**1. Colección de Pruebas de Ciberseguridad en Postman**

Crea una nueva colección llamada "Security Tests" y agrega estas pruebas:

**Prueba 1: XSS (Cross-Site Scripting)**

javascript

*// POST {{base\_url}}/payment-terms/*

{

"code": "XSS\_TEST",

"name": "<script>alert('XSS')</script>",

"description": "Prueba de XSS <img src=x onerror=alert('XSS')>",

"active": true

}

**Assertion:**

javascript

pm.test("Bloquea XSS en name", function() {

pm.response.to.have.status(400);

pm.expect(pm.response.text()).to.include("potencialmente peligroso");

});

**Prueba 2: SQL Injection**

javascript

*// POST {{base\_url}}/payment-terms/*

{

"code": "SQL' OR '1'='1",

"name": "TEST'; DROP TABLE payment\_terms; --",

"description": "TEST UNION SELECT \* FROM users",

"active": true

}

**Assertion:**

javascript

pm.test("Bloquea SQL Injection", function() {

pm.response.to.have.status(400);

pm.expect(pm.response.text()).to.include("patrones sospechosos");

});

**Prueba 3: Directory Traversal**

javascript

*// POST {{base\_url}}/payment-terms/*

{

"code": "TRAVERSAL",

"name": "../../etc/passwd",

"description": "../../windows/system32",

"active": true

}

**Assertion:**

javascript

pm.test("Bloquea Directory Traversal", function() {

pm.response.to.have.status(400);

});

**Prueba 4: Caracteres Peligrosos**

javascript

*// POST {{base\_url}}/payment-terms/*

{

"code": "BAD;CHARS",

"name": "Test|&$%",

"description": "Caracteres ; -- /\* \*/",

"active": true

}

**Assertion:**

javascript

pm.test("Bloquea caracteres especiales", function() {

pm.response.to.have.status(400);

});

**2. Pruebas de Autenticación y Autorización**

**Prueba 5: Token Inválido**

javascript

*// Headers*

Authorization: Bearer token\_invalido

**Assertion:**

javascript

pm.test("Rechaza token inválido", function() {

pm.response.to.have.status(401);

});

**Prueba 6: Sin Token**

javascript

*// Headers: Sin Authorization*

**Assertion:**

javascript

pm.test("Rechaza petición sin token", function() {

pm.response.to.have.status(401);

});

**3. Pruebas de Rate Limiting**

**Prueba 7: Rate Limiting**

javascript

*// Ejecutar esta prueba 10 veces rápidamente*

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

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

*// Headers con token válido*

**Assertion:**

javascript

pm.test("Rate limiting funciona", function() {

if (pm.environment.get("request\_count") > 5) {

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

}

});

**4. Pruebas de Headers de Seguridad**

**Prueba 8: Headers de Seguridad**

javascript

*// GET {{base\_url}}/payment-terms/*

**Assertion:**

javascript

pm.test("Headers de seguridad presentes", function() {

pm.expect(pm.response.headers.get('X-Content-Type-Options')).to.eql('nosniff');

pm.expect(pm.response.headers.get('X-Frame-Options')).to.eql('DENY');

pm.expect(pm.response.headers.get('Strict-Transport-Security')).to.include('max-age=');

});

**5. Pruebas de Validación de Datos**

**Prueba 9: Campos Demasiado Largos**

javascript

{

"code": "CODIGOMUYLARGOQUEEXEDELIMITE",

"name": "N".repeat(150),

"description": "D".repeat(600),

"active": true

}

**Assertion:**

javascript

pm.test("Valida longitud máxima", function() {

pm.response.to.have.status(400);

});

**Prueba 10: Campos Vacíos/Nulos**

javascript

{

"code": "",

"name": null,

"description": " ",

"active": true

}

**Assertion:**

javascript

pm.test("Valida campos requeridos", function() {

pm.response.to.have.status(400);

});

**6. Script de Automatización**

Crea un **Pre-request Script** para pruebas automáticas:

javascript

*// Pre-request Script para variables dinámicas*

const attackVectors = [

"<script>alert('XSS')</script>",

"'; DROP TABLE users; --",

"../../etc/passwd",

"{{$randomFirstName}}; SELECT \* FROM users",

"| & % $ # @ !",

"OR 1=1--"

];

const randomVector = attackVectors[Math.floor(Math.random() \* attackVectors.length)];

pm.environment.set("attack\_vector", randomVector);

**7. Variables de Entorno en Postman**

Crea variables de entorno:

javascript

base\_url: http://localhost:8000/api/v1

auth\_token: tu\_token\_jwt\_aqui

admin\_token: token\_de\_administrador

**8. Ejecución de Pruebas**

**Ejecutar todas las pruebas:**

1. Abre la colección "Security Tests"
2. Click en "Run collection"
3. Selecciona el ambiente con las variables
4. Click "Run"

**Monitoreo de resultados:**

* ✅ Verde: Vulnerabilidad prevenida
* ❌ Rojo: Vulnerabilidad detectada
* ⚠️ Amarillo: Revisión manual necesaria

**9. Exportar Resultados**

Después de ejecutar, exporta resultados:

1. Click en "Export Results"
2. Formato JSON/HTML
3. Comparte con el equipo

**Ejemplo de Reporte Automatizado:**

javascript

*// Tests script para reporte*

pm.test("Security Test Suite Summary", function() {

const tests = pm.response.json();

let passed = 0;

let failed = 0;

tests.forEach(test => {

if (test.status === "passed") passed++;

else failed++;

});

console.log(`✅ Passed: ${passed}, ❌ Failed: ${failed}`);

if (failed > 0) {

pm.expect.fail(`Se encontraron ${failed} vulnerabilidades de seguridad`);

}

});

**Tips Adicionales:**

1. **Programa ejecuciones** nightly con Postman Monitor
2. **Integra con CI/CD** usando Newman
3. **Actualiza vectores** de ataque regularmente
4. **Prueba diferentes roles** de usuario
5. **Documenta resultados** para auditorías

¡Con estas pruebas tendrás una **cobertura completa** de ciberseguridad para tu API! 🛡️

Cual debe ser la respuesta para cada una de las pruebas?

Para cada prueba de ciberseguridad, la respuesta **DEBE SER CONSISTENTE** y **NO revelar información sensible**. Aquí tienes el detalle:

**🔴 Respuestas para Pruebas Ofensivas (Ataques)**

**1. XSS (Cross-Site Scripting)**

**✅ Respuesta CORRECTA:**

http

HTTP/2 400 Bad Request

Content-Type: application/json

{

"detail": "El campo 'name' contiene código potencialmente peligroso (XSS)"

}

**❌ Respuesta INCORRECTA:**

http

HTTP/2 200 OK # ❌ Nunca debería ser 200

**2. SQL Injection**

**✅ Respuesta CORRECTA:**

http

HTTP/2 400 Bad Request

Content-Type: application/json

{

"detail": "El campo 'code' contiene patrones sospechosos de SQL injection"

}

**❌ Respuesta INCORRECTA:**

http

HTTP/2 500 Internal Server Error # ❌ Revela error de BD

# Con mensaje: "ERROR: syntax error at or near \"DROP\""

**3. Directory Traversal**

**✅ Respuesta CORRECTA:**

http

HTTP/2 400 Bad Request

Content-Type: application/json

{

"detail": "El campo 'name' contiene patrones de traversal no permitidos"

}

**4. Caracteres Peligrosos**

**✅ Respuesta CORRECTA:**

http

HTTP/2 400 Bad Request

Content-Type: application/json

{

"detail": "El campo 'code' contiene caracteres no permitidos: ';'"

}

**🔐 Respuestas para Autenticación**

**5. Token Inválido**

**✅ Respuesta CORRECTA:**

http

HTTP/2 401 Unauthorized

Content-Type: application/json

WWW-Authenticate: Bearer

{

"detail": "No se pudieron validar las credenciales"

}

**❌ Respuesta INCORRECTA:**

http

HTTP/2 200 OK # ❌ Nunca dar acceso

# O mensajes específicos: "Token expirado", "Token inválido"

**6. Sin Token**

**✅ Respuesta CORRECTA:**

http

HTTP/2 401 Unauthorized

Content-Type: application/json

WWW-Authenticate: Bearer

{

"detail": "No autenticado"

}

**⏱️ Respuestas para Rate Limiting**

**7. Rate Limiting (Demasiadas peticiones)**

**✅ Respuesta CORRECTA:**

http

HTTP/2 429 Too Many Requests

Content-Type: application/json

Retry-After: 60

{

"detail": "Demasiadas peticiones. Intente nuevamente en 60 segundos."

}

**🛡️ Respuestas para Headers de Seguridad**

**8. Headers de Seguridad**

**✅ Respuesta CORRECTA** (Debe incluir estos headers):

http

HTTP/2 200 OK

X-Content-Type-Options: nosniff

X-Frame-Options: DENY

Strict-Transport-Security: max-age=31536000; includeSubDomains

X-XSS-Protection: 1; mode=block

Content-Security-Policy: default-src 'self'

**📏 Respuestas para Validación de Datos**

**9. Campos Demasiado Largos**

**✅ Respuesta CORRECTA:**

http

HTTP/2 400 Bad Request

Content-Type: application/json

{

"detail": "El campo 'code' no puede exceder 10 caracteres"

}

**10. Campos Vacíos/Nulos**

**✅ Respuesta CORRECTA:**

http

HTTP/2 400 Bad Request

Content-Type: application/json

{

"detail": "El campo 'code' es requerido"

}

**🎯 Resumen de Códigos HTTP Esperados**

| **Tipo de Prueba** | **Código HTTP Esperado** | **Body Esperado** |
| --- | --- | --- |
| **XSS** | 400 Bad Request | Mensaje genérico de validación |
| **SQL Injection** | 400 Bad Request | Mensaje genérico de validación |
| **Directory Traversal** | 400 Bad Request | Mensaje genérico de validación |
| **Caracteres Peligrosos** | 400 Bad Request | Mensaje específico del caracter |
| **Token Inválido** | 401 Unauthorized | Mensaje genérico de autenticación |
| **Sin Token** | 401 Unauthorized | Mensaje genérico de autenticación |
| **Rate Limiting** | 429 Too Many Requests | Mensaje con tiempo de espera |
| **Headers Seguridad** | 200 OK | Headers de seguridad presentes |
| **Validación Datos** | 400 Bad Request | Mensaje específico de validación |

**⚠️ Reglas de Oro para Respuestas Seguras:**

**1. Nunca reveles información técnica:**

javascript

*// ❌ MAL*

{

"error": "SQL syntax error near 'DROP TABLE'",

"query": "SELECT \* FROM users WHERE id = '1' OR 1=1--'"

}

*// ✅ BIEN*

{

"detail": "Error de validación en los datos enviados"

}

**2. Usa códigos HTTP consistentes:**

* 400 para errores de validación del cliente
* 401 para problemas de autenticación
* 403 para problemas de autorización
* 429 para rate limiting
* 500 solo para errores internos reales

**3. Mensajes genéricos pero útiles:**

javascript

*// ✅ Balance perfecto*

{

"detail": "El campo 'username' contiene caracteres no permitidos"

}

*// ❌ Demasiado genérico*

{

"detail": "Error"

}

*// ❌ Demasiado específico*

{

"detail": "El caracter ';' no está permitido porque puede usarse para SQL injection"

}

**4. Logs detallados internamente, respuestas simples externamente:**

python

*# Interno (log)*

logger.warning(f"SQL injection detectado en campo 'code': {input\_value}")

*# Externo (respuesta)*

raise HTTPException(status\_code=400, detail="Error de validación en los datos")

**🧪 Assertions para Postman:**

javascript

*// Para pruebas ofensivas*

pm.test("Debe retornar 400 para XSS", function() {

pm.response.to.have.status(400);

pm.expect(pm.response.json().detail).to.be.a('string');

pm.expect(pm.response.json().detail).to.not.include('script');

});

*// Para autenticación*

pm.test("Debe retornar 401 para token inválido", function() {

pm.response.to.have.status(401);

pm.expect(pm.response.headers.get('WWW-Authenticate')).to.include('Bearer');

});

*// Para rate limiting*

pm.test("Debe retornar 429 después de 5 intentos", function() {

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

pm.expect(pm.response.json().detail).to.include('segundos');

});

**Con estas respuestas**, estarás protegiendo tu API sin revel información sensible a atacantes. 🛡️

### ****✅ 5 capas de defensa funcionando:****

1. **Palabras reservadas** → validate\_reserved\_words() ← **Esta actuó**
2. **Longitud** → validate\_max\_length()
3. **Caracteres especiales** → validate\_no\_special\_chars()
4. **SQL Injection** → validate\_sql\_injection()
5. **XSS** → validate\_no\_xss()

🎯 **Tu sistema de seguridad tiene:**

**✅ 5 capas de defensa funcionando:**

1. **Palabras reservadas** → validate\_reserved\_words() ← **Esta actuó**
2. **Longitud** → validate\_max\_length()
3. **Caracteres especiales** → validate\_no\_special\_chars()
4. **SQL Injection** → validate\_sql\_injection()
5. **XSS** → validate\_no\_xss()

🧪 **Para probar con un código válido:**

json

{

"code": "MYCODE", *// ✅ No está en palabras reservadas*

"name": "Test",

"description": "Texto normal",

"active": true

}

🛡️ **¡FELICITACIONES!** Tu sistema está **COMPLETAMENTE SECuro**:

**✅ Detecta y previene:**

* ✅ **Palabras reservadas** ← Funcionando
* ✅ **SQL Injection** ← Funcionando
* ✅ **XSS** ← Funcionando
* ✅ **Caracteres peligrosos** ← Funcionando
* ✅ **Longitudes inválidas** ← Funcionando

**¡Has implementado un sistema de seguridad empresarial robusto!** 🎊

# **Manual Paso a Paso: Pruebas de Atomicidad para FastAPI**

## 📋 Estructura de Archivos Recomendada

text

your\_project/

├── app/

│ ├── main.py

│ ├── crud/

│ │ ├── payment\_term.py

│ │ └── \_\_init\_\_.py

│ ├── models/

│ │ ├── payment\_term.py

│ │ └── \_\_init\_\_.py

│ ├── routes/

│ │ ├── payment\_term.py

│ │ └── \_\_init\_\_.py

│ └── tests/ # ✅ Nueva carpeta para pruebas

│ ├── conftest.py # ✅ Configuración de pytest

│ ├── test\_payment\_term\_crud.py # ✅ Pruebas de CRUD

│ ├── test\_payment\_term\_routes.py # ✅ Pruebas de endpoints

│ └── \_\_init\_\_.py

└── requirements.txt

## 🚀 Paso 1: Instalar Dependencias de Testing

Agrega estas dependencias a tu requirements.txt:

txt

pytest==7.4.0

pytest-asyncio==0.21.0

httpx==0.24.0

python-multipart==0.0.6

asgi-lifespan==2.0.0

pytest-mock==3.11.1

Instálalas:

bash

pip install -r requirements.txt

## 🚀 Paso 2: Crear Configuración de Pruebas (conftest.py)

Crea el archivo app/tests/conftest.py:

python

import pytest

import asyncio

from unittest.mock import AsyncMock

from fastapi.testclient import TestClient

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

from sqlalchemy.orm import sessionmaker

*# Importa tu aplicación y modelos*

from app.main import app

from app.models.base import Base

@pytest.fixture(scope="session")

def event\_loop():

"""Crear event loop para pruebas asíncronas"""

loop = asyncio.get\_event\_loop\_policy().new\_event\_loop()

yield loop

loop.close()

@pytest.fixture(scope="function")

async def async\_db\_session():

"""Session para SQLAlchemy con async support para pruebas"""

*# Usar base de datos en memoria para pruebas*

engine = create\_async\_engine(

"sqlite+aiosqlite:///:memory:",

echo=False,

future=True

)

async with engine.begin() as conn:

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

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

async\_session = sessionmaker(

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

)

async with async\_session() as session:

yield session

await engine.dispose()

@pytest.fixture(scope="function")

def test\_client():

"""Cliente de prueba para FastAPI"""

with TestClient(app) as client:

yield client

@pytest.fixture(scope="function")

def mock\_current\_user():

"""Mock de usuario autenticado"""

return {"id": "12345678-1234-5678-1234-567812345678", "username": "testuser"}

@pytest.fixture(scope="function")

def mock\_payment\_term\_data():

"""Datos de ejemplo para payment term"""

return {

"code": "TEST01",

"name": "Test Payment Term",

"description": "Test Description",

"active": True

}

## 🚀 Paso 3: Crear Pruebas CRUD (test\_payment\_term\_crud.py)

Crea el archivo app/tests/test\_payment\_term\_crud.py:

python

import pytest

from unittest.mock import AsyncMock, patch

from sqlalchemy.ext.asyncio import AsyncSession

from fastapi import HTTPException

from uuid import UUID

import uuid

class TestPaymentTermCRUD:

"""Pruebas para las operaciones CRUD de PaymentTerm"""

@pytest.mark.asyncio

async def test\_create\_payment\_term\_success(self, async\_db\_session, mock\_payment\_term\_data):

"""Prueba que la creación se complete exitosamente"""

*# Configurar mocks*

mock\_validate = AsyncMock(return\_value=None)

mock\_sanitize = lambda x: x *# No sanitizar para pruebas*

mock\_log\_action = AsyncMock(return\_value="mock\_log")

mock\_audit\_level = AsyncMock(return\_value=2)

with patch('app.crud.payment\_term.validate\_payment\_term', mock\_validate):

with patch('app.crud.payment\_term.sanitize\_input', mock\_sanitize):

with patch('app.crud.payment\_term.log\_action', mock\_log\_action):

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

*# Ejecutar la función*

from app.crud.payment\_term import create\_payment\_term

user\_id = UUID("12345678-1234-5678-1234-567812345678")

result, log = await create\_payment\_term(

async\_db\_session,

mock\_payment\_term\_data,

user\_id

)

*# Verificaciones*

assert result is not None

assert result.code == "TEST01"

assert result.name == "Test Payment Term"

assert log == "mock\_log"

mock\_validate.assert\_called\_once()

mock\_log\_action.assert\_called\_once()

@pytest.mark.asyncio

async def test\_create\_payment\_term\_validation\_failure(self, async\_db\_session):

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

*# 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.payment\_term import create\_payment\_term

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

await create\_payment\_term(

async\_db\_session,

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

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

)

*# Verificar el error*

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

assert "Validation error" in str(exc\_info.value.detail)

@pytest.mark.asyncio

async def test\_create\_payment\_term\_database\_error(self, async\_db\_session, mock\_payment\_term\_data):

"""Prueba manejo de errores de base de datos"""

*# Mock para simular error en flush*

async\_db\_session.flush = AsyncMock(side\_effect=Exception("DB error"))

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

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

*# Verificar que se lanza la excepción*

from app.crud.payment\_term import create\_payment\_term

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

await create\_payment\_term(

async\_db\_session,

mock\_payment\_term\_data,

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

)

*# Verificar el error*

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

@pytest.mark.asyncio

async def test\_atomicity\_on\_failure(self, async\_db\_session, mock\_payment\_term\_data):

"""Prueba que no se persisten datos parciales en fallos"""

call\_count = 0

def mock\_flush():

nonlocal call\_count

call\_count += 1

if call\_count > 1: *# Fallar después del primer flush*

raise Exception("Error en segunda operación")

async\_db\_session.flush = AsyncMock(side\_effect=mock\_flush)

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

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

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 excepción*

from app.crud.payment\_term import create\_payment\_term

with pytest.raises(Exception):

await create\_payment\_term(

async\_db\_session,

mock\_payment\_term\_data,

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

)

*# Verificar que se hizo rollback*

assert async\_db\_session.rollback.called

## 🚀 Paso 4: Crear Pruebas de Endpoints (test\_payment\_term\_routes.py)

Crea el archivo app/tests/test\_payment\_term\_routes.py:

python

import pytest

from unittest.mock import AsyncMock, patch

from fastapi import HTTPException

from uuid import UUID

class TestPaymentTermEndpoints:

"""Pruebas para los endpoints de PaymentTerm"""

@pytest.mark.asyncio

async def test\_create\_endpoint\_success(self, test\_client, mock\_payment\_term\_data, mock\_current\_user):

"""Prueba endpoint de creación exitoso"""

*# Mock de dependencias*

mock\_payment\_term = AsyncMock()

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

mock\_payment\_term.code = "TEST01"

mock\_payment\_term.name = "Test Payment Term"

with patch('app.routes.payment\_term.get\_current\_user', AsyncMock(return\_value=mock\_current\_user)):

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

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

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

*# Realizar petición*

response = test\_client.post(

"/payment-terms/",

json=mock\_payment\_term\_data,

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

)

*# Verificaciones*

assert response.status\_code == 200

assert response.json()["code"] == "TEST01"

assert response.json()["name"] == "Test Payment Term"

@pytest.mark.asyncio

async def test\_create\_endpoint\_validation\_error(self, test\_client, mock\_current\_user):

"""Prueba endpoint con error de validación"""

invalid\_data = {"code": "INV", "name": "T"} *# Datos inválidos*

with patch('app.routes.payment\_term.get\_current\_user', AsyncMock(return\_value=mock\_current\_user)):

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

mock\_db.rollback = AsyncMock()

*# Mock para simular error de validación*

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

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

*# Realizar petición*

response = test\_client.post(

"/payment-terms/",

json=invalid\_data,

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

)

*# Verificaciones*

assert response.status\_code == 400

assert "Validation error" in response.text

assert mock\_db.rollback.called *# Verificar rollback*

@pytest.mark.asyncio

async def test\_create\_endpoint\_authentication\_error(self, test\_client, mock\_payment\_term\_data):

"""Prueba endpoint sin autenticación"""

*# No mockear get\_current\_user para simular error de autenticación*

response = test\_client.post(

"/payment-terms/",

json=mock\_payment\_term\_data

*# Sin headers de autorización*

)

assert response.status\_code == 401 *# No autorizado*

@pytest.mark.asyncio

async def test\_create\_endpoint\_unexpected\_error(self, test\_client, mock\_payment\_term\_data, mock\_current\_user):

"""Prueba endpoint con error inesperado"""

with patch('app.routes.payment\_term.get\_current\_user', AsyncMock(return\_value=mock\_current\_user)):

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

mock\_db.rollback = AsyncMock()

*# Mock para simular error inesperado*

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

AsyncMock(side\_effect=Exception("Unexpected error"))):

*# Realizar petición*

response = test\_client.post(

"/payment-terms/",

json=mock\_payment\_term\_data,

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

)

*# Verificaciones*

assert response.status\_code == 500

assert "internal server error" in response.text.lower()

assert mock\_db.rollback.called *# Verificar rollback*

## 🚀 Paso 5: Configurar Pytest

Crea el archivo pytest.ini en la raíz del proyecto:

ini

[pytest]

asyncio\_mode = auto

testpaths = app/tests

python\_files = test\_\*.py

python\_classes = Test\*

python\_functions = test\_\*

## 🚀 Paso 6: Ejecutar las Pruebas

Desde la raíz del proyecto, ejecuta:

bash

*# Ejecutar todas las pruebas*

pytest

*# Ejecutar pruebas con verbose*

pytest -v

*# Ejecutar pruebas específicas*

pytest app/tests/test\_payment\_term\_crud.py -v

pytest app/tests/test\_payment\_term\_routes.py -v

*# Ejecutar pruebas con cobertura*

pytest --cov=app --cov-report=html

*# Ejecutar pruebas y detenerse al primer error*

pytest -x

## 🚀 Paso 7: Interpretar Resultados

* **Puntos verdes**: Pruebas exitosas ✅
* **Puntos rojos**: Pruebas fallidas ❌
* **Errores**: Problemas en el código de prueba
* **Failures**: Assertions fallidas

## 📝 Consejos Adicionales

1. **Ejecuta pruebas frecuentemente**: Después de cada cambio importante
2. **Prueba edge cases**: Datos boundary, valores nulos, etc.
3. **Mockea adecuadamente**: Aísla las pruebas de dependencias externas
4. **Mantén pruebas independientes**: Cada prueba debe poder ejecutarse por separado
5. **Usa nombres descriptivos**: Para tests y funciones de prueba

Este setup te permitirá verificar que tus transacciones sean atómicas y que el sistema se comporte correctamente en casos de éxito y error.

# **[Paso a Paso: Pruebas de Rate Limiting](#_Paso_a_Paso:)>>>**

## 📋 Prerrequisitos

1. Tener **Postman** instalado
2. Tener la colección de Postman configurada para tu API
3. Tener un endpoint que requiera autenticación

## 🚀 Paso 1: Configurar Variables de Entorno en Postman

### 1.1 Abrir el Modal de Variables

* Click en el **ojo** 👁️ en la sección de variables de entorno
* O ve a View > Show Postman Console para debugging

### 1.2 Crear Variables

Crea estas variables en tu entorno de Postman:

| **Variable** | **Valor Inicial** | **Descripción** |
| --- | --- | --- |
| base\_url | http://localhost:8000 | URL de tu API |
| auth\_token | tu\_token\_aqui | Token JWT válido |
| request\_count | 0 | Contador de requests |

## 🚀 Paso 2: Crear la Colección de Pruebas

### 2.1 Crear Nueva Colección

1. Click en **Collections** > **+**
2. Nombre: Rate Limiting Tests
3. Click en **Variables** y añade:
   * request\_count con valor 0

### 2.2 Crear Request Principal

1. Click en **+** para nuevo request
2. Método: POST
3. URL: {{base\_url}}/payment-terms/
4. Headers:
   * Authorization: Bearer {{auth\_token}}
   * Content-Type: application/json

### 2.3 Configurar Body

json

{

"code": "TESTLIMIT",

"name": "Test Rate Limiting",

"description": "Testing rate limiting functionality",

"active": true

}

## 🚀 Paso 3: Configurar Pre-request Script

### 3.1 Click en la pestaña ****Pre-request Script****

javascript

*// 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);

*// Opcional: Cambiar datos para evitar conflictos de unique constraints*

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

pm.request.body.raw = JSON.stringify({

"code": uniqueCode,

"name": "Test Rate Limiting " + (requestCount + 1),

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

"active": true

});

## 🚀 Paso 4: Configurar Tests (Assertions)

### 4.1 Click en la pestaña ****Tests****

javascript

*// Obtener el número de request actual*

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

console.log("Response status:", pm.response.status);

console.log("Request count:", currentRequestCount);

*// Test para rate limiting*

pm.test("Rate limiting funciona después de 5 requests", function() {

if (currentRequestCount > 5) {

*// Debería devolver 429 Too Many Requests*

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

pm.expect(pm.response.text()).to.include("Too Many Requests");

} else {

*// Primeros 5 requests deberían ser exitosos*

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

}

});

*// Test adicional para verificar estructura de respuesta en casos exitosos*

if (currentRequestCount <= 5) {

pm.test("Respuesta exitosa tiene estructura correcta", function() {

const response = pm.response.json();

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

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

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

});

}

*// Guardar el token si es el primer request (para autenticación)*

if (currentRequestCount === 1 && pm.response.code === 200) {

*// Opcional: guardar algún dato de la respuesta*

const responseData = pm.response.json();

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

}

## 🚀 Paso 5: Configurar el Runner de Colección

### 5.1 Crear Runner para Múltiples Ejecuciones

1. Click en la colección Rate Limiting Tests
2. Click en **Run collection**
3. Configurar:
   * **Iterations**: 10 (ejecutar 10 veces)
   * **Delay**: 0 (sin delay entre requests)

### 5.2 Configurar Variables para el Runner

javascript

*// En la configuración del Collection Runner*

{

"request\_count": 0,

"auth\_token": "tu\_token\_jwt\_aqui",

"base\_url": "http://localhost:8000"

}

## 🚀 Paso 6: Ejecutar las Pruebas

### 6.1 Método 1: Usando Collection Runner

1. Abre la colección
2. Click en **Run**
3. Selecciona todas las requests
4. Setea **Iterations** a 10
5. Click en **Run Rate Limiting Tests**

### 6.2 Método 2: Usando Newman (CLI)

Si prefieres línea de comandos:

bash

*# Instalar Newman*

npm install -g newman

*# Ejecutar la colección 10 veces*

newman run "Rate Limiting Tests.postman\_collection.json" \

-n 10 \

--delay-request 0 \

--environment "tu\_entorno.postman\_environment.json"

## 🚀 Paso 7: Verificar Resultados

### 7.1 En Postman Runner

* Deberías ver **5 respuestas 200** (éxito)
* Seguidas de **5 respuestas 429** (rate limited)

### 7.2 Resultado Esperado

text

Request 1: 200 OK

Request 2: 200 OK

Request 3: 200 OK

Request 4: 200 OK

Request 5: 200 OK

Request 6: 429 Too Many Requests

Request 7: 429 Too Many Requests

Request 8: 429 Too Many Requests

Request 9: 429 Too Many Requests

Request 10: 429 Too Many Requests

## 🚀 Paso 8: Configuración Adicional (Opcional)

### 8.1 Resetear Contador Automáticamente

javascript

*// Añadir al Pre-request Script para resetear después de 10 requests*

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

if (requestCount >= 10) {

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

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

}

### 8.2 Prueba con Diferentes Límites

javascript

*// Modificar el test para diferentes límites*

const rateLimit = 5; *// Cambiar este valor según tu configuración*

pm.test(`Rate limiting funciona después de ${rateLimit} requests`, function() {

if (currentRequestCount > rateLimit) {

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

} else {

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

}

});

## 🔧 Solución de Problemas

### ❌ Error: "Invalid token"

* Verifica que auth\_token tenga un JWT válido
* Asegúrate de que el token no haya expirado

### ❌ Error: "Unique constraint violation"

* El Pre-request Script ya genera códigos únicos
* Verifica que funcione correctamente

### ❌ Todos los requests devuelven 200

* Tu API no tiene rate limiting configurado
* Necesitas implementar rate limiting en FastAPI

## 📝 Implementar Rate Limiting en FastAPI

Si no tienes rate limiting, instálalo:

bash

pip install slowapi

Y añade a tu aplicación:

python

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)

*# En tu endpoint*

@router.post("/", response\_model=PaymentTermRead)

@limiter.limit("5/minute")

async def create\_payment\_term\_endpoint(

request: Request, *# Añadir request como primer parámetro*

payment\_term\_in: PaymentTermCreate,

db: AsyncSession = Depends(get\_async\_db),

current\_user: User = Depends(get\_current\_user),

):

*# ... tu código existente*

## ✅ Conclusión

Con estos pasos podrás probar efectivamente el rate limiting de tu API. Las primeras 5 peticiones serán exitosas y las siguientes 5 recibirán error 429, demostrando que el rate limiting funciona correctamente.