Sesión 3: Pruebas unitarias y pruebas de integración

Duración: 45 minutos

🎯 Objetivos de aprendizaje

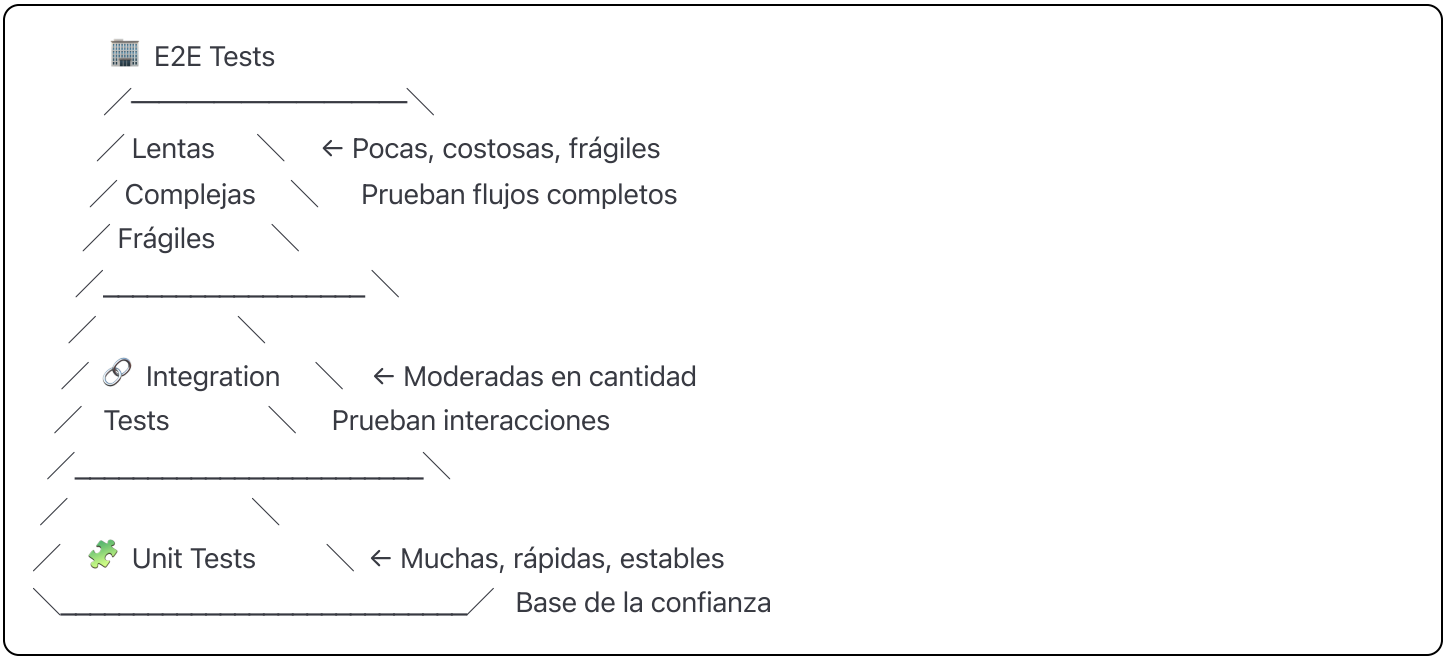
Al finalizar esta sesión, el participante será capaz de:

- Distinguir entre pruebas unitarias e integración
- Diseñar pruebas unitarias efectivas con mocking
- Implementar pruebas de integración realistas
- Aplicar la pirámide de testing en proyectos Python
- Configurar tests con diferentes niveles de aislamiento
- Usar test doubles (mocks, stubs, fakes) apropiadamente

📋 Contenido

1. La Pirámide de Testing (8 min)

Analogía: Una pirámide de testing es como la construcción de un edificio. Los cimientos (pruebas unitarias) deben ser sólidos y numerosos, las vigas (integración) conectan los componentes, y la fachada (E2E) asegura que todo funcione para el usuario final.



Distribución recomendada (Regla 70-20-10):

- **70% Unit Tests:** Rápidas, numerosas, enfoque en lógica de negocio
- **20% Integration Tests:** Moderadas, prueban componentes trabajando juntos
- **10% E2E Tests:** Pocas, prueban workflows críticos del usuario

2. Pruebas Unitarias - Definición y Características (10 min)

¿Qué es una prueba unitaria? Una prueba que verifica el comportamiento de una **unidad mínima** de código (función, método, clase) de forma **aislada**.

Características FIRST:

```
┌── F: FAST ──────────────┐  ← Ejecuta en milisegundos
├── I: INDEPENDENT ──┤  ← No depende de otras pruebas
├── R: REPEATABLE ───┤  ← Mismo resultado siempre
├── S: SELF-VALIDATING ┤  ← Pass/Fail claro
└── T: TIMELY ──────────────┘  ← Escrita junto con el código
```

Principios de aislamiento:

- **No base de datos real** → Usar mocks o in-memory DB
- **No llamadas HTTP** → Mock requests/responses
- **No sistema de archivos** → Mock file operations
- **No dependencias externas** → Inyección de dependencias

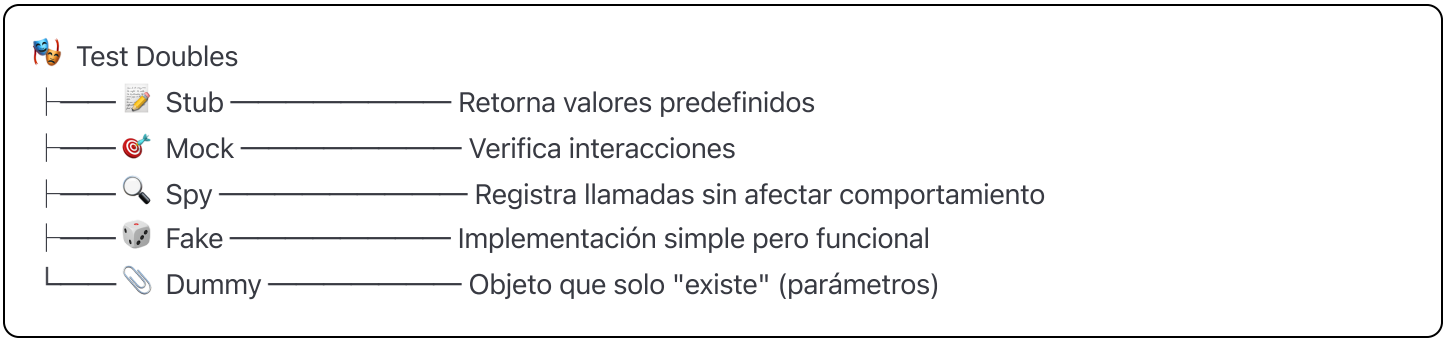
Ejemplo de dependencia vs aislamiento:

python

```python
# ❌ MAL: Dependencia externa (no unitaria)
def test_enviar_email():
    resultado = enviar_email("test@example.com", "Hola")
    assert resultado.status_code == 200  # Requiere SMTP real

# ✅ BIEN: Aislada con mock
@patch('mi_modulo.smtp_client')
def test_enviar_email(mock_smtp):
    mock_smtp.send.return_value = True
    resultado = enviar_email("test@example.com", "Hola")
    assert resultado == True
    mock_smtp.send.assert_called_once()
```

3. Test Doubles - Tipos y Uso (8 min)

Familia de Test Doubles:



Cuándo usar cada uno:

- **Stub:** Cuando necesitas datos de entrada predecibles
- **Mock:** Cuando necesitas verificar que se llamó algo específico
- **Spy:** Cuando quieres observar sin interferir
- **Fake:** Cuando necesitas comportamiento real pero simple
- **Dummy:** Cuando necesitas llenar parámetros obligatorios

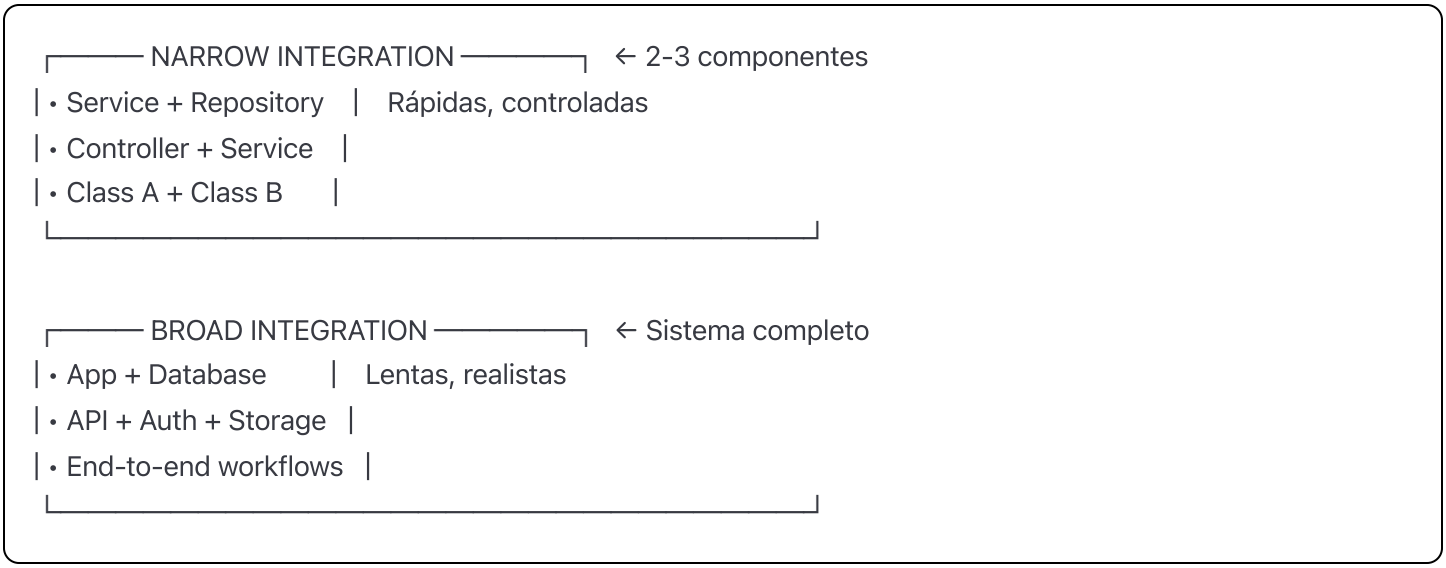
Ejemplo práctico:

```python
# Stub: Solo retorna datos
user_stub = Mock()
user_stub.get_name.return_value = "Juan"

# Mock: Verifica comportamientos
email_mock = Mock()
send_notification(email_mock, "mensaje")
email_mock.send.assert_called_with("mensaje")

# Fake: Base de datos en memoria
fake_db = {"users": {"1": {"name": "Ana"}}}
```

4. Pruebas de Integración - Definición y Alcance (10 min)

¿Qué es una prueba de integración? Prueba que verifica la **interacción correcta** entre múltiples componentes o sistemas.

Niveles de integración:



Qué testear en integración:

- **Flujo de datos** entre capas
- **Configuración** de componentes reales
- **Serialización/Deserialización**
- **Transacciones** de base de datos
- **Autenticación/Autorización** real
- **Manejo de errores** entre sistemas

Ejemplo de integración narrow:

```python
def test_usuario_service_con_repository():
    """Prueba que UserService y UserRepository trabajen juntos"""
    # Arrange: Base de datos de prueba real
    test_db = create_test_database()
    repo = UserRepository(test_db)
    service = UserService(repo)

    # Act: Flujo completo
    user_id = service.create_user("Ana", "ana@test.com")
    user = service.get_user(user_id)

    # Assert: Verificar flujo completo
    assert user.name == "Ana"
    assert user.email == "ana@test.com"
```

5. Estrategias de Testing en Python (5 min)

Patrón de organización recomendado:

```
tests/
├── unit/           ← Pruebas rápidas, aisladas
│   ├── test_models.py
│   ├── test_services.py
│   └── test_utils.py
├── integration/    ← Pruebas de componentes
│   ├── test_api.py
│   ├── test_database.py
│   └── test_workflows.py
├── e2e/            ← Pruebas completas
│   └── test_user_journey.py
└── conftest.py     ← Fixtures compartidas
```

Configuración con markers:

python

```
# pytest.ini
markers =
    unit: pruebas unitarias rápidas
    integration: pruebas de integración
    slow: pruebas que tardan >1 segundo
    database: pruebas que requieren BD
    external: pruebas con servicios externos
```

Ejecución estratégica:

bash

```
# Desarrollo: Solo unitarias (rápido)
pytest -m unit

# CI/CD: Todas menos externas
pytest -m "not external"

# Release: Suite completa
pytest
```

6. Database Testing - Patterns (4 min)

Estrategias para testing con bases de datos:

1. In-Memory Database:

```python
@pytest.fixture
def memory_db():
    engine = create_engine("sqlite:///:memory:")
    Base.metadata.create_all(engine)
    return engine
```

2. Test Database con Transactions:

```python
@pytest.fixture
def db_session():
    session = TestSession()
    yield session
    session.rollback()  # Rollback automático
    session.close()
```

3. Database Fixtures:

```python
@pytest.fixture
def user_in_db(db_session):
    user = User(name="Test", email="test@example.com")
    db_session.add(user)
    db_session.commit()
    return user
```

⚖️ Unit vs Integration: Cuándo usar cada una

Aspecto	Unit Tests	Integration Tests
Velocidad	<1ms	10ms-1s
Aislamiento	Total (mocks)	Parcial/Ninguno
Cobertura	Lógica específica	Flujos de datos
Debugging	Fácil y preciso	Más complejo
Mantenimiento	Bajo	Medio/Alto
Confianza	En componente	En integración

Reglas de decisión:

- **Unit:** Lógica de negocio, cálculos, validaciones
- **Integration:** APIs, persistencia, workflows, configuración

---

⚠️ Errores Comunes y Soluciones

❌ Errores frecuentes:

- **Test unitario que toca BD** → Convertir a integration o usar mock
- **Integration test demasiado amplio** → Dividir en componentes más pequeños
- **Mocks muy complejos** → Señal de mal diseño en el código
- **Tests que fallan aleatoriamente** → Problemas de aislamiento o timing

✅ Mejores prácticas:

- **Empezar siempre con unit tests** → Base sólida y rápida
- **Integration tests focalizados** → Probar una integración específica
- **Separar claramente los tipos** → Carpetas y markers diferentes
- **Mock en la frontera** → Mock servicios externos, no internos

---

🔧 Herramientas Específicas

Para Unit Testing:

- `unittest.mock` - Mocking nativo de Python
- `pytest-mock` - Sintaxis mejorada para mocks
- `factory_boy` - Generación de datos de prueba
- `faker` - Datos sintéticos realistas

Para Integration Testing:

- `pytest-django` - Testing específico para Django
- `pytest-flask` - Testing específico para Flask
- `testcontainers` - Contenedores para servicios reales
- `pytest-xdist` - Ejecución paralela de tests

📝 Resumen Final

1. **La pirámide de testing** guía la distribución: 70% unit, 20% integration, 10% E2E
2. **Unit tests** deben ser FAST, INDEPENDENT, REPEATABLE con aislamiento total
3. **Test doubles** (mocks, stubs, fakes) permiten aislamiento efectivo
4. **Integration tests** verifican que componentes colaboren correctamente
5. **Organización clara** en carpetas y markers facilita la ejecución selectiva
6. **Database testing** requiere estrategias específicas (in-memory, transactions)
7. **Equilibrio** entre velocidad (unit) y confianza (integration)

🏃 Actividad Práctica Sugerida

Ejercicio: Toma un módulo existente con dependencias externas (ej: servicio que consulta API y guarda en BD) y crea:

1. **3-5 unit tests** usando mocks para aislar dependencias
2. **2-3 integration tests** con base de datos de prueba real
3. **Organización** en carpetas unit/ e integration/
4. **Ejecución selectiva** usando markers

Tiempo estimado: 25-30 minutos adicionales