

«Talento Tech»

Automation Testing

Clase 13



Clase N° 13: Reportes y Logging

Temario

- ¿Por qué necesitamos reportes y logs?
- Generación de reportes HTML con pytest-html
- Capturas de pantalla automáticas para fallos UI
- Implementación de logging con el módulo logging
- Estrategias de gestión de errores (soft-asserts, retries, teardown)
- Ejercicio práctico TalentoLab – habilitar reportes y logs para el framework final

Objetivos de la clase

En esta clase aprenderemos a configurar **pytest-html** para generar reportes claros y visuales que faciliten el análisis de resultados. También integraremos **capturas de pantalla** automáticas para detectar fácilmente fallas en pruebas de interfaz, y sumaremos un **logger centralizado** que registre cada paso del proceso en archivos rotativos. Todo esto nos permitirá sentar las bases de **observabilidad** que deberá estar presente en la entrega final para comunicar con claridad los hallazgos a los stakeholders del proyecto.

¿Por qué necesitamos reportes y logs?

Antes de comenzar con las herramientas concretas, es importante entender **por qué estamos sumando reportes y logs a nuestro framework de testing**. A medida que las suites de pruebas crecen, también lo hace la complejidad para interpretarlas. Ya no alcanza con mirar la consola para saber qué falló o cuánto demoró cada test. Necesitamos una forma clara y estructurada de **comunicar los resultados**, tanto para identificar problemas técnicos como para presentar evidencia entendible para stakeholders no técnicos. Reportes y logs nos dan esa doble mirada: por un lado, una visión general del estado de las pruebas, y por otro, un registro detallado que nos ayuda a depurar, documentar y mejorar. Esa combinación es lo que vamos a empezar a construir hoy.



Veamos un ejemplo concreto para entender mejor este punto. Imaginá que ejecutás 50 pruebas y solo tenés acceso al resultado en consola.

None

test_login.py::test_valid_user PASSED

test_login.py::test_invalid_user FAILED

test_inventory.py::test_add_product PASSED

test_checkout.py::test_payment_flow FAILED

... (46 tests más)

A simple vista, sabés qué tests pasaron y cuáles fallaron, pero... ¿sabés cuánto tardaron?, ¿cuánto tardó cada test? ¿por qué falló exactamente? ¿qué estaba viendo el usuario cuando ocurrió el error? ¿qué estaba haciendo el sistema en el momento del error?, ¿qué vio el usuario? Esa falta de contexto es lo que vamos a abordar a continuación.

Como verás es muy difícil cuando la aplicación crece ir revisando cada test uno a uno.

✓ La solución: reportes + logs trabajando juntos

Los reportes HTML responden "¿qué pasó?"

- ¿Qué tests corrieron? → Vista general del estado de la suite
- Su estado y duración → Identificar tests lentos o problemáticos
- Evidencia visual → Screenshots, logs, traceback en un solo lugar

Los logs responden "¿qué ocurrió exactamente?"

- Registro cronológico paso a paso
- Facilitan la depuración post-mortem sin reproducir el escenario
- Contexto detallado para entender el flujo de ejecución

Metodología recomendada

None

1. Reportes primero → Vista rápida del estado general

2. Logs después → Análisis detallado de fallos específicos

¿Por qué en ese orden? El reporte te dice dónde mirar, los logs te dicen cómo solucionarlo.

Como vimos, los reportes y logs no solo mejoran nuestra capacidad de análisis, sino que también nos permiten **comunicar de forma clara y profesional** los resultados de nuestras pruebas. Los reportes responden rápidamente a la pregunta "¿qué pasó?", mientras que los logs nos ayudan a entender "¿qué ocurrió exactamente y en qué orden?". Esta combinación nos ahorra tiempo, reduce errores y aporta transparencia. Ahora que entendemos su importancia, vamos a ver **cómo generar nuestros primeros reportes HTML con pytest-html**.

Pytest-html



Ahora que entendemos por qué los reportes son fundamentales, vamos a aprender a generarlos de forma automática usando una herramienta llamada pytest-html. Esta es una extensión de pytest, nuestro framework de testing, que permite producir reportes visuales en formato HTML con solo ejecutar un comando.

Con pytest-html, cada vez que corramos nuestra suite de pruebas, se va a generar un archivo HTML que resume el estado de todos los tests: cuáles pasaron, cuáles fallaron, cuánto tardaron y, si lo configuramos bien, hasta puede incluir capturas de pantalla y mensajes de error detallados. Esto es clave tanto para el debugging como para compartir resultados con otras personas del equipo o con stakeholders.

Vamos a comenzar instalando pytest-html y generando nuestro primer reporte básico.

Instalación

Ejecuta el comando una sola vez en tu terminal (o virtualenv) para agregar la dependencia a tu proyecto:

```
pip install pytest-html
```

Primer reporte paso a paso

1. **Abre la terminal** en la raíz del repositorio (donde está tu `pytest.ini`)


Ejecuta:

```
pytest -v --html=reports/ui_api_report.html  
--self-contained-html
```

Explicación del comando que ejecutamos:

- `-v` muestra salida verbose (cada nombre de test)
- `--html=...` indica la ruta y nombre del reporte que se generará
- `--self-contained-html` incrusta todo (CSS, JS, imágenes) en un único archivo <1 MB

2. Luego de ejecutar ese comando andá a la carpeta `reports/` y abre `ui_api_report.html` en tu navegador: verás una tabla con estado, duración y detalles de cada test

 **Consejo:** agrega `reports/` al `.gitignore`; el reporte debe generarse fresco en cada pipeline, no versionarse.

Personalizar título y metadatos con pytest.ini


¿Qué es pytest.ini y por qué es importante?


El archivo `pytest.ini` es el **centro de control** de tu proyecto de testing. Piénsalo como el "panel de configuración" donde defines:


- ¿Cómo se ejecutan tus tests por defecto?
- ¿Qué información aparece en los reportes?
- ¿Qué markers están disponibles?
- Opciones que no quieres escribir cada vez

Configuración paso a paso

Si **ya tienes** un archivo `pytest.ini` en tu proyecto, ábrelo y agrega las nuevas líneas. Si **no existe**, créalo en la raíz del proyecto (al mismo nivel que tus carpetas de tests).

```
None
[pytest]
#  Opciones que se ejecutan automáticamente
addopts = -v --html=reports/report.html --self-contained-html

#  Personalización del reporte HTML
html_report_title = TalentoLab - Resumen de ejecución
html_report_description = Suite UI + API completa

#  Tus markers existentes (si los tienes)
markers =
    smoke: pruebas críticas y rápidas
    api: pruebas de API
    ui: pruebas de interfaz
```

Desglose detallado de cada configuración

Configuración	¿Qué hace?	Ejemplo visual
<code>addopts = -v</code>	Modo verbose: muestra cada test individualmente	✅ test_login_valido PASSED en lugar de solo un punto
<code>--html=reports/report.html</code>	Genera el reporte HTML en esa ruta específica	📄 Crea el archivo reports/report.html
<code>--self-contained-html</code>	Incluye CSS, JS e imágenes en un solo archivo	📦 Un HTML de ~800KB vs múltiples archivos
<code>html_report_title</code>	Título principal del reporte	📋 "TalentoLab – Resumen de ejecución"
<code>html_report_description</code>	Subtítulo descriptivo	📝 "Suite UI + API completa"

El antes y después

❌ Antes (sin pytest.ini)

Shell

Comando largo y propenso a errores

```
pytest -v --html=reports/report.html --self-contained-html tests/
```

Si olvidas alguna opción, el reporte será inconsistente

```
pytest tests/ # ¡Ups! No se genera reporte
```

✅ Después (con pytest.ini)

Shell

Comando simple y consistente

```
pytest
```


O si quieres ejecutar solo tests específicos

```
pytest tests/ui/
```

```
pytest -k "login"
```


Beneficios clave


1. **Consistencia:** Todos en el equipo generan reportes idénticos
2. **Simplicidad:** No necesitas recordar comandos largos
3. **Mantenibilidad:** Si cambias la configuración, se aplica automáticamente
4. **CI/CD amigable:** Los pipelines usan el mismo comando simple


 **Importante:** No borres configuraciones existentes

Si ya tienes un `pytest.ini` con contenido como este:

```
None
[pytest]
markers =
    integration: pruebas de integración
    slow: pruebas que tardan más de 10 segundos
testpaths = tests
python_files = test_*.py
```

NO lo reemplaces. Simplemente agrega las nuevas líneas:

```
None
[pytest]
#  Configuraciones existentes (manténlas)
markers =
    integration: pruebas de integración
    slow: pruebas que tardan más de 10 segundos
testpaths = tests
python_files = test_*.py

#  Nuevas configuraciones (agrégalas)
addopts = -v --html=reports/report.html --self-contained-html
html_report_title = TalentoLab - Resumen de ejecución
html_report_description = Suite UI + API completa
```

Verificación rápida

Para comprobar que tu configuración funciona:

1. **Ejecuta:** `pytest --collect-only`
2. **Verifica:** Debe mostrar tus tests sin ejecutarlos
3. **Ejecuta:** `pytest` (sin parámetros)
4. **Comprueba:** Debe generar `reports/report.html`

Tip profesional

Puedes tener diferentes configuraciones para diferentes entornos:

```
None
[pytest]
# Configuración base
addopts = -v --html=reports/report.html --self-contained-html

# Para desarrollo local
# addopts = -v --html=reports/dev_report.html --self-contained-html -x

# Para CI/CD (se puede sobrescribir desde el pipeline)
# addopts = -v --html=reports/ci_report.html --self-contained-html
--tb=short
```

Con esta configuración, tu equipo tendrá reportes profesionales y consistentes sin esfuerzo adicional.

Enriquecer la tabla de resultados desde conftest.py

¿Por qué personalizar la tabla de resultados?

El reporte HTML básico te muestra **qué** test falló, pero no te dice **dónde** estaba el usuario cuando ocurrió el error. Para tests UI, saber la URL exacta puede ser la diferencia entre 5 minutos y 2 horas de debugging.

Tabla estándar vs personalizada

Reporte estándar	Reporte personalizado
test_checkout_flow FAILED	test_checkout_flow FAILED en /cart.html
test_login_invalid FAILED	test_login_invalid FAILED en /login

Implementación paso a paso

1. Ubicación del código

Abre tu archivo `conftest.py` y agrega el siguiente código **después de tus imports existentes**:

python

```
Python
# conftest.py (fragmento - NO reemplaces tu código anterior)

def pytest_html_results_table_header(cells):
    """Añade una columna 'URL' justo después de 'Test ID'."""
    cells.insert(2, 'URL')

def pytest_html_results_table_row(report, cells):
    """Rellena la columna con la URL almacenada en el atributo
    'page_url'."""
    cells.insert(2, getattr(report, 'page_url', '-'))
```


2. ¿Cómo funciona?

Hook	¿Cuándo se ejecuta?	¿Qué hace?
<code>pytest_html_results_table_header</code>	Una vez al generar el reporte	Crea la columna "URL" en la cabecera
<code>pytest_html_results_table_row</code>	Por cada test ejecutado	Rellena la celda con la URL del test

3. Usar la información en tus tests

Para que la URL aparezca en el reporte, tus tests deben guardar esta información:

Python

```
def test_login_valido(driver, request):  
    # Tu código de test...  
    login_page.abrir()  
  
    #  Guarda la URL actual para el reporte  
    request.node.page_url = driver.current_url  
  
    # Resto del test...
```

Otras columnas útiles que puedes agregar

Python

```
# Ejemplos de personalización adicional  
def pytest_html_results_table_header(cells):  
    cells.insert(2, 'URL')  
    cells.insert(3, 'Browser') # Navegador usado  
    cells.insert(4, 'Environment') # Entorno (dev/staging/prod)  
  
def pytest_html_results_table_row(report, cells):  
    cells.insert(2, getattr(report, 'page_url', '-'))  
    cells.insert(3, getattr(report, 'browser_name', '-'))  
    cells.insert(4, getattr(report, 'environment', '-'))
```

Importante

- **No elimines** tu código existente en `conftest.py`
- Los hooks se ejecutan automáticamente - no necesitas llamarlos
- Si no guardas `page_url` en un test, aparecerá - en la columna
- La posición `2` coloca la columna después de "Test ID"

Tu reporte HTML ahora tendrá contexto visual inmediato sobre dónde ocurrió cada fallo, haciendo el debugging mucho más eficiente.

Capturas de pantalla automáticas

Ya tenemos nuestros reportes HTML configurados y funcionando, pero aún podemos **hacerlos mucho más útiles**, sobre todo cuando trabajamos con pruebas de interfaz gráfica (UI). Imaginá que un test falla: ver el error en el reporte ayuda, pero **ver qué estaba pasando en la pantalla en ese momento puede hacer la diferencia entre resolver el problema en segundos o estar una hora adivinando**.



Por eso, en esta sección vamos a aprender cómo integrar **capturas de pantalla automáticas** que se generen **solo cuando un test falla**, y que se adjunten directamente al reporte HTML. Esto nos permite entender visualmente el contexto del error, sin necesidad de reproducirlo manualmente. Además, es una práctica muy valorada en cualquier equipo de QA automatizado.

Escenario típico sin capturas:

None

```
✖ test_checkout_payment FAILED
  AssertionError: Expected "Payment successful" but got
    "Payment failed"
```

¿Qué preguntas surgen?

- ¿Estaba en la página correcta?
- ¿Se cargaron todos los elementos?
- ¿Había algún error visual no detectado?
- ¿El formulario estaba completo?

Con capturas automáticas: Una imagen responde todas estas preguntas al instante.

Implementación paso a paso

Paso 1: Preparación del entorno

Python

```
# conftest.py - Agrega al inicio del archivo
import pytest
import pathlib

# 📁 Carpeta donde guardaremos las capturas
target = pathlib.Path('reports/screens')
target.mkdir(parents=True, exist_ok=True) # Crea si no existe
```

Paso 2: Hook de captura inteligente

Python

```
@pytest.hookimpl(tryfirst=True, hookwrapper=True)
def pytest_runtest_makereport(item, call):
    """Se ejecuta después de cada fase (setup/call/teardown)
    de cada test."""
    outcome = yield # Dejamos que Pytest genere el
    # reporte
    report = outcome.get_result() # Obtenemos el objeto report

    # 🎯 Solo capturamos en fallos de la fase principal
    if report.when == 'call' and report.failed:
        driver = item.funcargs.get('driver') # Obtenemos el
        # fixture driver
        if driver:
            file_name = target / f"{item.name}.png"
            driver.save_screenshot(str(file_name)) # 📸

        ¡Captura!

    # 📄 Adjuntamos al reporte HTML
    if hasattr(report, 'extra'):
        report.extra.append({
            'name': 'screenshot',
            'format': 'image',
            'content': str(file_name)
        })
```

Anatomía del hook explicada

Componente	¿Qué hace?	¿Por qué es importante?
<code>tryfirst=True</code>	Ejecuta este hook antes que otros	Captura el estado exacto del fallo
<code>hookwrapper=True</code>	Permite envolver la ejecución	No interfiere con el flujo normal
<code>outcome = yield</code>	Pausa para que Pytest haga su trabajo	Obtenemos el reporte real
<code>report.when == 'call'</code>	Solo en la fase de ejecución principal	Evita capturas en setup/teardown
<code>report.failed</code>	Solo cuando el test falló	No desperdicia espacio con tests exitosos

Flujo de ejecución visual

None

Test ejecutándose → ❌ FALLA → Hook detecta fallo → 📷
Captura → 📋 Adjunta al reporte

Ventajas de esta implementación

Automática

- Cero configuración por test
- No necesitas recordar capturar manualmente

Eficiente

- Solo captura cuando hay fallos
- Evita imágenes duplicadas de setup/teardown

Integrada

- Las imágenes aparecen directamente en el reporte HTML
- Un solo archivo contiene todo el contexto

Estructura de archivos resultante

None

```
proyecto/

├─ reports/

|   ├─ report.html          # Reporte principal
|   └─ screens/            # Capturas automáticas
|       └─ test_login_invalid.png
|       └─ test_checkout_error.png
|       └─ test_payment_failed.png
|
└─ conftest.py             # Hook configurado
```

Consideraciones importantes

- **Solo para tests UI:** El hook verifica que exista el fixture `driver`
- **Naming automático:** Los archivos se nombran según el test (`{item.name}.png`)
- **Sobreescritura:** Si ejecutas pytest dos veces, las capturas se reemplazan
- **Dependencia:** Requiere que pytest-html esté instalado y configurado

Resultado en el reporte

Cuando abras `reports/report.html`, los tests fallidos mostrarán:

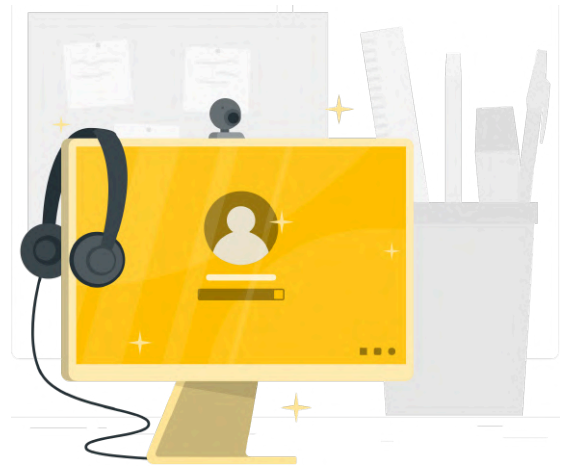
1. **Error message** - Qué falló
2. **Traceback** - Dónde falló
3. **Screenshot** - Cómo se veía la página

Con esto ahora tenemos debugging completo en segundos en lugar de minutos

Logging con el módulo logging

Hasta ahora nos enfocamos en **ver qué pasó en nuestras pruebas** mediante reportes HTML y capturas visuales. Pero aún nos falta una pieza clave para entender a fondo **cómo ocurrió** cada paso: los logs.

El **logging** nos permite registrar de forma cronológica todo lo que sucede durante la ejecución de nuestras pruebas: qué acciones se realizaron, con qué datos, en qué momento y con qué resultado. A diferencia del reporte, que resume, el log cuenta **la historia completa paso a paso**.



En esta sección vamos a aprender a configurar un logger centralizado usando el módulo **logging** de Python. Este logger escribirá automáticamente en un archivo todo lo que va ocurriendo en nuestros tests, permitiéndonos hacer un análisis mucho más detallado cuando algo sale mal. Además, es una base fundamental de **observabilidad**, algo que va a ser parte del entregable final del proyecto.

Vamos a ver cómo implementarlo de forma simple y reutilizable.

Reportes vs Logs: complementos perfectos

- **Reportes:** "El test `test_checkout` falló a las 14:30"
- **Logs:** "14:30:15 Navegando a `/cart` → 14:30:16 Agregando producto → 14:30:18 ERROR: Botón 'Checkout' no encontrado"

Los logs cuentan la **historia completa** paso a paso: qué usuario usamos, a qué URL fuimos, cuánto tardó cada request.

Configuración

Paso 1: Crear el archivo de configuración

Crea el archivo `utils/logger.py`:

```
Python
import logging
import pathlib

# 📁 Carpeta donde se almacenarán los logs
audit_dir = pathlib.Path('logs')
audit_dir.mkdir(exist_ok=True)

# ⚙️ Configuración global (una sola vez)
logging.basicConfig(
    filename=audit_dir / 'suite.log',      # Archivo único
    level=logging.INFO,                   # INFO y superiores
    format='%(asctime)s %(levelname)s %(name)s - %(message)s',
    datefmt='%H:%M:%S'
)

# 🛠️ Logger específico para tus tests
logger = logging.getLogger('talentolab')
```

Paso 2: Entender el formato de salida

```
None
14:30:15 INFO talentolab - Iniciando login con usuario:
standard_user
14:30:16 INFO talentolab - Completando credenciales...
14:30:18 ERROR talentolab - Botón 'Checkout' no encontrado
```

Componente	Ejemplo	¿Qué aporta?
<code>%(asctime)s</code>	14:30:15	Cronología exacta
<code>%(levelname)s</code>	INFO/ERROR	Severidad del evento
<code>%(name)s</code>	talentolab	Contexto del logger
<code>%(message)s</code>	Iniciando login...	Acción específica

Uso práctico en tests

Una vez que el logger está configurado, podemos usarlo en cualquier test importando el objeto `logger`. Por ejemplo:

Python

```
from utils.logger import logger

def test_login_usuario_valido(driver):
    logger.info('Iniciando login con usuario: %s',
                'standard_user')

    login_page = LoginPage(driver)
    login_page.abrir()

    logger.info('Completando credenciales...')
    login_page.login_completo('standard_user', 'secret_sauce')

    logger.info('Verificando redirección exitosa')
    assert "inventory.html" in driver.current_url

    logger.info('Test de login completado exitosamente')
```

Esto genera un registro paso a paso de lo que ocurre en el test, que se guarda automáticamente en el archivo `logs/suite.log`.

Resultado en `logs/suite.log`

Después de ejecutar el test anterior, el archivo `logs/suite.log` contendrá algo como esto:

None

```
14:25:10 INFO talentolab - Iniciando login con usuario:
standard_user
14:25:11 INFO talentolab - Completando credenciales...
14:25:12 INFO talentolab - Verificando redirección exitosa
14:25:12 INFO talentolab - Test de login completado
exitosamente
14:25:15 INFO talentolab - Iniciando test_add_to_cart...
```


¿Qué aporta cada campo?

Componente	Ejemplo	¿Qué aporta?
%(asctime)s	14:25:10	Hora exacta del evento
%(levelname)s	INFO	Nivel de importancia (INFO, ERROR...)
%(name)s	talentolab	Contexto del logger (útil en proyectos grandes)
%(message)s	Iniciando login...	Acción concreta que se está registrando

Ventajas clave

- **Un solo archivo** para toda la suite
- **Cronología precisa** de todas las acciones
- **Crecimiento automático** con cada ejecución
- **Debugging eficiente** sin reproducir escenarios

Tip avanzado: Rotación automática

Si tu suite genera muchos logs, agrega rotación:

Python

```
from logging.handlers import RotatingFileHandler

handler = RotatingFileHandler(
    audit_dir / 'suite.log',
    maxBytes=1024*1024, # 1MB por archivo
    backupCount=5       # Mantener 5 archivos históricos
)
```

Con esta configuración, tendrás un registro detallado y cronológico de cada ejecución, perfecto para análisis post-mortem.

Estrategias de gestión de errores



A esta altura ya contamos con una base técnica sólida: tenemos reportes visuales, capturas automáticas de pantalla y logs detallados que registran todo lo que ocurre. Pero para que nuestra suite de testing sea **robusta, confiable y mantenible**, necesitamos pensar también **qué pasa cuando los tests fallan**.

No todos los errores son iguales. Algunos son intermitentes, otros son causados por fallos de red, otros por datos que quedaron "sucios" de pruebas anteriores. Por eso, vamos a incorporar tres estrategias que nos permiten **gestionar los errores de forma más inteligente**:

1. **Soft-asserts** → para no detenernos en el primer fallo
2. **Retries** → para que los errores temporales no rompan la ejecución
3. **Teardown limpio** → para que un test no contamine a los demás

Veamos cada una en detalle.

1. Soft-asserts – diagnóstico completo

El problema:

Con un `assert` tradicional, si una validación falla, el test se corta y **no se ejecutan las siguientes verificaciones**. Esto puede hacer que perdamos información útil..

Python

```
# ❌ Con assert normal - se detiene en el primer fallo
assert resp.status_code == 200           # FALLA aquí
assert 'id' in data                       # NUNCA se ejecuta

assert elapsed < 1                        # NUNCA se ejecuta
```

La solución:

Con **soft-asserts**, usamos una librería como `pytest-check` para que todas las validaciones se ejecuten aunque alguna falle. Así podemos reportar **todos los errores de un test juntos**.

Shell

```
pip install pytest-check
```

Con soft-asserts vamos a hacer que todos los tests corran aunque alguno de ellos falle.

Python

```
import pytest_check as check
def test_api_validaciones_múltiples():
    resp = requests.get('https://jsonplaceholder.typicode.com/users/1')

    #  Verifica TODO aunque algo falle
    check.equal(resp.status_code, 200, "El status no es 200")

    data = resp.json()
    check.is_in('id', data, "Falta la clave 'id'")

    elapsed = resp.elapsed.total_seconds()
    check.is_true(elapsed < 1, f"Demoró {elapsed:.2f}s > 1s")
```

Resultado: Pytest lista **todos** los fallos en un solo reporte. ¡Diagnóstico completo de una vez!

2. Retries automáticos – resiliencia ante fallos intermitentes

Un **retry** es volver a ejecutar automáticamente un test que falló, esperando que el segundo (o tercer) intento sea exitoso. Es como cuando pierdes conexión a internet y tu navegador te pregunta "¿Reintentar?"

El problema:

A veces los tests fallan por razones ajenas al código: latencia de red, caídas momentáneas de APIs, o inestabilidad del entorno de CI.

Escenario típico sin retries

```
None
🔄 Test ejecutándose...
🌐 Llamada a API externa...
🕒 Timeout por latencia de red (5.1 segundos, límite: 5s)
❌ FAILED - Test marcado como fallido
😞 "¿Pero la API funciona bien... fue solo un problema temporal?"
```

La solución:

Podemos **reintentar automáticamente** los tests que fallan usando el plugin `pytest-rerunfailures`.

El mismo escenario con retries

```
None
🔄 Test ejecutándose...
🌐 Llamada a API externa...
🕒 Timeout por latencia de red
🔄 RETRY 1: Reintentando en 3 segundos...
🌐 Segunda llamada a API...
✅ PASSED - Respuesta exitosa en 2.1 segundos
```

Implementación global

```
Shell

pip install pytest-rerunfailures
```

```
pytest --reruns 2 --reruns-delay 3 # 2 reintentos con 3s de pausa
```

Si cualquier test falla, reintentará hasta 2 veces esperando 3 segundos entre intentos.

Implementación selectiva

Python

```
@pytest.mark.flaky(reruns=2, reruns_delay=2)

def test_api_tiempo_aleatorio():

    # Solo este test se reintentará automáticamente

    resp = requests.get('https://api.externa.com/data')

    assert resp.status_code == 200
```

Solo los tests "inestables" se reintentan, ahorrando tiempo en tests determinísticos.

Los retries convierten fallos intermitentes en tests estables, pero úsalos con criterio: **son para problemas de infraestructura, no de código.**

3. Teardown limpio – tests independientes

Teardown es el proceso de "limpieza" que ocurre después de que un test termina. Es como lavar los platos después de cocinar: si no lo haces, el siguiente cocinero encontrará la cocina sucia.

El problema:

Cuando un test deja "basura" (usuarios creados, sesiones abiertas, datos sin borrar), puede **romper a otros tests** que dependen de un entorno limpio.

Imagina esta secuencia:

None

```
test_crear_usuario()  
└─ Crea usuario "test_user_123" ✓  
└─ ✗ NO lo borra al terminar  
  
test_crear_mismo_usuario()  
└─ Intenta crear "test_user_123"  
└─ ✗ FALLA: "Usuario ya existe"  
└─ 😡 ¡El test anterior contaminó este!
```

Un test "roto" hace fallar a otros tests que están perfectamente bien.

Python

```
# ✗ Test 1 crea un usuario y no lo borra  
# ✗ Test 2 falla porque el usuario ya existe
```


La solución:

Usamos **fixtures con yield** para crear y luego limpiar los datos automáticamente.

```
Python
@pytest.fixture
def usuario_temp():
    # 🛠️ Setup - crear recurso

    resp = requests.post('https://jsonplaceholder.typicode.com/posts',
                        json={'title': 'tmp', 'body': 'test'})
    post_id = resp.json()['id']

    yield post_id # ← Entrega al test

    # 🧹 Teardown - borrar recurso (SIEMPRE se ejecuta)

    requests.delete(f'https://jsonplaceholder.typicode.com/posts/{post_id}')
```

Usando el fixture en un test

```
Python
def test_modificar_post(usuario_temp):
    post_id = usuario_temp # Recibe el ID del post creado

    # Modificar el post

    resp = requests.put(f'https://jsonplaceholder.typicode.com/posts/{post_id}',
                        json={'title': 'actualizado'})

    assert resp.status_code == 200
    # ✅ Al terminar, el fixture AUTOMÁTICAMENTE borra el post
```

¿Cómo funciona este fixture?

1. **Setup (antes del test):** Crea el recurso temporal
2. **Yield:** Pausa y entrega el recurso al test
3. **Test se ejecuta:** Usa el recurso como necesite
4. **Teardown (después del test):** Limpia automáticamente

Garantías del teardown con yield

- **Se ejecuta SIEMPRE:** Incluso si el test falla con excepción
- **Es automático:** No tienes que recordar limpiar manualmente
- **Es confiable:** Pytest garantiza que el código después de `yield` se ejecute

Beneficios de tests independientes

- **Ejecutables en cualquier orden:** `pytest --random-order`
- **Paralelización segura:** `pytest -n 4` (con `pytest-xdist`)
- **Debugging más fácil:** Un test roto no afecta a otros
- **CI/CD estable:** No hay "fallos fantasma" por contaminación

El teardown limpio es la diferencia entre una suite de tests profesional y una que "a veces funciona".

Tabla resumen de estrategias

Estrategia	¿Cuándo usarla?	Implementación
Soft-asserts	Validar múltiples campos y reportar todos los fallos juntos	<code>pytest-check</code> o <code>try/catch</code> con <code>registro de errores</code>
Retries	Entornos inestables (servicio ajeno)	Plugin <code>pytest-rerunfailures</code> o GitHub Actions <code>max-attempts</code>
Teardown limpio	Cerrar sesiones, borrar datos de prueba	Fixtures con <code>yield</code> o <code>finalizer</code>

Con estas tres estrategias, tus tests serán:

- **Informativos** → Muestran todos los problemas
- **Resilientes** → Se recuperan de fallos temporales
- **Independientes** → No se afectan entre sí

Generando reportes en TalentoLab



Silvia abrió el ticket:



QA-145 – Reporte & Logging MVP:

Historia:

Como QA Lead necesito reportes HTML con capturas y un log unificado, para evaluar la estabilidad del framework antes de la demostración al cliente.

Criterios de aceptación:

1. Ejecutar `pytest` genera `reports/report.html` autocontenido
2. Los fallos UI incluyen captura
3. Existe `logs/suite.log` con nivel INFO+ y timestamps
4. Artefactos adjuntos en el workflow de GitHub Actions

Matías añade un comentario:



"Lo que hagas aquí quedará como la sección 'Generación de Reportes' del proyecto final. Asegúrate de integrarlo tanto con los tests UI de SauceDemo como con los tests API de JSONPlaceholder."

Tarea

1. Instala `pytest-html`, `pytest-rerunfailures` y actualiza `requirements.txt`
2. Añade carpeta `reports/` y `logs/` al repo (vacías, pero versionadas con `.gitkeep`)
3. Implementa el hook de captura en `conftest.py`
4. Crea archivo `pytest.ini` con título y metadata de autor

Conexión con el proyecto final

Este caso cubrirá el apartado de **Generación de reportes y logging** exigido en la entrega final y demostrará que tu framework produce evidencia legible para el cliente.

Próximos pasos

En la **Clase 14** nos adentraremos en **BDD con Behave**: escribirás criterios de aceptación en Gherkin que invocarán estos mismos tests por debajo, cerrando el círculo de trazabilidad.

¡Tu framework está casi completo!



Buenos Aires
aprende
Agencia de Habilidades para el Futuro

BA Buenos
Aires
Ciudad