«Talento Tech»

Automation Testing

Clase 12

Buenos Aires Ciudad

Clase N° 12: Automatización de Pruebas de API - Parte 2

Temario

- Métodos HTTP PUT, PATCH y DELETE
- Actualizar y eliminar recursos en JSONPlaceholder
- Validación avanzada de respuestas
    - Códigos de estado
    - Contenido y estructura JSON
    - Headers y tiempos
- Integración con Pytest (markers, fixtures comunes, asserts encadenados)
- Ejercicio práctico TalentoLab: ciclo de vida completo de un recurso

Objetivos de la clase

En esta clase vamos a profundizar en el testing de APIs aprendiendo a manejar distintos métodos HTTP más allá del clásico GET y POST. Nos enfocaremos en comprender la diferencia entre **PUT** (que reemplaza un recurso completo) y **PATCH** (que actualiza solo una parte), así como en automatizar pruebas de eliminación de datos usando **DELETE**. También incorporaremos validaciones más avanzadas, verificando no solo que las respuestas tengan los códigos correctos, sino que cumplan con la estructura esperada, los tipos de datos y los tiempos máximos de respuesta. Finalmente, vamos a extender nuestra carpeta `tests_api/` cubriendo estos tres métodos nuevos, consolidando así una base sólida de pruebas automatizadas para tu framework final.

Métodos PUT, PATCH y DELETE

¿Qué son? y ¿para qué sirven?

Cada método HTTP tiene un propósito específico. Entender la diferencia te permite elegir la herramienta adecuada para cada escenario de negocio.

Antes de escribir código conviene entender cuándo usamos cada verbo:

- **PUT ➜** Reemplazar por completo un recurso existente o crearlo si aún no existe. Ejemplo: cuando el usuario guarda su perfil completo desde un formulario de "Editar perfil"

- **PATCH ➜** Cambiar solo ciertos campos de un recurso. Útil para actualizar el job o la foto sin tocar el resto del objeto. Evita enviar todo el payload otra vez

- **DELETE ➜** Quitar un recurso de forma permanente. Caso típico: eliminar un usuario, borrar un archivo o dar de baja una suscripción

💡Conocer estos métodos te permite automatizar pruebas que reflejen cambios reales en la aplicación, simulando flujos completos de usuario y negocio.

Con la teoría clara, veamos la tabla de referencia:

Método	Uso típico	Idempotente	Ejemplo URL
PUT	Reemplaza/crea recurso completo	✅	`PUT /api/posts/1`
PATCH	Actualiza solo campos enviados	❌	`PATCH /api/posts/1`
DELETE	Elimina recurso	✅	`DELETE /api/posts/1`

📌**Idempotente** significa que repetir la llamada produce el mismo resultado (por eso PUT/DELETE lo son y PATCH no).

Actualizar y eliminar recursos en JSONPlaceholder

Usaremos JSONPlaceholder como API de práctica, aplicando cada método sobre el recurso `/posts/1`.

En esta sección haremos tres cosas:

1. **PUT:** Enviamos todos los campos y comprobamos que la respuesta los refleje.
2. **PATCH:** Solo modificamos un campo, validando que el resto no cambie.
3. **DELETE:** Borramos el recurso y confirmamos que el servidor lo reporte correctamente.

Cada llamada será seguida de aserciones que prueban que el servidor devolvió el código correcto y campos esperados.

PUT – reemplazo total

Vamos a crear nuestro primer test de actualización completa. PUT reemplaza todo el contenido del recurso, así que enviaremos un payload completo con todos los campos necesarios.

En este ejemplo:

- Enviaremos una petición PUT al endpoint **`/posts/1`** de **JSONPlaceholder**
- Incluiremos un payload con `id`, `title`, `body` y `userId`
- Validaremos que el servidor responda con código 200 y los campos actualizados - Mediremos el tiempo de respuesta para asegurar que sea menor a 1 segundo

El flujo será: preparar datos → enviar PUT → validar respuesta → verificar performance.

```
import requests
import pytest
import time

URL = 'https://jsonplaceholder.typicode.com/posts/1'

payload = {
    'id': 1,
    'title': 'Automation Testing Guide',
    'body': 'Guía completa de testing automatizado',
    'userId': 1
```

```
}

def test_put_post():
    start = time.time()
    r = requests.put(URL, json=payload)
    assert r.status_code == 200
    body = r.json()
    assert body['title'] == 'Automation Testing Guide'
    assert body['body'] == 'Guía completa de testing automatizado'
    assert body['id'] == 1
    assert r.elapsed.total_seconds() < 1

    print(f"✅ PUT completado en {r.elapsed.total_seconds():.3f}s")

# Para ejecutar como script normal
if __name__ == "__main__":
    test_put_post()
    print("Test completado exitosamente")
```

¿Qué hicimos?

Enviamos título + body + userId para reemplazar el post #1. JSONPlaceholder responde con los mismos campos actualizados.

PATCH – actualización parcial

Ahora probaremos PATCH, que es más eficiente cuando solo necesitas cambiar algunos campos específicos. A diferencia de PUT, PATCH no reemplaza todo el recurso, sino que actualiza únicamente los campos enviados.

En este test:

- Enviaremos solo el campo `title` en el payload (no todo el objeto)
- Validaremos que el servidor actualice únicamente ese campo
- Confirmaremos que los demás campos (`body`, `userId`) permanezcan intactos
- Verificaremos que la respuesta incluya tanto los campos actualizados como los originales

Primero definamos el payload parcial:

```
PATCH_PAYLOAD = { 'title': 'Título actualizado por PATCH' }
```

Y ahora el test:

```
def test_patch_post():
    r = requests.patch(URL, json=PATCH_PAYLOAD)
    assert r.status_code == 200
    body = r.json()
    assert body['title'] == 'Título actualizado por PATCH'
    # El resto de campos se mantienen
    assert 'body' in body
    assert 'userId' in body
    print(f"✅ PATCH completado - Solo título actualizado")
```

Solo cambia el campo `title`, deja el resto intacto.

DELETE – eliminar

Finalmente, completaremos el ciclo CRUD con la eliminación del recurso. **DELETE** es la operación más simple en términos de payload (no envía datos), pero requiere validaciones específicas para confirmar que la eliminación fue exitosa.

En este test:

- Enviaremos una petición DELETE sin payload al endpoint `**/posts/1**` Validaremos el código de respuesta (**200** en **JSONPlaceholder**, aunque típicamente sería 204) -
- Verificaremos que el cuerpo de la respuesta indique eliminación exitosa
- Confirmaremos que el recurso ya no contiene los datos originales

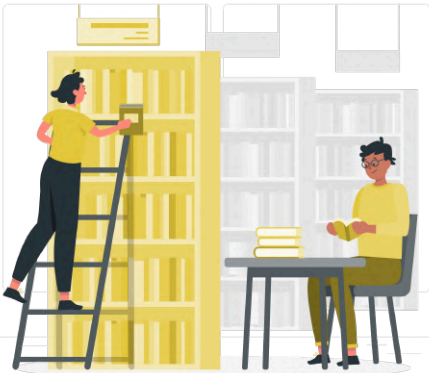
Ten en cuenta que diferentes APIs manejan DELETE de forma distinta: algunas devuelven 204 (No Content), otras 200 con confirmación, y algunas 404 si intentas acceder al recurso eliminado después.

```
def test_delete_post():
    r = requests.delete(URL)
    assert r.status_code == 200  # JSONPlaceholder devuelve 200, no 204
    # JSONPlaceholder simula la eliminación devolviendo objeto vacío
    body = r.json()
    assert body == {} or 'id' not in body or body['id'] is None
    print("✅ DELETE completado - Recurso eliminado")
```

Nota: JSONPlaceholder simula las operaciones, por lo que DELETE devuelve 200 en lugar del típico 204.

👉 Accedé acá a los ejemplos en el repositorio: Ejemplos

Validación avanzada de respuestas



Aplicando los 5 niveles a PUT, PATCH y DELETE

En la clase anterior aprendimos los 5 niveles de validación de respuestas con GET. Ahora los aplicaremos a los métodos de modificación (PUT, PATCH, DELETE) que tienen particularidades específicas.

Validaciones específicas para cada método HTTP

PUT - Validaciones de reemplazo completo:

Validaremos que PUT reemplace completamente el recurso y devuelva exactamente los datos que enviamos.

Python

```python
def test_put_with_advanced_validation():
    payload = {'id': 1, 'title': 'Nuevo título', 'body': 'Nuevo contenido', 'userId': 1}
    r = requests.put(URL, json=payload)

    # Nivel 1: Status code específico para PUT
    assert r.status_code == 200  # PUT actualiza, no crea

    # Nivel 2: Cabeceras - debe devolver JSON
    assert 'application/json' in r.headers['Content-Type']

    # Nivel 3: Estructura - debe contener TODOS los campos enviados
    body = r.json()
    sent_fields = set(payload.keys())
    received_fields = set(body.keys())
```

```
   assert sent_fields <= received_fields, f"PUT no devolvió
todos los campos: {sent_fields - received_fields}"

   # Nivel 4: Contenido - debe reflejar exactamente lo enviado
   assert body['title'] == payload['title'], "PUT no actualizó
el título"
   assert body['body'] == payload['body'], "PUT no actualizó
el cuerpo"

   # Nivel 5: Performance - PUT no debe ser más lento que GET
   assert r.elapsed.total_seconds() < 1.5, "PUT demasiado
lento"
```

PATCH - Validaciones de actualización parcial:

Verificaremos que PATCH actualice solo los campos enviados mientras preserva el resto del recurso intacto.

Python

```python
def test_patch_with_advanced_validation():
    patch_payload = {'title': 'Solo título actualizado'}
    r = requests.patch(URL, json=patch_payload)

    # Nivel 1: Status code
    assert r.status_code == 200

    # Nivel 2: Cabeceras
    assert 'application/json' in r.headers['Content-Type']

    # Nivel 3: Estructura - debe mantener campos originales + actualizados
    body = r.json()
    expected_fields = {'id', 'title', 'body', 'userId'}  # Campos completos del recurso
    assert expected_fields <= set(body.keys()), "PATCH eliminó campos existentes"

    # Nivel 4: Contenido específico de PATCH
    assert body['title'] == patch_payload['title'], "PATCH no actualizó el campo enviado"
    assert 'body' in body and body['body'] is not None, "PATCH no preservó campos no enviados"

    # Nivel 5: Performance - PATCH debe ser más rápido que PUT
    assert r.elapsed.total_seconds() < 1, "PATCH debería ser más rápido"
```

DELETE - Validaciones de eliminación:

Confirmaremos que DELETE elimine correctamente el recurso y maneje los diferentes códigos de respuesta posibles.

Python

```python
def test_delete_with_advanced_validation():
    r = requests.delete(URL)

    # Nivel 1: Status code - puede variar según la API
    assert r.status_code in [200, 204], f"DELETE devolvió
{r.status_code}, esperado 200 o 204"

    # Nivel 2: Cabeceras específicas de DELETE
    if r.status_code == 204:
        assert r.headers.get('Content-Length') == '0', "204 no
debe tener contenido"

    # Nivel 3: Estructura - cuerpo vacío o confirmación
    if r.status_code == 200:
        body = r.json()
        # JSONPlaceholder devuelve objeto vacío o con campos
null
        assert body == {} or all(v is None for v in
body.values()), "DELETE debe vaciar el recurso"

    # Nivel 4: Contenido - verificar que no hay datos del
recurso original
    if r.text:  # Si hay contenido
        body = r.json()
        assert body.get('id') != 1 or body.get('id') is None,
"El recurso no fue eliminado"

    # Nivel 5: Performance - DELETE debe ser la operación más
rápida
    assert r.elapsed.total_seconds() < 0.8, "DELETE demasiado
lento"
```

Diferencias clave en las validaciones

- **PUT**: Valida que la respuesta contenga exactamente lo que enviaste
- **PATCH**: Verifica que solo se actualicen los campos enviados, manteniendo el resto
- **DELETE**: Confirma que el recurso fue eliminado o vaciado correctamente

Implementación práctica

Combina estas validaciones en una función reutilizable:

Python

```python
def validate_api_response(response, expected_status,
expected_fields=None, max_time=1.0):
    """Función helper para validar respuestas API con los 5
niveles"""
    # Nivel 1: Status
    assert response.status_code == expected_status

    # Nivel 2: Headers
    if expected_status != 204:  # 204 No Content puede no tener
Content-Type
        assert 'application/json' in
response.headers.get('Content-Type', '')

    # Nivel 3-4: Estructura y contenido (si hay expected_fields)
    if expected_fields and response.text:
        body = response.json()
        assert expected_fields <= set(body.keys())

    # Nivel 5: Performance
    assert response.elapsed.total_seconds() < max_time

    return response.json() if response.text else {}
```

Esta aproximación te permite reutilizar las validaciones en todos tus tests mientras mantienes la especificidad de cada método HTTP.

Integración con Pytest – ejecutarlo todo con un solo comando

Conectando todo: de tests individuales a un framework robusto

Hasta ahora hemos creado tests individuales para PUT, PATCH y DELETE con validaciones avanzadas. Pero ejecutar cada test por separado no es práctico en un entorno profesional. Necesitamos:

- **Ejecutar todos los tests API de una vez** con un solo comando
- - **Organizar el código** para evitar duplicación de URLs y configuraciones
- - **Filtrar tests** según el contexto (solo API, solo regresión, etc.)
- - **Compartir datos** entre tests que dependen unos de otros

Aquí es donde Pytest transforma nuestros scripts sueltos en un framework de testing profesional.

Incorporar los tests API a Pytest nos da:

✅ **Uniformidad**: misma herramienta para UI y API ⇒ un solo reporte
✅ **Selectividad**: correr solo `-m api` en pipelines rápidos o todo en regresión
✅ **Fixtures reutilizables**: crear URLs, tokens o datos aleatorios en un único lugar

Fixture común de URL – reutiliza en todas las funciones

Hasta ahora hemos estado hardcodeando la URL `'https://jsonplaceholder.typicode.com/posts/1'` en cada test.

Esto crea varios problemas:

- Si cambia el dominio, tenemos que actualizarlo en 10+ archivos
- No podemos alternar fácilmente entre entornos (dev, staging, prod)
- Duplicamos código innecesariamente

Los fixtures resuelven esto creando valores compartidos que se inyectan automáticamente en tus tests cuando los necesitas.

`conftest.py` en `tests_api/`:

```python
import pytest

BASE = 'https://jsonplaceholder.typicode.com'

@pytest.fixture(scope='module')
def posts_url():
    return f"{BASE}/posts"

@pytest.fixture(scope='module')
def post_by_id_url():
    def _get_url(post_id):
        return f"{BASE}/posts/{post_id}"
    return _get_url
```

¿Cómo funciona esto?

- posts_url() devuelve la URL base para operaciones sobre la colección
- post_by_id_url() devuelve una función que construye URLs específicas según el ID
- scope='module' significa que se crea una vez por archivo de test, no en cada función

Ahora tus tests se ven así:

Python

```python
def test_put_post(post_by_id_url):
    url = post_by_id_url(1)  # Genera: https://jsonplaceholder.typicode.com/posts/1
    payload = {'id': 1, 'title': 'Nuevo título', 'body': 'Nuevo contenido', 'userId': 1}
    r = requests.put(url, json=payload)
    # resto del test...

def test_patch_post(post_by_id_url):
    url = post_by_id_url(1)
    patch_payload = {'title': 'Solo título actualizado'}
    r = requests.patch(url, json=patch_payload)
    # resto del test...
```

Los fixtures en pytest permiten crear datos o configuraciones reutilizables que se comparten entre múltiples tests, evitando duplicar código. En este ejemplo, se crean dos fixtures que generan URLs base para posts, donde **posts_url** devuelve la URL general y **post_by_id_url** devuelve una función que construye URLs específicas para cada post según su ID.

Markers – etiquetar para filtrar

En un proyecto real tendrás cientos de tests: UI, API, integración, unitarios, smoke tests, regresión completa. No siempre quieres ejecutar todos.

Los markers te permiten crear "etiquetas" para organizar y filtrar tus tests según el contexto.

¿Por qué necesitas markers?

- Pipeline rápido: ejecutar solo tests críticos (smoke) en cada commit.
- Testing de API: correr solo tests de backend cuando cambias la API.
- Regresión completa: ejecutar todo antes de un release.
- Debugging: aislar un grupo específico de tests que está fallando. Un marker es una etiqueta que Pytest agrega al test para seleccionarlo.

```
@pytest.mark.api
@pytest.mark.regression
def test_put_post_complete():
    # Test code here
    pass

@pytest.mark.api
@pytest.mark.smoke
def test_get_posts_basic():
    # Test crítico que debe pasar siempre
    pass

@pytest.mark.ui
@pytest.mark.regression
def test_login_form():
    # Test de interfaz para regresión completa
    pass
```

Ejecución selectiva:

```shell
pytest -m api                    # Solo tests de API
pytest -m regression             # Tests críticos para release
pytest -m "api and regression"   # Combinación: API + críticos
pytest -m "smoke and not ui"     # Smoke tests que no sean de UI
pytest -m "api or ui"            # Todos los tests de API o UI
```

Los markers en pytest son etiquetas que se agregan a los tests para categorizarlos y poder ejecutar solo grupos específicos según necesidades. En el ejemplo de arriba, el test está marcado con `@pytest.mark.api` y `@pytest.mark.regression`, permitiendo ejecutar solo tests de API o solo tests críticos de regresión usando comandos como `pytest -m api`.

Tip profesional: Define tus markers en `pytest.ini` para evitar warnings:

Si ejecutas `pytest -m api` sin configurar los markers previamente, Pytest te mostrará warnings como "PytestUnknownMarkWarning: Unknown marker". Para evitar esto y documentar qué significa cada marker en tu proyecto, crea un archivo de configuración.

```none
[tool:pytest]
markers =
    api: Tests de API/backend
    ui: Tests de interfaz de usuario
    smoke: Tests críticos que deben pasar siempre
    regression: Suite completa de regresión
```

Encadenar asserts – usar el id creado en POST

Hasta ahora hemos probado cada operación CRUD por separado usando IDs fijos (como `/posts/1`). Pero en la vida real, primero creas un recurso con POST y luego lo modificas o eliminas usando el ID que te devolvió el servidor.

Esta técnica se llama "encadenamiento de tests" y simula el flujo real de una aplicación: crear → leer → actualizar → eliminar.

El problema con IDs fijos:

- No pruebas el flujo completo de la aplicación
- Los tests no son independientes entre sí
- No validas que el ID generado en POST funcione en operaciones posteriores

La solución: fixtures que crean datos reales

Cuando necesitas datos generados en tiempo real (por ejemplo el id que devuelve un POST) puedes compartirlos con otros tests:

Python

```python
@pytest.fixture(scope='module')
def created_post():
    """Crea un post y devuelve sus datos para otros tests"""
    payload = {
        'title': 'Post para testing',
        'body': 'Contenido de prueba',
        'userId': 1
    }

    response =
requests.post('https://jsonplaceholder.typicode.com/posts',
json=payload)
    assert response.status_code == 201

    return response.json()

def test_update_created_post(created_post):
    """Usa el post creado en la fixture"""
    post_id = created_post['id']

    update_payload = {'title': 'Título actualizado'}
```

```
    response =
requests.patch(f'https://jsonplaceholder.typicode.com/posts/{p
ost_id}',
                               json=update_payload)

    assert response.status_code == 200
    assert response.json()['title'] == 'Título actualizado'

def test_delete_created_post(created_post):
    """Elimina el post creado en la fixture"""
    post_id = created_post['id']

    response =
requests.delete(f'https://jsonplaceholder.typicode.com/posts/{
post_id}')
    assert response.status_code == 200
```

¿Cómo funciona el encadenamiento?

1. El fixture `created_post` ejecuta un POST real y guarda la respuesta
2. Todos los tests que reciban `created_post` como parámetro usarán los mismos datos
3. El `scope='module'` asegura que el POST se ejecute solo una vez por archivo
4. Los tests subsiguientes usan el ID real generado por el servidor

Esta técnica permite crear un flujo de tests dependientes donde un fixture crea un recurso real (POST) y devuelve sus datos para ser reutilizados por otros tests. En el ejemplo, el fixture `created_post` crea un post y devuelve su información completa, luego el test `test_update_created_post` usa el ID generado para actualizar ese mismo post, creando así un flujo de testing más realista que simula el comportamiento real de la aplicación.

Ventajas del encadenamiento:

- Pruebas el flujo completo de la aplicación
- Validás que los IDs generados funcionen correctamente
- Tests más realistas que reflejan el uso real
- Detección temprana de problemas de integración entre operaciones CRUD

Automatizando TalentoLab

Silvia creó en Jira la tarea **QA-132 – Validar ciclo de vida de recurso (v2)** en el proyecto TalentoLab – API Migration.

Jira Ticket: QA-132

- **Resumen:** Automatizar ciclo de vida completo de post (create-update-delete) en JSONPlaceholder
- **Tipo:** Story
- **Prioridad:** Alta 🟠

Criterios de Aceptación:

1. Se debe crear un post con datos aleatorios y recibir 201 + id
2. Se debe actualizar el título usando PATCH y recibir 200
3. Se debe eliminar el post y recibir 200 (JSONPlaceholder)
4. El flujo debe ejecutarse en menos de 3s y sin fallos

Definition of Done:

- ✔ Test marcado `@pytest.mark.e2e` pasa en CI
- ✔ Reporte HTML adjunto al pipeline

Ejercicio Práctico

Matías se acerca y te explica cómo realizar esa tarea de Jira:

1. En `tests_api/` crea `test_post_lifecycle.py`
2. **POST** `/posts` con Faker (title, body). Guarda el id
3. **PATCH** `/posts/{id}` actualizando título a "Título actualizado por QA"
4. **DELETE** `/posts/{id}` y valida 200
5. Usa asserts de esquema, tipo y tiempo
6. Marca el test con `@pytest.mark.e2e`

"Este caso cubrirá los tres métodos requeridos en la entrega final y demostrará que tu framework soporta operaciones CRUD."

Conexión con el proyecto final

Los tests de ciclo de vida que construyas hoy serán fundamentales porque:

1. **Demuestran CRUD completo**: Create, Read, Update, Delete en un solo flujo
2. **Validaciones avanzadas**: Esquema, tipos, performance y headers
3. **Integración con Pytest**: Fixtures, markers y reportes unificados
4. **Preparación para CI/CD**: Tests end-to-end que validan flujos críticos

En tu proyecto final, estos tests mostrarán que puedes automatizar APIs complejas con validaciones profesionales.

Próximos pasos

En la **Clase 13** generaremos reportes HTML combinando UI + API y añadiremos logging y screenshots para completar la trazabilidad del framework.

¡El framework está tomando forma profesional!


Buenos Aires
aprende
Agencia de Habilidades para el Futuro
Buenos
Aires
Ciudad