«Talento Tech»

Automation Testing

Clase 11

Buenos Aires Ciudad

Clase N° 11: Automatización de Pruebas de API - Parte 1

Temario

- ¿Por qué probar APIs además de la interfaz?
- Fundamentos de HTTP y REST (métodos, códigos de estado, JSON)
- Instalación y uso básico de la librería Requests
- Métodos GET y POST – ejemplos con la API pública reqres.in
- Validación de respuestas (status, cuerpo, cabeceras, tiempo)
- Ejercicio práctico TalentoLab – suite mínima de API que sumará al proyecto final

Objetivos de la clase

En esta clase vamos a descubrir cómo las **APIs** pueden acelerar y potenciar nuestras pruebas automatizadas, permitiendo aislar la lógica de negocio de la interfaz de usuario. Aprenderemos a enviar solicitudes GET y POST utilizando la librería `requests`, y a validar tanto los códigos de respuesta como el contenido recibido. Además, veremos cómo conectar los datos obtenidos a través de la API con los tests de Selenium que ya hemos desarrollado, logrando así una integración real entre pruebas de backend y frontend. Finalmente, sentaremos las bases para crear una carpeta `tests_api/` que será parte clave del framework final del proyecto.

Application Programming Interface

Una **API (Application Programming Interface)** es una puerta de entrada para que dos sistemas intercambien datos. En lugar de hacer clic por la interfaz, envías mensajes HTTP (GET, POST, etc.) y recibes respuestas en JSON o XML.

Piensa en ella como el camarero que lleva tu pedido (parámetros) a la cocina (servidor) y trae de vuelta el plato (respuesta) sin que veas el proceso interno.

¿Por qué probar APIs?

Probar la API antes o paralelo a la UI aporta ventajas decisivas:

✅ **Velocidad**: una llamada REST dura milisegundos y no necesita arrancar un navegador

✅ **Aislamiento de capas**: si la UI falla, pero la API responde bien, sabes que el problema es de frontend; si la API falla, no pierdes tiempo depurando selectores

✅ **Mayor cobertura lógica**: reglas de negocio como límites de cantidades, descuentos, stock se aplican en el servidor; validarlas solo por UI deja huecos

✅ **Fiabilidad en CI/CD**: los tests API son menos frágiles a cambios de diseño y cargan menos infraestructura (no abren ventanas)

✅ **Datos para la UI**: a veces necesitas crear un usuario o producto vía API para que exista en la base antes de lanzar la prueba UI

Te conviene siempre pensar: **"Primero API, luego UI"**. Así acotamos fallos y reducimos el tiempo de feedback.

Breve repaso de HTTP y REST

Estos son los métodos que utilizaremos para testear nuestras pruebas:

Método	Acción típica	Idempotencia	Ejemplo de URL
GET	Obtener recursos	✅	`GET https://api.example.com/products/42`
POST	Crear recurso / iniciar sesión	❌	`POST https://api.example.com/orders`
PUT	Reemplazar recurso	✅	`PUT https://api.example.com/products/42`
PATCH	Actualizar parcialmente	❌	`PATCH https://api.example.com/products/42/price`
DELETE	Eliminar	✅	`DELETE https://api.example.com/products/42`

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

Códigos importantes:

Luego de enviar una petición REST, obtenemos como respuesta un código. Puede ser de éxito o error. Estos son los que debieras tener presentes para seguir adelante:

- **2XX** = éxito
- **4XX** = error cliente
- **5XX** = error servidor

Recuerda que el cuerpo suele viajar en **JSON** (application/json).

La librería Requests

Hasta ahora, la mayoría de nuestras pruebas de automatización se enfocaron en la interfaz gráfica (UI) usando Selenium. Pero en el mundo real, la lógica de negocio, los datos y muchas validaciones importantes de una aplicación suelen estar en el backend, expuestos a través de APIs. Por eso, aprender a probar APIs es fundamental para lograr una cobertura completa y detectar problemas antes de que lleguen a la interfaz.

Requests es la librería estándar en Python para interactuar con APIs. Nos permite, con unas pocas líneas de código, enviar solicitudes HTTP como GET o POST, recibir respuestas, analizar el código de estado, leer cabeceras y convertir respuestas JSON directamente en diccionarios de Python.
Esto hace que automatizar pruebas sobre APIs sea simple, rápido y robusto.

¿Por qué es útil para testing?

- Velocidad y aislamiento: Probar APIs es mucho más rápido que abrir navegadores, y te permite detectar si los problemas están en el backend o la UI.
- Mantenimiento: Los tests de API no se rompen con cambios en la interfaz.
- Preparación para la UI: Muchas veces, necesitás crear datos vía API antes de ejecutar pruebas de interfaz.
- Mejores prácticas de la industria: Los equipos profesionales automatizan la validación de APIs para asegurar la calidad del software antes de liberar cambios.

Instalación: pip install requests

¿Cómo se usa Requests?

Con Requests, podés hacer cosas como:

```
import requests

# 1) Hacemos la solicitud HTTP
resp = requests.get('https://api.github.com/')
# 2) Imprimimos el código de estado
print(resp.status_code)  # 200 = OK
# 3) Vemos el tipo de contenido devuelto
print(resp.headers['content-type'])  # application/json
# 4) Convertimos el JSON a dict de Python
payload = resp.json()
print(payload['current_user_url'])
```

Para correr el ejemplo solo debes ejecutar:
`python3 nombre_del_archivo.py`

¿Qué es lo que hicimos?

En este ejemplo realizamos una solicitud GET para demostrar las capacidades básicas de Requests. Accedimos a la información de la respuesta incluyendo el código de estado, las cabeceras y convertimos el cuerpo JSON a un diccionario Python. Esta implementación sencilla muestra cómo las APIs pueden ser interrogadas con pocas líneas de código limpio y legible.

Importante: `resp.json()` hace el deserializado automático: transforma la cadena JSON en estructura Python para que asserts y comparaciones sean sencillos.

Entonces, **Requests** es la herramienta que nos permite automatizar pruebas de API de manera simple, directa y legible, tal como Selenium nos sirvió para la UI. Usar Requests es el primer paso para aislar la lógica de negocio y empezar a construir un framework de testing completo, moderno y listo para la vida real.

GET y POST con la API reqres.in

A lo largo de lo que resta del curso usaremos **reqres.in** como nuestro entorno de práctica de APIs. Es un servicio de demostración que expone endpoints REST predecibles y seguros, perfectos para aprender sin credenciales ni riesgo de romper datos reales.

Cada ejemplo que construyamos aquí se quedará en tu framework final: luego podrás cambiar la URL base para apuntar a cualquier back-end real.

Solicitud GET – listar usuarios

Empezaremos con el método más común: GET. Este endpoint nos permite obtener una lista de usuarios paginada. La URL incluye el parámetro ?page=2 para solicitar específicamente la segunda página de resultados.

En este ejemplo crearemos nuestro primer test automatizado que:

- Envía una solicitud GET a la API de reqres.in
- Verifica que el servidor responda correctamente (código 200)
- Confirma que recibimos la página solicitada
- Valida que efectivamente hay usuarios en la respuesta

Python

```python
import requests
import pytest

URL = 'https://reqres.in/api/users?page=2'

def test_get_users():
    r = requests.get(URL)
    assert r.status_code == 200
    data = r.json()
    assert data['page'] == 2
    assert len(data['data']) > 0  # al menos un usuario
```

`data['data']` contiene la lista de usuarios y cada elemento es un dict con `id`, `email`, `first_name`, `last_name` y `avatar`. Estamos verificando:

- **Status 200** → el servidor respondió correcto
- **page == 2** → la API devolvió la página solicitada
- **len(data) > 0** → al menos un resultado para no procesar listas vacías

Solicitud POST – crear un usuario

Ahora veremos cómo crear recursos usando el método POST. A diferencia del GET que solo consulta información, POST envía datos al servidor para crear un nuevo registro.

En este test automatizado vamos a:

- Preparar los datos del usuario en formato JSON (nombre y trabajo)
- Enviar una solicitud POST con esos datos como payload
- Verificar que el servidor confirme la creación con código 201
- Validar que la respuesta contenga los datos enviados y campos adicionales generados por el servidor

Python

```python
import requests

CREATE_URL = 'https://reqres.in/api/users'

def test_create_user():
    payload = {'name': 'Matias QA', 'job': 'tester'}

    r = requests.post(CREATE_URL, json=payload)
    assert r.status_code == 201  # Created
    new_user = r.json()
    assert new_user['name'] == 'Matias QA'
    assert 'id' in new_user and 'createdAt' in new_user
```

Al enviar el JSON, el servidor genera un nuevo registro y responde con:

- **201 Created** → confirma que el recurso fue creado
- **Campo id** → identificador único que podríamos usar en una prueba DELETE más adelante
- **Campo createdAt** → timestamp ISO 8601 para validar la fecha del servidor

Endpoint de login (extra)

El login es uno de los endpoints más críticos en cualquier aplicación. Aquí practicaremos cómo automatizar la autenticación enviando credenciales y validando la respuesta del servidor.

En este test vamos a:

- Enviar credenciales válidas (email y password) mediante POST
- Verificar que el servidor acepte la autenticación con código 200
- Confirmar que recibimos un token de acceso en la respuesta
- Preparar la base para futuros tests de escenarios negativos (credenciales incorrectas)

Python

```python
import requests

LOGIN_URL = 'https://reqres.in/api/login'
def test_login_successful():
        creds = {'email': 'eve.holt@reqres.in', 'password': 'cityslicka'}
   resp = requests.post(LOGIN_URL, json=creds)

   assert resp.status_code == 200
   assert 'token' in resp.json()
```

Este código demuestra una autenticación simulada: si envías email + password correctos recibes un token (200). Si omites el campo password la API responde 400 Bad Request, ideal para practicar escenarios negativos.

Validación de respuestas

Cuando automatizas APIs no basta con "llegar al 200". Un test robusto debe verificar múltiples aspectos de la respuesta para garantizar que la API funciona correctamente en todos los niveles. Piensa en esto como una inspección de calidad por capas:

Nivel 1: Status code – la primera barrera de calidad

```python
assert r.status_code == 200  # éxito
```

El código de estado te dice si la operación fue exitosa antes de analizar el contenido. Cambia el valor según el escenario: 201 para creación, 400 para datos inválidos, 404 para recurso no encontrado, etc.

Nivel 2: Cabeceras – garantizan tipo y codificación

```python
assert r.headers['Content-Type'] == 'application/json; charset=utf-8'
```

Las cabeceras confirman que el servidor envía el formato esperado. Si esperas JSON pero recibes HTML, algo está mal en el servidor o en tu solicitud.

Nivel 3: Estructura JSON – evita que falten campos clave

```python
data = r.json()

assert {'id', 'email', 'first_name'} <= set(data.keys())
```

Esta validación usa operaciones de conjuntos para verificar que todos los campos obligatorios estén presentes. Si falta algún campo, el test fallará inmediatamente.

Nivel 4: Contenido específico – valida reglas de negocio

```python
assert data['email'].endswith('@reqres.in')

assert data['id'] > 0  # IDs deben ser positivos

assert len(data['first_name']) > 0  # nombres no vacíos
```

Aquí verificas que los datos cumplan las reglas de tu aplicación: formatos de email, rangos numéricos, longitudes mínimas, etc.

Nivel 5: Tiempo de respuesta – rendimiento básico

```python
assert r.elapsed.total_seconds() < 1, 'API muy lenta'
```

Una API lenta puede ser técnicamente correcta pero inutilizable. Establece límites de tiempo para detectar problemas de performance antes de que lleguen a producción.

Tip profesional: Implementa estas validaciones gradualmente. Empieza con status code y estructura, luego agrega validaciones de negocio específicas según tu aplicación.

Agregar estas aserciones por capas hará que tu suite detecte regresiones de estructura, datos y performance, convirtiéndote en la primera línea de defensa contra bugs que podrían afectar a los usuarios finales.

🗨️**Storytelling**

Automatizando APIS en TalentoLab

Como Automation Trainee en TalentoLab, tu rol crece: ahora debes garantizar que los datos que el equipo de backend entregue cumplan formato y tiempos.

Silvia (PO) te comenta durante la daily:

"El equipo de backend está migrando usuarios a producción. Antes de que la UI los consuma, necesitamos certificar que los endpoints respondan correctamente. Usa la misma estructura de datos que aplicaste en la Clase 10."

Matías (Automation Lead) agrega:

"Construye la carpeta `tests_api/` con la misma organización que `tests/` para UI. Lo que hagas hoy se integrará al framework final y se ejecutará en paralelo con Selenium en CI/CD."

Ejercicios Prácticos.

Estructura a crear: (La mayoría de los archivos los tienes generados en las clases pasadas)

```
proyecto/
├── tests_api/
│   ├── __init__.py
│   ├── test_login_api.py
│   ├── test_users_api.py
│   └── test_create_user_api.py
├── utils/
│   └── api_utils.py  (nuevo)
└── pytest.ini  (actualizar)
```

Tareas específicas:

1. `test_login_api.py` (parametrizado):

- Debes validar los siguientes datos: → email `eve.holt@reqres.in`, password `cityslicka`, esperar 200 y token
- Datos inválidos → email sin password, esperar 400
- Usar `@pytest.mark.parametrize` con los casos

2. `test_users_api.py`:

- Hace GET a `/api/users?page=1` y verifica que cada usuario tenga las claves `id`, `email`, `first_name`, `last_name`
- Extra: validar que el `avatar` termina en `.jpg`

3. `test_create_user_api.py`:

- Envía POST `/api/users` con distintos nombres y trabajos
- Usa parametrización (puede ser con datos fijos o Faker)
- Comprueba 201 y que `createdAt` incluya el año actual

4. Configuración:

- Marca estos tests con `@pytest.mark.api`
- Actualiza `pytest.ini` para poder ejecutar `pytest -m api`

Comandos de ejecución esperados:

 # Ejecutar solo tests de API

pytest -m api -v

 # Ejecutar todo (UI + API)

pytest tests/ tests_api/ -v

 # Generar reporte combinado

pytest tests/ tests_api/ --html=reporte_completo.html

Conexión con el proyecto final

Los tests API que construyas hoy serán fundamentales para tu entrega final porque:

1. **Demuestran cobertura completa**: UI + API en un solo framework
2. **Preparación para CI/CD**: cuando subas a GitHub Actions (Clase 15) tendrás UI y API ejecutándose en paralelo
3. **Separación de responsabilidades**: si mañana la UI de SauceDemo cambia, tus tests API seguirán funcionando
4. **Datos para UI**: podrás crear usuarios vía API y luego validar que aparecen correctamente en la interfaz

Próximos pasos

En la **Clase 12** cubriremos métodos **PUT, PATCH y DELETE**, validaciones más finas (headers, tiempos) e integraremos los tests API a Pytest con markers y parametrización avanzada.


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