POST
* **URL**: {{base\_url}}/payment-terms/

### 4.4 Headers:

* Click en "Headers"
* Agregar:

| **Key** | **Value** |
| --- | --- |
| Authorization | Bearer {{auth\_token}} |
| Content-Type | application/json |

### 4.5 Body:

* Click en "Body"
* Seleccionar "raw" y "JSON"
* Pegar este contenido:

json

{

"code": "TEST01",

"name": "Test Payment Term",

"description": "Test Description",

"active": true

}

<https://i.imgur.com/8vG7Q4E.png>

## 🎯 5. Pre-request Script

### 5.1 Click en la pestaña "Scripts" debajo de la URL

### 5.2 Verás dos sub-pestañas: "Pre-request" y "Post-response"

### 5.3 Asegúrate de estar en "Pre-request"

### 5.4 Pegar este código:

javascript

*// ==================== PRE-REQUEST SCRIPT ====================*

*// Incrementar contador de requests*

const requestCount = parseInt(pm.environment.get("request\_count") || "0");

pm.environment.set("request\_count", requestCount + 1);

console.log("🔄 Request número:", requestCount + 1);

*// Hacer único cada request para evitar conflictos*

const uniqueCode = "TEST" + (requestCount + 1);

const requestData = {

"code": uniqueCode,

"name": "Test Payment Term " + (requestCount + 1),

"description": "Testing - Request " + (requestCount + 1),

"active": true

};

*// Actualizar el body del request*

pm.request.body.raw = JSON.stringify(requestData);

console.log("📦 Body actualizado:", requestData);

### 5.5 Click en "Save" (Ctrl+S)

<https://i.imgur.com/9QYzK9c.png>

## 🎯 6. Tests

### 6.1 En la misma pestaña "Scripts", click en "Post-response"

### 6.2 Pegar este código:

javascript

*// ==================== TESTS ====================*

*// Obtener el número de request actual*

const currentRequestCount = parseInt(pm.environment.get("request\_count") || "0");

console.log("🔢 Request número:", currentRequestCount);

console.log("📊 Status code:", pm.response.code);

console.log("⏰ Response time:", pm.response.responseTime + "ms");

*// Test principal de rate limiting*

pm.test(`✅ Rate Limiting Test - Request ${currentRequestCount}`, function() {

if (currentRequestCount > 5) {

*// Requests 6-10 deberían recibir 429 (Too Many Requests)*

pm.response.to.have.status(429);

pm.expect(pm.response.text().toLowerCase()).to.include("too many");

} else {

*// Requests 1-5 deberían recibir 200 (OK)*

pm.response.to.have.status(200);

}

});

*// Tests adicionales para requests exitosos*

if (currentRequestCount <= 5 && pm.response.code === 200) {

pm.test("📋 Response tiene JSON body", function() {

pm.response.to.be.json;

});

pm.test("🔍 Response contiene campos requeridos", function() {

const responseData = pm.response.json();

pm.expect(responseData).to.have.property('id');

pm.expect(responseData).to.have.property('code');

pm.expect(responseData).to.have.property('name');

pm.expect(responseData).to.have.property('active');

});

*// Guardar el ID del último creado para pruebas futuras*

if (responseData.id) {

pm.environment.set("last\_created\_id", responseData.id);

console.log("💾 ID guardado:", responseData.id);

}

}

*// Auto-reset después de 10 requests*

if (currentRequestCount >= 10) {

pm.environment.set("request\_count", "0");

console.log("🔄 Contador reseteado a 0");

}

console.log("=".repeat(50));

### 6.3 Click en "Save" (Ctrl+S)

<https://i.imgur.com/7V5DpMk.png>

## 🎯 7. Ejecutar Pruebas

### 🎪 MÉTODO 1: Manual (Recomendado para empezar)

#### **7.1 Click en el botón "Send" 10 veces rápidamente**

#### **7.2 Observa los resultados en la parte inferior**

### 🎪 MÉTODO 2: Collection Runner

#### **7.3 Click en la colección "API Payment Terms Tests"**

#### **7.4 Click en "Run" (botón con play ▶️)**

#### **7.5 En el modal:**

* **Iterations**: 10
* **Delay**: 0 ms
* **Data**: No seleccionar archivo
* **Persist variables**: ✅ Checked

#### **7.6 Click en "Run API Payment Terms Tests"**

<https://i.imgur.com/3kL8W2c.png>

## 🎯 8. Ver Resultados

### 8.1 Después de hacer "Send", mira la pestaña "Test Results" abajo

### 8.2 Deberías ver:

text

✅ Rate Limiting Test - Request 1 (200 OK)

✅ Rate Limiting Test - Request 2 (200 OK)

✅ Rate Limiting Test - Request 3 (200 OK)

✅ Rate Limiting Test - Request 4 (200 OK)

✅ Rate Limiting Test - Request 5 (200 OK)

✅ Rate Limiting Test - Request 6 (429 Too Many Requests)

✅ Rate Limiting Test - Request 7 (429 Too Many Requests)

✅ Rate Limiting Test - Request 8 (429 Too Many Requests)

✅ Rate Limiting Test - Request 9 (429 Too Many Requests)

✅ Rate Limiting Test - Request 10 (429 Too Many Requests)

### 8.3 También puedes ver los console.logs en:

* Click en "Console" en la esquina inferior izquierda
* O ve a "View" > "Show Postman Console"

<https://i.imgur.com/5jT6Z4f.png>

## 🎯 9. Solución de Problemas

### ❌ Problema: "Error: Invalid token"

**Solución:**

1. Obtén un token válido de tu API
2. Ve a Environments > "API Testing Environment"
3. Actualiza auth\_token con tu token real
4. Formato: Bearer tu\_token\_jwt\_aqui

### ❌ Problema: Todos los requests dan 200

**Solución:**

* Tu API no tiene rate limiting configurado
* Necesitas implementarlo en FastAPI:

bash

pip install slowapi

python

*# En tu FastAPI*

from slowapi import Limiter

from slowapi.util import get\_remote\_address

from slowapi.middleware import SlowAPIMiddleware

limiter = Limiter(key\_func=get\_remote\_address)

app.state.limiter = limiter

app.add\_middleware(SlowAPIMiddleware)

@router.post("/payment-terms/")

@limiter.limit("5/minute")

async def create\_payment\_term(request: Request, payment\_term\_in: PaymentTermCreate):

*# tu código aquí*

### ❌ Problema: "Unique constraint violation"

**Solución:**

* El Pre-request Script ya hace únicos los códigos
* Verifica que esté funcionando revisando los console.logs

### ❌ Problema: No veo la pestaña "Scripts"

**Solución:**

* Asegúrate de tener la versión más reciente de Postman
* Haz click en "Body" y luego verás "Scripts" al lado

## 🎯 EXTRA: Probar Endpoints GET

### Crear nuevo request para GET:

1. Click en "Add a request" en la colección
2. Method: GET
3. URL: {{base\_url}}/payment-terms/{{last\_created\_id}}
4. Headers: Mismo Authorization
5. Tests:

javascript

*// Tests para GET*

pm.test("Status code is 200", function () {

pm.response.to.have.status(200);

});

pm.test("Response has JSON body", function () {

pm.response.to.be.json;

});

pm.test("Response contains correct data", function () {

const responseData = pm.response.json();

pm.expect(responseData).to.have.property('id');

pm.expect(responseData).to.have.property('code');

pm.expect(responseData).to.have.property('name');

});

## ✅ Checklist Final

* Postman instalado
* Colección creada
* Environment configurado
* Request POST creado
* Pre-request Script configurado
* Tests configurados
* Token de autenticación configurado
* Ejecutado 10 veces
* Resultados verificados