



Módulo 4: Automatización de pruebas

El concepto de regresividad en las pruebas

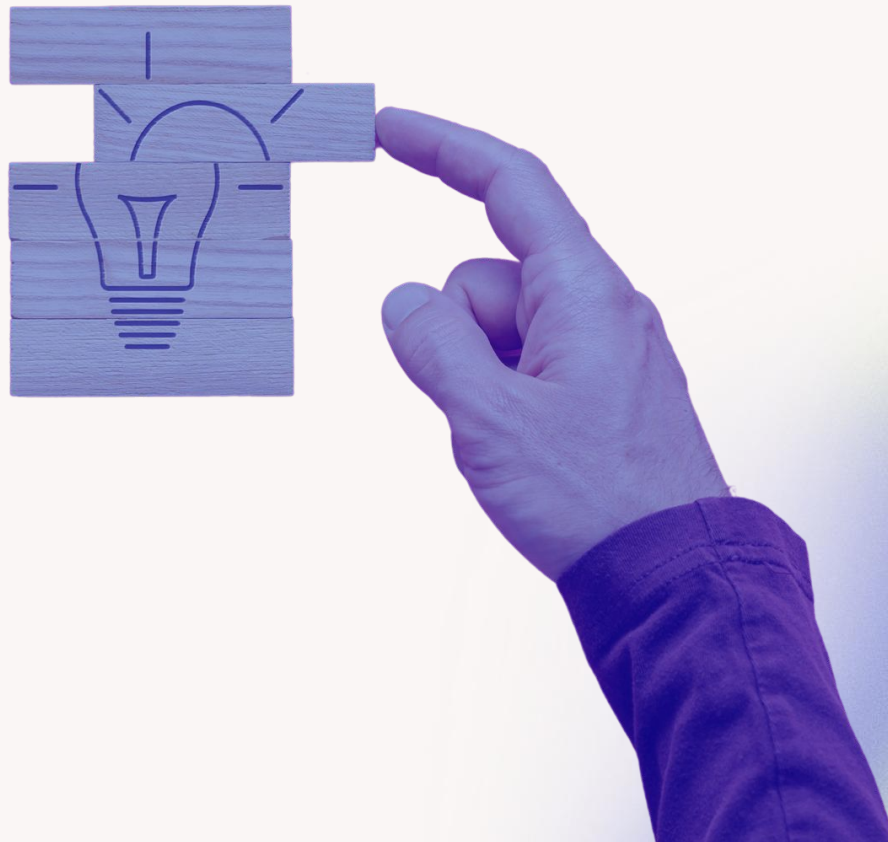


MÓDULO 4

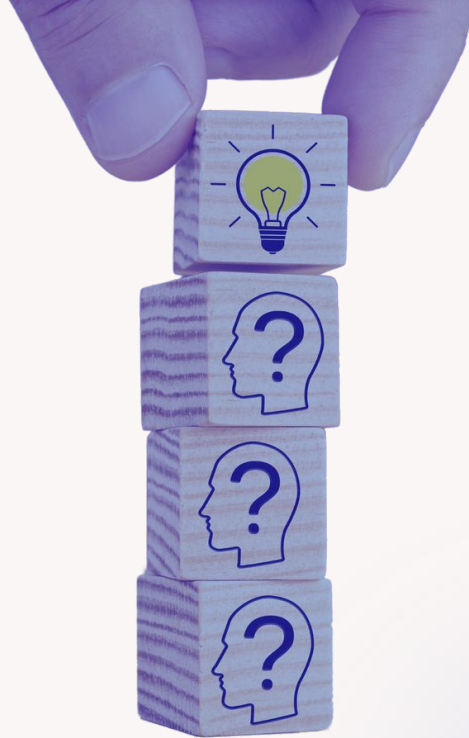
- Testing en un entorno ágil
- **El concepto de regresividad en las pruebas**
- Pruebas unitarias
- Pruebas unitarias de servicios web
- Pruebas de rendimiento
- Pruebas funcionales

Objetivos

Comprender el concepto de regresividad en las pruebas, su rol en la calidad del software y cómo integrarlo con estrategias de automatización y CI/CD en entornos ágiles.




¿Por qué es importante considerar distintos tipos de pruebas dentro de un proceso ágil de desarrollo de software?



Fundamentos y Conceptos Claves en Pruebas de Software

El concepto de regresividad en las pruebas: definición e importancia

 **Las pruebas de regresión** aseguran que nuevas funcionalidades no rompan el comportamiento ya existente del sistema. Cada vez que se modifica el código, se ejecutan estas pruebas para validar que no hay efectos colaterales.

Importancia:

- Detectan errores introducidos por cambios.
- Previenen la reaparición de bugs solucionados.
- Dan confianza para iterar rápidamente.


El concepto de regresividad en las pruebas: definición e importancia

📖 **En nuestro sistema de vuelos:** Se añade un nuevo campo opcional para *clase de asiento* (*económica, ejecutiva, primera*). Una prueba de regresión debe asegurar que las búsquedas **anteriores** sin ese campo sigan funcionando.

✏️ Ejemplo en **pytest**:

```
def test_regresion_busqueda_sin_clase():  
    resultados = buscar_vuelos("BOG", "MDE", "2025-07-01")  
    assert all("clase" in r for r in resultados)
```


Pruebas de regresión vs. pruebas nuevas: diferencias y enfoques

 Las **pruebas nuevas** validan funcionalidad reciente. Las **de regresión** revalidan lo anterior para evitar efectos colaterales.

Comparativa:

Tipo de prueba	Objetivo	Frecuencia
Nuevas	Verificar una funcionalidad nueva	Una vez por feature
Regresión	Asegurar que nada se rompió	En cada cambio o release

Pruebas de aceptación: validación de requisitos y criterios de éxito

 Validan que el software **cumple con lo acordado con el cliente o el usuario final**. Derivan directamente de los criterios de aceptación definidos por el Product Owner.

Características:

- Generalmente automatizadas.
- Enfocadas en la experiencia de usuario.
- Validan historias de usuario completas.

Pruebas de aceptación: validación de requisitos y criterios de éxito


Criterio de aceptación ejemplo:

“Como usuario, quiero ver el precio total del vuelo al buscarlo”.

Automatización con **pytest**:

```
def test_precio_total_visible():  
    vuelos = buscar_vuelos("BOG", "MDE", "2025-07-01")  
    for vuelo in vuelos:  
        assert "precio_total" in vuelo
```

Pruebas de humo: detección rápida de fallos críticos

 Se ejecutan para validar si el sistema **básicamente funciona**, antes de pasar a pruebas más profundas. También se llaman pruebas de sanidad.

✅ Características:


- Se ejecutan tras un nuevo despliegue.
- Detectan bloqueos evidentes.
- Son rápidas y automatizadas.

Ejemplo con **requests**:

```
import requests

def test_smoke_api_respuesta():
    response =
requests.get("http://localhost:5000/api/buscar?origen=BOG&destino=MDE&fecha=2025-07-01")
assert response.status_code == 200
```

Otros tipos de pruebas en entornos de desarrollo y producción

 En un flujo ágil y DevOps, se combinan distintos tipos de pruebas para cubrir todo el ciclo.

Tipos adicionales:


Pruebas exploratorias: manuales, libres.

Pruebas A/B: producción, dos versiones a comparar.

Pruebas en staging: entorno casi idéntico a producción.

Pruebas de recuperación: resiliencia ante fallos.

Beneficios de implementar estrategias de prueba en proyectos ágiles

 Una estrategia de pruebas bien implementada genera velocidad sin sacrificar calidad.


Beneficios concretos:

- Feedback temprano y continuo.
- Reducción de bugs en producción.
- Entregas más confiables.
- Mejora la confianza del equipo en el producto.

En nuestro caso práctico:

Gracias a una suite de regresión estable, el equipo agrega nuevas rutas y filtros de vuelo sin temor a romper funcionalidades previas.

Desafíos comunes en la gestión de pruebas de software

 No todo es fácil: las pruebas pueden volverse difíciles de mantener, lentas o irrelevantes si no se gestionan bien.

Desafíos típicos:

- Pruebas rotas que no se actualizan.
- Alta dependencia entre pruebas.
- Demoras por suites pesadas.
- Duplicación innecesaria.

Soluciones:

- Refactorizar suites regularmente.
- Revisar y eliminar pruebas obsoletas.
- Usar etiquetas para filtrar pruebas por contexto (`@smoke`, `@regression`).

Importancia de la cobertura de pruebas en la calidad del software

📌 La cobertura mide qué porcentaje del código ha sido ejecutado por las pruebas. Aunque no lo es todo, es una métrica clave para detectar áreas sin validar.

✓ Tipos de cobertura:

- Cobertura de líneas.
- Cobertura de funciones.
- Cobertura de ramas (condicionales).


📖 Herramientas:

- Python: **coverage**
- Java: **JaCoCo**
- JS: **nyc**, **jest**

🔧 Ejemplo de cobertura con **pytest**:

```
pytest --cov=mi_app tests/
```

Estrategias para mejorar la eficiencia en la ejecución de pruebas

 Cuando el proyecto crece, se necesitan estrategias para no afectar la velocidad del desarrollo.


✓ Estrategias efectivas:

- **Paralelizar** ejecución de pruebas (GitHub Actions, Jenkins).
- **Dividir por tipo** (unitarias → rápido, integración → más espaciadas).
- **Ejecutar solo pruebas afectadas** por los cambios (test impact analysis).
- **Mockear servicios externos** para ganar velocidad.

En nuestro caso:

Las pruebas de conexión a la API de vuelos reales se ejecutan solo en la noche. Durante el día, se mockean para acelerar la validación.

Casos de estudio sobre implementación efectiva de pruebas

 Aprendamos de buenas prácticas reales:

Caso 1: equipo de e-commerce

- Implementó **TDD** para carrito de compras.
- Redujo bugs en producción en un 40%.
- Usó Cypress para pruebas E2E con cada PR.

Caso 2: startup fintech

- Usó **pytest** con 85% de cobertura.
- Automatizó validaciones de reglas fiscales.
- Desplegaban 3 veces al día con pipelines y suites rápidas.

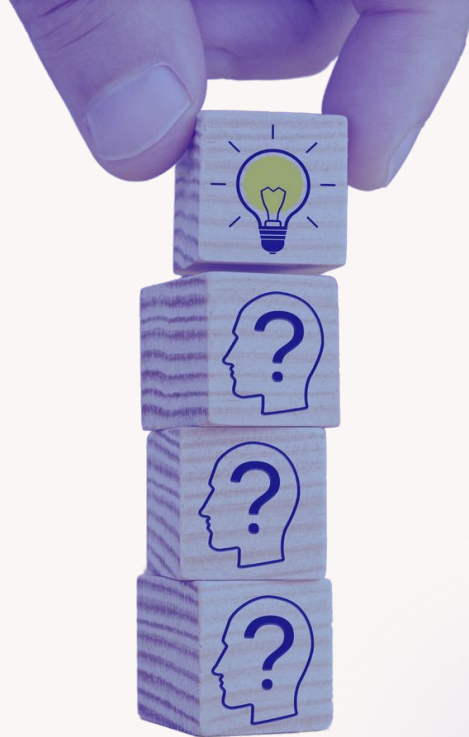
Casos de estudio sobre implementación efectiva de pruebas



Caso 3: nuestro sistema de vuelos


- Automatiza regresión + humo en cada push.
- Ejecuta pruebas funcionales completas cada noche.
- Refactoriza pruebas viejas cada 3 sprints.

¿Cómo influye la correcta implementación de pruebas de regresión, aceptación y humo en la calidad y estabilidad del software?



Automatización de Pruebas y su Integración en CI/CD

Introducción a la automatización de pruebas: objetivos y beneficios

 Automatizar pruebas significa reemplazar la ejecución manual repetitiva por scripts que se ejecutan automáticamente como parte del ciclo de desarrollo.


Objetivos principales:

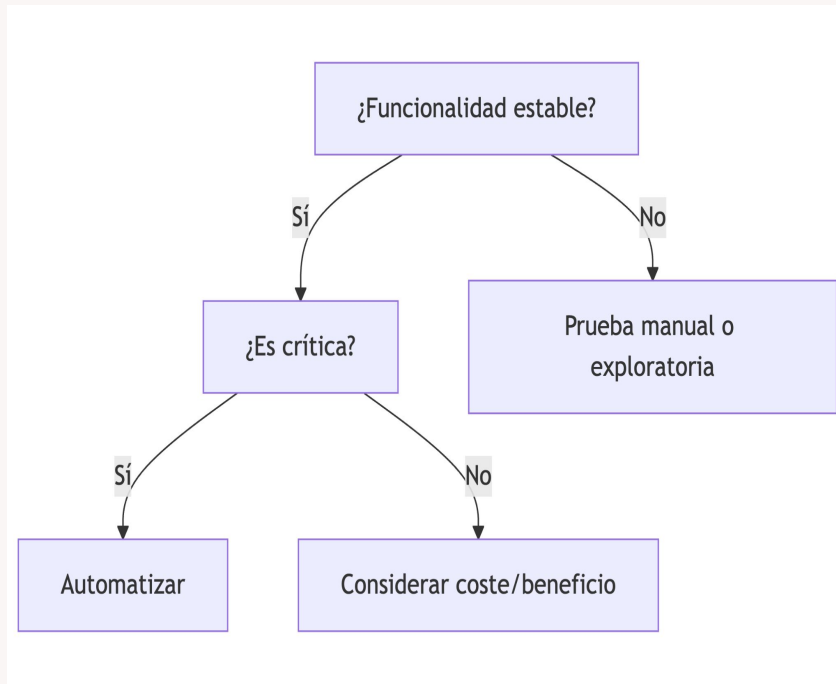
- Acelerar la retroalimentación.
- Asegurar calidad constante.
- Disminuir errores humanos.
- Reutilizar pruebas en múltiples entornos.

Beneficios:

- Reducción de tiempos de validación.
- Mayor cobertura de pruebas.
- Costos reducidos a largo plazo.

Cuándo automatizar y cuándo optar por pruebas manuales

 No todo debe automatizarse. Hay casos donde las pruebas manuales siguen siendo necesarias.



Cuándo automatizar y cuándo optar por pruebas manuales



Automatizar cuando:

- La funcionalidad es estable y repetitiva.
- Es crítica para el negocio (e.g. pago, login).
- Puede causar regresiones frecuentes.
- Se requiere feedback constante (CI/CD).



Mantener pruebas manuales cuando:

- Hay cambios de UI frecuentes.
- Se explora el sistema en fases iniciales.
- Se validan emociones, diseño o accesibilidad.


Principales herramientas para la automatización de pruebas

Tipo	Herramientas comunes
Unitarias	pytest, JUnit, TestNG, RSpec
UI	Selenium, Cypress, Playwright
E2E (extremo a extremo)	Robot Framework, Cucumber
Rendimiento	Locust, JMeter, k6
CI/CD	GitHub Actions, Jenkins, GitLab CI

Comparación de herramientas populares

Herramienta	Lenguaje	Ideal para	Pros	Contras
Selenium	Multilenguaje	UI web complejas	Madurez, comunidad amplia	Verboso, lento
JUnit	Java	Pruebas unitarias	Integración con Maven	Solo para Java
Cypress	JavaScript	UI moderna y ágil	Rápido, fácil de configurar	Limitado fuera de navegador
TestNG	Java	Integración avanzada	Flexible, anotaciones	Complejidad creciente

Implementación de pruebas automatizadas en entornos de desarrollo

 En Agile, la automatización se integra desde el día 1. Las historias de usuario vienen acompañadas de criterios automatizables.

Historia de usuario:

- Como usuario, quiero recibir resultados de búsqueda con precio total, impuestos incluidos.

Prueba automatizada en **pytest**:

```
def test_precio_total_incluye_impuestos():  
    vuelos = buscar_vuelos("BOG", "MDE", "2025-07-01")  
  
    for vuelo in vuelos:  
        assert vuelo["precio_total"] >= vuelo["precio_base"]
```

Desafíos y mejores prácticas en la automatización de pruebas

 La automatización trae retos técnicos y organizacionales.


Desafíos comunes:

- Flaky tests (fallan aleatoriamente).
- Mantenimiento costoso.
- Dependencias externas inestables.
- Bajo retorno en pruebas mal priorizadas.

Buenas prácticas:

- Establecer una pirámide de pruebas (unitarias > integración > UI).
- Revisión frecuente de pruebas.
- Uso de mocks y fixtures para aislar lógica.
- Reportes claros y legibles para fallos.

Integración de la automatización de pruebas en pipelines de CI/CD

 El mayor valor de la automatización se logra cuando las pruebas se integran en el pipeline y se ejecutan **con cada cambio**.


Ejemplo de GitHub Actions + pytest:

```
name: Test API de vuelos

on: [push, pull_request]

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - name: Instalar dependencias
        run: pip install -r requirements.txt
      - name: Ejecutar pruebas
        run: pytest --maxfail=2 --disable-warnings
```

Optimización de tiempos de ejecución en pruebas automatizadas

 A medida que crecen los proyectos, optimizar el tiempo de ejecución se vuelve crucial.

✓ Técnicas de optimización:

- **Ejecutar solo pruebas afectadas** por cambios recientes.
- **Etiquetar pruebas** (`@smoke`, `@regression`, etc.).
- **Paralelizar pruebas** (CI runners).
- **Usar caché** de dependencias para no reinstalar todo.


Ejemplo con **pytest** y marcas:

```
import pytest

@pytest.mark.smoke
def test_api_respuesta_200():
    ...
```

En el pipeline:

Reportes y análisis de resultados en la automatización de pruebas

 Los reportes deben ser comprensibles por el equipo, no solo por QA.


Tipos de reportes:

- HTML con estadísticas detalladas.
- Reportes en CI (Jenkins, GitHub Actions).
- Integración con dashboards (Allure, TestRail, ReportPortal).

Ejemplo con **pytest** y plugin **pytest-html**:

```
pytest --html=report.html
```

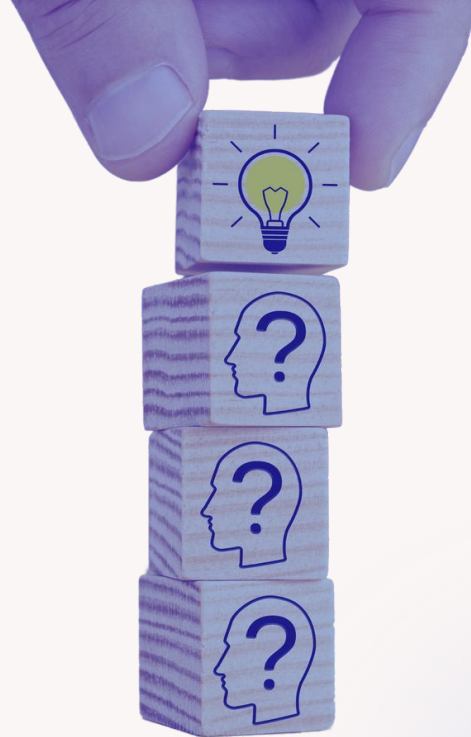
Tendencias y futuro de la automatización en la industria del software

 La automatización está evolucionando rápidamente gracias a nuevas tecnologías.

Tendencias emergentes:

- Pruebas impulsadas por IA (ej: test case generation).
- Automatización visual sin código.
- Autohealing en pruebas UI (adaptación a cambios).
- Automatización en producción (observabilidad + testing).
- Shift Left + Shift Right: pruebas antes y después del release.

¿Qué criterios deben considerarse al decidir qué pruebas automatizar en un entorno ágil con integración continua?





Ejercicio Guiado: Pruebas de Regresión Automatizadas en una API de Cupones



Ejercicio Guiado: Pruebas de Regresión Automatizadas en una API de Cupones

En este ejercicio vas a construir una API básica en Python para aplicar **cupones de descuento**. A medida que avances, vas a simular un cambio en el código que rompe una funcionalidad existente: una **regresión**. El objetivo es aprender a:

- Detectar regresiones con pruebas automatizadas.
- Automatizar esas pruebas con CI usando GitHub Actions.
- Mejorar la calidad del software en un entorno ágil.

Objetivos

- Escribir una API simple con Flask.
- Implementar pruebas unitarias y de regresión.
- Simular un cambio regresivo.
- Automatizar pruebas en GitHub Actions.





Paso 1: Crear la API (Node.js + Express)



¿Qué vas a hacer?

Configurarás el entorno de Python y crearás las carpetas básicas del proyecto.



Por qué es importante:

Tener una estructura organizada te permitirá separar la lógica, las pruebas y los scripts del proyecto.



Instrucciones

- Abre tu terminal o consola.
- Ejecuta estos comandos:

```
mkdir cupones-api
cd cupones-api
python -m venv venv
source venv/bin/activate      # En Linux/Mac
# venv\Scripts\activate.bat   # En Windows
pip install flask pytest
pip freeze > requirements.txt
```

Ahora crea estas carpetas y archivos (puedes hacerlo manualmente o con comandos):

```
cupones-api/
├── app/
│   ├── __init__.py
│   ├── cupones.py
│   └── api.py
├── tests/
│   ├── test_cupones.py
│   └── test_api.py
└── requirements.txt
```



Paso 2 – Implementar la lógica de negocio



¿Qué vas a hacer?

Crearás la lógica para aplicar cupones de descuento y calcular el precio final con impuestos.



Por qué es importante:

Esta será la funcionalidad central que protegerás con pruebas para evitar regresiones.



Instrucciones

Abre el archivo `app/cupones.py` escribe:

```
def aplicar_cupon(precio, cupon):
    descuentos = {
        "OFERTA10": 0.10,
        "SUPER20": 0.20,
        "BIENVENIDA": 0.15
    }
    if cupon in descuentos:
        return round(precio * (1 - descuentos[cupon]), 2)
    return precio
def calcularPrecioFinal(precio_base, cupon, impuesto=0.19):
    precio_desc = aplicar_cupon(precio_base, cupon)
    return round(precio_desc * (1 + impuesto), 2)
```



Paso 3 – Crear la API con Flask



¿Qué vas a hacer?

Vas a exponer esa lógica como una API REST con Flask para que se pueda consumir desde otro sistema.



Instrucciones

Abre `app/api.py` agrega:

```
from flask import Flask, request, jsonify
from app.cupones import calcular_precio_final

app = Flask(__name__)

@app.route('/precio', methods=['POST'])
def calcular():
    data = request.get_json()
    precio = data.get("precio")
    cupon = data.get("cupon")
    impuesto = data.get("impuesto", 0.19)

    final = calcular_precio_final(precio, cupon, impuesto)
    return jsonify({"precio_final": final})
```



Paso 4 – Escribir pruebas unitarias



¿Qué vas a hacer?

Probarás que los cupones se aplican correctamente desde la lógica del backend.



Por qué es importante:

Estas pruebas te dirán si la lógica se rompe más adelante (regresión).



Instrucciones

Abre `tests/test_cupones.py` y escribe:



Ejecuta las pruebas:

```
pytest
```



Verifica que **todas pasen en verde**.

```
from app.cupones import aplicar_cupon, calcular_precio_final

def test_descuento_oferta10():
    assert aplicar_cupon(100, "OFERTA10") == 90.0
def test_descuento_super20():
    assert aplicar_cupon(200, "SUPER20") == 160.0
def test_descuento_bienvenida():
    assert aplicar_cupon(100, "BIENVENIDA") == 85.0
def test_precio_final_con_impuesto():
    assert calcular_precio_final(100, "OFERTA10") == 107.1
```



Paso 5 – Simular una regresión



¿Qué vas a hacer?

Vas a modificar el código de manera que se elimine un cupón sin darte cuenta (regresión silenciosa).



Instrucciones

Edita `app/cupones.py` y elimina uno de los cupones:

```
def aplicar_cupon(precio, cupon):
    descuentos = {
        "OFERTA10": 0.10,
        "SUPER20": 0.20
        # El cupon "BIENVENIDA" se eliminó sin querer
    }
    if cupon in descuentos:
        return round(precio * (1 - descuentos[cupon]), 2)
    return precio
```



Ejecuta las pruebas de nuevo:

pytest



Debería fallar la prueba

`test_descuento_bienvenida`.



¡Detectaste una regresión!

Paso 6 – Corregir el error

¿Qué vas a hacer?

Restaurarás el código para que vuelva a funcionar el cupón perdido.

Instrucciones

Vuelve a poner el cupón en cupones.py:

```
"BIENVENIDA": 0.15,
```

Vuelve a ejecutar:

```
pytest
```

 ¡Todo debería pasar!



Paso 7 – Automatizar con GitHub Actions



¿Qué vas a hacer?

Crearás un archivo YAML para que las pruebas se ejecuten automáticamente en GitHub.



Instrucciones

1. Crea la carpeta `.github/workflows/`
2. Dentro, crea el archivo `test-regresion.yml` con el siguiente contenido:

```
name: Pruebas de Regresión - Cupones
```

```
on:
```

```
  push:
```

```
    branches: [main]
```

```
  pull_request:
```

```
    branches: [main]
```

```
jobs:
```

```
  test:
```

```
    runs-on: ubuntu-latest
```

```
  steps:
```

```
    - uses: actions/checkout@v3
```

```
    - name: Configurar Python
```

```
      uses: actions/setup-python@v4
```

```
      with:
```

```
        python-version: '3.10'
```

```
- name: Instalar dependencias
```

```
  run: |
```

```
    python -m venv venv
```

```
    source venv/bin/activate
```

```
    pip install -r requirements.txt
```

```
- name: Ejecutar pruebas
```

```
  run: |
```

```
    source venv/bin/activate
```

```
    pytest
```



Paso 7 – Automatizar con GitHub Actions

1. Sube tu proyecto a un repositorio en GitHub y haz push.
2. Ve a la pestaña **Actions** y observa que el workflow se ejecuta.

Preguntas finales

1. ¿Por qué fue difícil detectar la regresión sin pruebas?
2. ¿Cómo te ayuda el testing automatizado a mantener calidad?
3. ¿Qué otras partes del código deberías cubrir con pruebas?
4. ¿Cómo evitarías errores similares en el futuro?

Entregable:

Cada equipo debe entregar:

- El código funcional del proyecto.
- Una captura de pantalla del test fallando (regresión) y otra en verde (corregido).
- Captura del workflow corriendo en GitHub Actions.
- Un resumen (2-3 minutos) de cómo detectaron la regresión y cómo la evitarán en el futuro.



Resumen de lo aprendido

- **Pruebas de Regresión:** Evalúan que los cambios recientes no afecten funcionalidades previas.
- **Estrategias de Pruebas:** Inclusión de pruebas de aceptación, humo y regresión para validar requisitos, detectar fallos críticos y mantener calidad.
- **Automatización en CI/CD:** Uso de herramientas como Selenium, JUnit, Cypress y TestNG, integradas en pipelines para mejorar eficiencia y cobertura.
- **Mejores Prácticas y Tendencias:** Automatizar pruebas repetitivas, analizar reportes, optimizar tiempos de ejecución y adaptarse a nuevas tecnologías del testing ágil.

Próxima clase...

Pruebas unitarias

