

«Talento Tech»

# Automation Testing

Clase 14



# Clase N.º 14 | Behavior-Driven Development (BDD) con Behave

## Temario

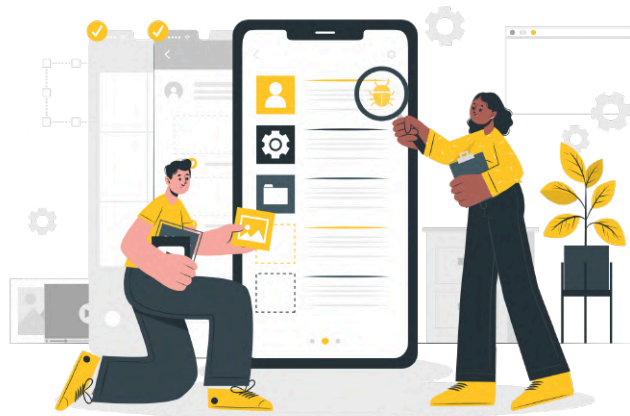
- Introducción: del requisito a la prueba viva
- Sintaxis Gherkin
- Dónde vive Behave en tu proyecto
- Instalación
- Tres ejemplos completos — Python, UI con SauceDemo y API con JSONPlaceholder
- Hooks globales: el backstage que prepara y limpia
- Ejecutar, reportar y compartir
- Integrar Behave a Pytest para unificar la orquesta

## Objetivo de la clase

En esta clase aprenderemos a transformar requisitos escritos en lenguaje natural en **pruebas automatizadas vivas** utilizando **Gherkin y Behave**. Exploraremos cómo escribir escenarios legibles por humanos que se conectan directamente con la ejecución de tests, permitiendo una trazabilidad clara entre lo que se espera y lo que se verifica. Veremos ejemplos prácticos en diferentes contextos (UI y API), aprenderemos a configurar e integrar Behave dentro de nuestro proyecto, y descubriremos cómo los **hooks globales** permiten preparar y limpiar el entorno de forma profesional. También veremos cómo **integrar Behave a Pytest** para consolidar reportes y ejecución, cerrando así el ciclo de automatización. Esta clase representa un paso clave hacia el entregable final, incorporando prácticas de **BDD (Behavior Driven Development)** para comunicar mejor con stakeholders y mantener una suite de pruebas alineada con los objetivos del producto.

# Introducción: del requisito a la prueba viva

En esta clase vamos a dar un paso importante en la evolución de nuestro framework: vamos a aprender a **convertir requisitos funcionales en pruebas automatizadas que cualquier persona del equipo pueda leer y entender**. Esa es la esencia de **BDD (Behavior-Driven Development)**: una práctica que busca acortar la distancia entre lo que se espera del sistema y lo que realmente se prueba.



En lugar de escribir casos de prueba técnicos que solo puede interpretar quien conoce el código, BDD propone que usemos un **lenguaje estructurado pero legible por humanos**, llamado **Gherkin**, para describir comportamientos del sistema. A partir de esos escenarios escritos en lenguaje natural, herramientas como **Behave** nos permiten automatizar su ejecución.

Este enfoque tiene un objetivo claro: que **todas las personas involucradas — testers, desarrolladores, analistas, product owners e incluso clientes — puedan comprender y validar qué se está probando y por qué**. Así, las pruebas dejan de ser una caja negra y se convierten en una **especificación ejecutable**.

Para entender este flujo, vamos a ver un ejemplo concreto: cómo un simple requerimiento como “un usuario válido debe poder iniciar sesión” se transforma en un archivo **.feature**, en código que automatiza ese comportamiento, y finalmente en un **reporte legible que demuestra que el sistema cumple con lo esperado**.

Con esta lógica de fondo, nos preparamos para conocer Gherkin, su sintaxis y empezar a escribir nuestras primeras pruebas vivas.

# ¿Qué es Gherkin?

**Gherkin** es un lenguaje de especificación diseñado para ser entendido por cualquier persona del equipo, sin importar si sabe programar o no. Utiliza palabras clave simples en inglés (o español) como “**Dado**” (**Given**), “**Cuando**” (**When**) y “**Entonces**” (**Then**) para describir el comportamiento de una aplicación de manera estructurada.



Por eso, cuando hablamos de “pruebas vivas”, nos referimos a **especificaciones que no solo documentan qué debe hacer el sistema, sino que además se ejecutan y generan evidencia real**. Y todo parte de escribir una **feature** correctamente.

En esta sección vamos a ver cómo se construye un archivo **.feature** desde cero, cómo estructurar los distintos escenarios, cómo usar conectores como “**And**” o “**But**”, y cómo aprovechar herramientas avanzadas como **Background**, **Scenario Outline** y **tablas de datos** para evitar repeticiones y cubrir más casos con menos esfuerzo. Vamos a comenzar con los elementos esenciales: **Feature**, **Scenario**, **Given**, **When** y **Then**.

## Características principales de Gherkin:

- **Legible para humanos:** Un analista de negocio puede leer y entender perfectamente una especificación
- **Ejecutable por máquinas:** Las herramientas como Behave pueden convertir estas especificaciones en pruebas automatizadas
- **Estructurado:** Sigue un formato consistente que facilita la comunicación entre equipos
- **Agnóstico del lenguaje:** Puede implementarse en Python, Java, JavaScript, etc.

## El flujo completo de BDD

Con Gherkin como base, el flujo de BDD es lineal y transparente:

1. **Requisito claro** → Lo escribimos en lenguaje natural (Gherkin) dentro de un archivo **.feature**
2. **Behave lo lee** → Cada línea **Given/When/Then** se conecta a una función Python llamada step-definition
3. **Step-definitions ejecutan lógica real** → Pueden llamar a Page Objects de Selenium, a funciones que consumen una API con requests, o a un módulo puro de Python
4. **Evidencia inmediata** → El resultado aparece en la consola y en tu reporte HTML, junto a los tests de Pytest



Así cerramos el ciclo completo: **requisito** → **especificación** → **prueba automatizada** → **evidencia**.

### Ejemplo:

**Requisito:** "Un usuario válido debe poder iniciar sesión y ver el inventario."

**En Gherkin (login.feature):**

None

```
Feature: Login en SauceDemo
  Como usuario registrado
  Quiero poder iniciar sesión
  Para acceder al inventario de productos
  Scenario: Login exitoso con credenciales válidas
    Given estoy en la página de login
      When ingreso usuario "standard_user" y contraseña
        "secret_sauce"
      Then debería ver la página de inventario
```

### Lo que sucede:

- Lo traducimos a Gherkin (login.feature)
- Escribimos los steps que interactúan con LoginPage y InventoryPage
- Al correr Behave obtenemos un reporte verde (o rojo, si algo falla) que cualquiera del equipo puede leer

## ¿Por qué elegimos esta aproximación?

**Basicamente porque cualquier involucrado la puede entender:**

- **Product Owner:** Lee el feature y confirma que es exactamente lo que pidió
- **Desarrollador:** Implementa los step-definitions conectándolos con el código real
- **Tester:** Ejecuta las pruebas y obtiene reportes claros
- **Cliente:** Entiende qué está siendo probado sin conocimiento técnico

Con esta idea en mente, avancemos a la sintaxis que hace posible esa conversación estructurada.

## **BDD = Behavior-Driven Development**

Una forma de escribir los requisitos de negocio como historias legibles y ejecutables; la especificación se convierte en prueba viva.

## **Sintaxis Gherkin**

Gherkin es un lenguaje estructurado que sigue reglas simples pero específicas. **Cada archivo termina en .feature**, debe estar codificado en **UTF-8** y cada línea comienza con una **palabra clave** que define su propósito.

### **Lo esencial: Feature, Scenario y Given/When/Then**

#### **1. Feature: El paraguas funcional**

La **Feature** es el contenedor principal que describe una funcionalidad completa desde la perspectiva del usuario. Sigue el patrón de historia de usuario:

None

```
Feature: [Nombre de la funcionalidad]
  Como [tipo de usuario]
  Quiero [acción que desea realizar]
  Para [beneficio que espera obtener]
```

#### **2. Scenario: Un ejemplo concreto**

Cada **Scenario** representa un caso de uso específico dentro de la feature. Es un ejemplo ejecutable de cómo debería comportarse el sistema.

None

```
Scenario: Login exitoso con credenciales válidas
```

### 3. Given/When/Then: La estructura fundamental

Esta es la columna vertebral de cada escenario:

- **Given** (Dado): Establece el **contexto inicial** o precondiciones
- **When** (Cuando): Define la **acción** que realiza el usuario
- **Then** (Entonces): Describe el **resultado esperado** o verificación

None

```
Given estoy en la página de login
When ingreso usuario "standard_user" y contraseña
"secret_sauce"
Then debería ver la página de inventario
```

#### Ejemplo completo:

None

```
Feature: Compra de productos
  Como visitante del sitio web
  Quiero agregar productos al carrito
  Para luego comprarlos

  Scenario: Carrito vacío muestra contador 0
    Given estoy en la página de inventario
    And el carrito está vacío
    When visualizo el icono del carrito
    Then el contador muestra "0"
```

#### Conectores adicionales

- **And**: Continúa el tipo de paso anterior (Given, When, o Then)
- **But**: Similar a And, pero con connotación de contraste

# Herramientas avanzadas: Background, Outline y tablas

Una vez que dominamos la estructura básica de un escenario (**Given-When-Then**), llega el momento de **hacer nuestras pruebas más eficientes y expresivas**. Gherkin nos ofrece herramientas avanzadas que permiten **evitar repeticiones, reducir el código duplicado y probar múltiples casos en un mismo escenario**.

Vamos a ver tres recursos fundamentales para escalar nuestras pruebas en BDD:



## 1. Background: Preparación común

Cuando varios escenarios requieren los mismos pasos iniciales, en vez de repetirlos una y otra vez, podemos definirlos una sola vez con **Background**. Esto mejora la **legibilidad** y **mantenibilidad** del archivo.

None

Feature: Gestión de carrito

Background:

Given estoy registrado como usuario "standard\_user"  
And he iniciado sesión correctamente  
And estoy en la página de inventario

Scenario: Agregar producto al carrito

When selecciono "Sauce Labs Backpack"  
Then el contador del carrito muestra "1"

Scenario: Agregar múltiples productos

When selecciono "Sauce Labs Backpack"  
And selecciono "Sauce Labs Bike Light"  
Then el contador del carrito muestra "2"



## 2. Scenario Outline: Pruebas con múltiples datos

Cuando queremos probar **el mismo comportamiento con diferentes combinaciones de datos**, usamos **Scenario Outline** junto con **Examples**. Es ideal para validaciones como login, operaciones matemáticas o reglas de negocio repetitivas.

None

Scenario Outline: Validar precios de productos


Given estoy en la página de inventario

When busco el producto "<producto>"

Then el precio mostrado es "<precio>"

Examples:

producto	precio
Sauce Labs Backpack	\$29.99
Sauce Labs Bike Light	\$9.99
Sauce Labs Onesie	\$7.99

 Ventaja: un solo escenario, múltiples validaciones con datos distintos.

## 3. Tablas de datos

Las tablas pueden usarse en cualquier paso para manejar datos estructurados:

None

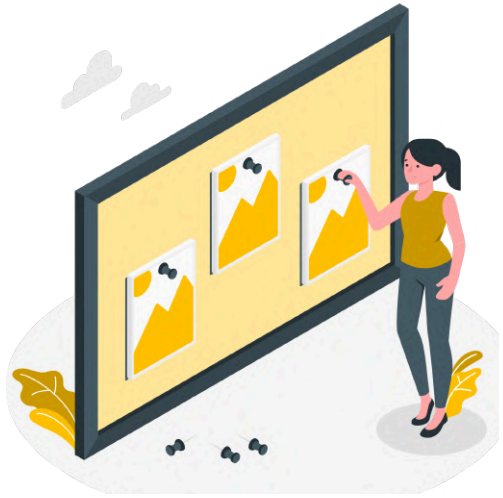
Given los siguientes usuarios están registrados:

usuario	email	rol
standard_user	standard@example.com	buyer
admin_user	admin@example.com	admin

# Tags: Pequeños post-its digitales

A medida que nuestra suite de pruebas crece, vamos a necesitar **organizar, filtrar y ejecutar escenarios específicos** según el contexto: pruebas rápidas, pruebas de regresión, escenarios que aún están en desarrollo, etc.

Para eso, Gherkin nos permite utilizar **tags** (etiquetas), que funcionan como pequeños **post-its virtuales** que se colocan sobre **Features** o **Scenarios**. Se escriben con **@** al comienzo de la línea y nos permiten **agrupar, seleccionar o excluir pruebas** fácilmente desde la línea de comandos o el pipeline de CI.



## ¿Para qué sirven los tags?

- Ejecutar solo pruebas críticas (@smoke)
- Ignorar escenarios en desarrollo (~@wip)
- Separar tests por tipo: UI, API, lógica
- Organizar pruebas por sprint, módulo o epic

## Tipos de tags más comunes:

### Por prioridad o tipo:

- @smoke: Pruebas críticas que deben pasar siempre
- @regression: Suite completa de regresión
- @wip: Work-in-progress, escenarios en desarrollo

### Por componente:

- @login: Funcionalidad de autenticación
- @cart: Carrito de compras
- @api: Pruebas de servicios REST

### Por organización:

- @sprint\_12: Funcionalidades del sprint actual
- @checkout\_epic: Relacionados con una épica específica

## Ejemplo con tags:

```
None
@smoke @login
Feature: Autenticación de usuarios

  @positive
  Scenario: Login exitoso
    Given estoy en la página de login
    When ingreso credenciales válidas
    Then accedo al sistema correctamente

  @negative @wip
  Scenario: Login con credenciales inválidas
    Given estoy en la página de login
    When ingreso credenciales incorrectas
    Then veo un mensaje de error
```

## ¿Cómo usarlos?

Comando	Qué hace
<code>behave -t @smoke</code>	Ejecuta solo los escenarios etiquetados <code>@smoke</code>
<code>behave -t ~@wip</code>	Ejecuta todo menos los que estén en progreso
<code>behave -t @api,@ui</code>	Ejecuta los que tengan al menos uno de esos tags

**Ejemplo real:** en CI corremos solo `@smoke`; en la regresión nightly usamos `@regression`.

## Estrategia real de tags:

- **En CI:** Corremos solo `@smoke` para feedback rápido
- **Regresión nocturna:** Usamos `@regression` para cobertura completa
- **Durante desarrollo:** Excluimos `@wip` para evitar fallos conocidos
- **Por componente:** `@api` para pruebas de servicios, `@ui` para interfaz

Los tags transforman tu suite de pruebas en una herramienta flexible que se adapta a diferentes necesidades y momentos del ciclo de desarrollo.

# ¿Dónde vive Behave en tu proyecto?

Ahora que ya sabés escribir escenarios con Gherkin y organizarlos con tags, es momento de ver **cómo se integra todo esto dentro de tu proyecto**. En esta sección vamos a explorar **la estructura recomendada para trabajar con Behave de forma ordenada y mantenible**.

La idea es separar claramente las distintas responsabilidades:

- por un lado, los archivos **.feature**, legibles para cualquier miembro del equipo,
- y por otro, los conectores Python que ejecutan la lógica de automatización.

La organización del código BDD sigue una estructura estándar que facilita la colaboración y el mantenimiento. **Tener todo BDD bajo **features/** evita colisiones y facilita a cualquier lector encontrar las historias**.

## Estructura recomendada

```
proyecto/
├─ features/
│   ├─ steps/
│   │   ├─ login_steps.py
│   │   └─ cart_steps.py
│   └─ environment.py
│   └─ login.feature
│   └─ cart.feature
├─ pages/          # Tus Page Objects existentes
├─ tests/          # Tests de Pytest
└─ reports/        # Reportes HTML y capturas
```

## ¿Por qué esta estructura?

Separación clara de responsabilidades

- **features/** → contiene solo las historias (**.feature**) que pueden ser leídas por cualquier persona del equipo.
- **steps/** → contiene el código Python que conecta cada línea Gherkin con la lógica real (step definitions).
- **environment.py** → se encarga de configurar lo que ocurre antes, durante y después de ejecutar los escenarios.
- **pages/** → guarda tus Page Objects reutilizables (si hacés pruebas UI).
- **reports/** → guarda los resultados de la ejecución (HTML, JSON, screenshots...).

## Convención sobre configuración

- **Behave busca automáticamente** la carpeta `features/` en el directorio actual
- **Los step definitions** se cargan desde `features/steps/`
- `environment.py` se ejecuta automáticamente para configurar hooks

## Comandos desde diferentes ubicaciones

Shell

*# Desde la raíz del proyecto*

`behave` *# Ejecuta toda la suite*

`behave features/login.feature` *# Ejecuta solo login*

*# Desde cualquier subcarpeta*

`behave ../features/` *# Behave encuentra automáticamente*

*# Con filtros*

`behave -t @smoke` *# Solo pruebas críticas*

`behave features/ecommerce/` *# Solo módulo de e-commerce*

Esta estructura no es solo organización; es una **convención que facilita la colaboración** entre roles técnicos y no técnicos, manteniendo el código limpio y las historias accesibles para todos.

# Instalación de Behave

Ya tenemos la estructura clara y los conceptos fundamentales del lenguaje Gherkin. El siguiente paso es preparar el entorno para **ejecutar nuestras pruebas vivas con Behave**.

Behave es una herramienta de línea de comandos que interpreta archivos `.feature` y ejecuta los steps asociados escritos en Python. Para que todo funcione correctamente, vamos a instalar **Behave** junto con un formateador visual que nos permita generar reportes en HTML.

## 1. Instalar dependencias

Desde la raíz de tu proyecto, ejecutá:

```
pip install behave behave-html-formatter # librería y reporte HTML genérico
```

Esto instala:

- behave: el motor principal que corre las pruebas BDD
- behave-html-formatter: herramienta para generar reportes HTML autocontenidos (colores, tablas, capturas)

👉 Tip: no te olvides de agregar estas dependencias al requirements.txt si estás trabajando en equipo o con entornos virtuales.

## 2. Crear carpetas necesarias

También vamos a crear la estructura mínima de carpetas donde vivirá nuestra suite:

```
mkdir -p features/steps reports/screens # carpetas básicas
```

- `features/`: donde irán los archivos `.feature`
- `steps/`: donde escribiremos los archivos `.py` que contienen la lógica
- `reports/screens`: donde se guardarán las capturas automáticas cuando haya fallos

Ejecuta estos comandos en la raíz del proyecto (donde vive tu `requirements.txt`). VS Code detectará las carpetas y te mostrará los `.feature` con coloreado de sintaxis.



### 3. Crear behave.ini

Este archivo opcional pero recomendable permite definir configuraciones por defecto para Behave. Crea en la raíz del proyecto:

En la misma raíz del repo, crea un archivo de texto llamado `behave.ini`.

```
[behave]
lang = es    # Palabras clave en español (Escenario, Dado, Cuando...)
default_tags = ~@wip  # Excluye en progreso por defecto
```

Behave busca automáticamente este archivo desde la carpeta actual hacia arriba, así que tenerlo al lado de `pytest.ini` lo hace global para todo el equipo.

#### ¿Por qué esta configuración es útil?

- Establece el idioma para que podamos usar Gherkin en español
- Permite **excluir automáticamente** los tests en progreso (`@wip`)
- Evita errores por configuración inconsistente entre integrantes del equipo

# Tres ejemplos completos

Para apreciar la versatilidad de BDD vamos a escribir 3 features que demuestran cómo la misma sintaxis Gherkin sirve para diferentes capas de testing. Empezamos con **lógica pura de Python** (lo más sencillo), continuamos con **interfaz web de SauceDemo** y terminamos con **API de JSONPlaceholder**. Así verás que BDD no se limita solo a UI.

## 1. Lógica pura: Operaciones matemáticas

Imagina que el equipo de Backend nos pidió validar un módulo de utilidades matemáticas antes de integrarlo al microservicio de cálculo. Empezamos con la función más simple.

### 1. Código a testear

utils/operaciones.py

```
Python
"""Operaciones matemáticas simples utilizadas por el
micro-servicio de cálculo."""

def multiplicar(a: int, b: int) -> int:
    """Devuelve el producto de dos enteros y registra la
operación."""
    return a * b
```

### 2. Feature: operaciones.feature

```
None
@unit @smoke
Feature: Multiplicación de enteros
    Para asegurar consistencia en el micro-servicio de cálculo
    Como desarrollador del equipo Backend
    Quiero verificar que la función multiplicar devuelve el
    producto correcto

    Scenario Outline: Producto <a> x <b>
        When multiplico <a> y <b>
```

Then obtengo <resultado>

Examples:

a	b	resultado
2	3	6
5	0	0
7	4	28
-3	2	-6

### 3. Steps: operaciones\_steps.py

Python

```
from behave import when, then
from utils.operaciones import multiplicar

@when('multiplico {a:d} y {b:d}')
def step_multiplicar(context, a, b):
    """Ejecuta la multiplicación y guarda el resultado en
    context."""
    context.resultado = multiplicar(a, b)

@then('obtengo {esperado:d}')
def step_validar(context, esperado):
    """Verifica que el resultado coincide con lo esperado."""
    assert context.resultado == esperado, (
        f"Esperaba {esperado}, obtuve {context.resultado}"
    )
```

**¿Qué logramos?** Convertimos una función Python simple en una especificación ejecutable que cualquier miembro del equipo puede leer y entender. El **Scenario Outline** nos permite probar múltiples casos con una sola definición.

## 2. UI: Login en SauceDemo

Ahora damos el salto al navegador. Queremos demostrar que un usuario válido puede iniciar sesión y acceder al inventario, reutilizando nuestros Page Objects.

### 1. Feature: login.feature

None

@ui @smoke

Feature: Login en SauceDemo

Como usuario registrado

Quiero poder autenticarme con credenciales válidas

Para acceder al inventario de productos

Background:

Given el navegador abre la página de login

Scenario: Credenciales válidas

When ingreso usuario "standard\_user" y clave "secret\_sauce"

Then la URL contiene "inventory.html"

And veo el título "Products"

Scenario: Credenciales inválidas

When ingreso usuario "invalid\_user" y clave "wrong\_password"

Then veo el mensaje de error "Username and password do not match"

### 5.2.2 Steps clave explicados

Python

```
from behave import given, when, then
```

```
from pages.login_page import LoginPage
```

```
from pages.inventory_page import InventoryPage
```

```
@given('el navegador abre la página de login')
```

```
def step_open_login(context):
```

```
    """Abre la página de login usando nuestro Page Object."""
```

```
    context.login_page = LoginPage(context.driver)
```

```

context.login_page.abrir()

@when('ingreso usuario "{username}" y clave "{password}"')
def step_login(context, username, password):
    """Realiza el proceso de login."""
    context.login_page.ingresar_credenciales(username,
password)
    context.login_page.hacer_click_login()

@then('la URL contiene "{texto}"')
def step_verify_url(context, texto):
    """Verifica que la URL actual contiene el texto
esperado."""
    assert texto in context.driver.current_url, (
        f"URL actual: {context.driver.current_url}"
    )

```

**¿Qué ocurre?** El `LoginPage` encapsula los selectores CSS; el step solo narra la intención del usuario. Esto mantiene el feature legible y el código mantenible. El `Background` evita repetir la preparación en cada escenario.

### 3. API: Gestión de posts en JSONPlaceholder

Por último validamos que nuestro framework puede interactuar con servicios REST, algo fundamental para testing integral.

#### 1. Feature: post\_api.feature

None

@api

Feature: Posts JSONPlaceholder

Como desarrollador de la API

Quiero verificar que los endpoints de posts funcionan correctamente

Para asegurar la integridad del servicio

Scenario: Crear y eliminar post

When creo un post con title "BDD Test" y body "Contenido de prueba"

Then la respuesta POST es 201 y contiene id

When elimino el post recién creado

Then la respuesta DELETE es 200

Scenario: Obtener post existente

When consulto el post con id 1

Then la respuesta GET es 200

And el title contiene texto

#### 2. Steps destacados con Requests

Python

```
import requests
```

```
from behave import when, then
```

```
BASE_URL = 'https://jsonplaceholder.typicode.com'
```

```
@when('creo un post con title "{title}" y body "{body}"')
```

```
def step_create_post(context, title, body):
```

```
    """Crea un nuevo post y guarda la respuesta."""
```

```
    response = requests.post(f'{BASE_URL}/posts', json={
        'title': title,
```



```

        'body': body,
        'userId': 1
    })
    context.response = response
    context.post_id = response.json()['id']

@then('la respuesta POST es {status_code:d} y contiene id')
def step_verify_post_creation(context, status_code):
    """Verifica que el post se creó correctamente."""
    assert context.response.status_code == status_code
    assert 'id' in context.response.json()
    assert context.post_id > 0

@when('elimino el post recién creado')
def step_delete_post(context):
    """Elimina el post usando el ID guardado."""
    response =
requests.delete(f'{BASE_URL}/posts/{context.post_id}')
    context.response = response

```

¿Qué validamos? Que nuestra capa de API funciona correctamente y que los steps pueden compartir datos a través de `context` para pasos posteriores. El objeto `context` actúa como memoria compartida entre pasos del mismo escenario.

## Resumen de los tres enfoques

Tipo	Qué prueba	Herramientas	Ventaja clave
Lógica pura	Funciones Python	Imports directos	Rapidez y simplicidad
UI	Interfaz web	Selenium + Page Objects	Pruebas end-to-end realistas
API	Servicios REST	Requests + JSON	Validación de contratos

**El poder de BDD:** La misma sintaxis Gherkin sirve para todas las capas, creando un lenguaje común entre desarrolladores, testers y stakeholders, sin importar la complejidad técnica subyacente.

# Hooks globales: el backstage (y ejemplos concretos)

Hasta ahora trabajamos en el “escenario principal”: escribimos features legibles, conectamos cada paso a código real, y estructuramos nuestro proyecto. Pero, como en toda buena obra, hay mucho que sucede **detrás de escena** para que la función salga bien.

En Behave, eso que sucede “detrás del telón” se programa usando los **hooks globales**, funciones especiales que se ejecutan **antes o después** de ciertos momentos clave durante la ejecución de las pruebas.

Estos hooks permiten:

- **Preparar el entorno** antes de que comience la ejecución (abrir navegador, configurar conexión...)
- **Capturar evidencias** si ocurre un fallo (como una screenshot)
- **Limpiar recursos** al finalizar (cerrar navegador, borrar datos, liberar memoria)

## ¿Dónde se escriben los hooks?

Todos los hooks se definen en un archivo especial:

**features/environment.py**

Este archivo actúa como el director técnico de tu suite: no define los pasos visibles de los actores, pero sí controla luces, escenografía y efectos especiales que afectan cómo se ejecuta cada escenario.

## ¿Cuáles son los hooks más comunes?

Hook	¿Cuándo se ejecuta?	¿Para qué sirve?
<b>before_all()</b>	Antes de que comience toda la suite	Inicializar navegador o recursos globales
<b>before_scenario()</b>	Antes de cada escenario individual	Registrar logs, preparar datos
<b>after_step()</b>	Después de cada paso	Tomar screenshot si el paso falla
<b>after_all()</b>	Al finalizar toda la ejecución	Cerrar navegador, limpiar recursos

## ¿Por qué son importantes?

Sin estos hooks, tendrías que repetir lógica técnica (como abrir el navegador) en cada step, haciendo el código menos mantenible y más propenso a errores. Usar hooks te permite:

- Separar la lógica de negocio del manejo técnico
- Centralizar configuraciones que deben ejecutarse siempre
- Automatizar tareas clave sin contaminar los steps

## Hooks más comunes

### before\_all

Aquí inicializamos recursos costosos como el navegador web que serán compartidos por todos los escenarios.

**Se ejecuta una sola vez antes de la primera línea Gherkin.**

```
Python
from selenium import webdriver
# --- HOOKS ---
def before_all(context):
    """Arranca el navegador y pone tiempo de espera
    implícito."""
    context.driver = webdriver.Chrome()
    context.driver.implicitly_wait(5) # evita sleeps fijos
```

**¿Por qué aquí?** — Para no abrir un navegador por cada escenario y ahorrar minutos de ejecución.

## before\_scenario

Usamos este hook para registrar el inicio de cada escenario y preparar el estado inicial si es necesario.

**Sirve para anotar en el log o resetear datos entre escenarios.**

```
Python
import logging
logger = logging.getLogger('talentolab')

def before_scenario(context, scenario):
    logger.info('▶ Iniciando escenario: %s', scenario.name)
```

## after\_step

Captura automáticamente una screenshot cuando cualquier step falla, proporcionando evidencia visual del error sin modificar el código de los steps.

```
Python
import pathlib
from datetime import datetime

SCREEN_DIR = pathlib.Path('reports/screens')
SCREEN_DIR.mkdir(parents=True, exist_ok=True)

def after_step(context, step):
    if step.status == 'failed' and hasattr(context, 'driver'):
        timestamp = datetime.utcnow().strftime('%H%M%S')
        file = SCREEN_DIR / f"{step.name}_{timestamp}.png"
        context.driver.save_screenshot(str(file))
```

**Resultado:** si un paso falla, el reporte HTML de Behave mostrará la imagen; en CI la captura queda como artefacto.

## after\_all

Limpia y libera todos los recursos al finalizar la ejecución completa de la suite, evitando procesos huérfanos.

Python

```
def after_all(context):  
  
    if hasattr(context, 'driver'):  
  
        context.driver.quit()
```

Cierra el navegador y libera memoria; crucial para que la máquina de CI no se quede con procesos colgados.

## Ejecutar y leer resultados — paso a paso

Ya tenemos nuestras pruebas escritas, los steps implementados, y los hooks funcionando. Es momento de ver todo **en acción**: cómo ejecutamos las historias BDD, **cómo se visualizan los resultados**, y qué tipo de reportes podemos generar para compartir con el equipo o documentar nuestra entrega final.


Esta sección te muestra **dos flujos habituales de ejecución**, según el objetivo que tengas:

### 1. Ejecución rápida (smoke)

Shell

```
behave -t @smoke -f pretty
```

- **-t @smoke** filtra solo los escenarios marcados con ese tag. Ideal para un chequeo express antes de commitear
- **-f pretty** es el formatter por defecto: muestra cada paso en consola con ✓ o ✗ y colorea los mensajes

 Ideal para: revisar antes de hacer un commit o compartir tu trabajo.

### Ejemplo de salida compacta:

None

```
Login en SauceDemo -- @smoke @ui
```

```
Escenario: Credenciales válidas ✓
```

```
1 feature passed, 0 failed, 0 skipped - 0.84s
```



## 2. Suite completa + artefactos

Cuando necesitas dejar **evidencia para un cliente, docente o el pipeline de CI**, es importante generar reportes persistentes en **formato HTML o JSON**.

Shell

```
behave \  
  -f json -o reports/behave.json \  
  -f html -o reports/behave.html \  
  -f pretty
```

- **-f json** genera un fichero máquina-legible (behave.json) que el pipeline podrá combinar con otros reportes
- **-f html** crea un documento auto-contenido (behave.html) con colores y tablas
- **-f pretty** mantiene la salida en consola para que veas el progreso en tiempo real

🎯 Con esta ejecución, todo queda guardado en la carpeta **/reports**: tanto los logs como los screenshots, si se produjeron errores.

## Leer el resultado

**Consola:** al final verás un resumen similar a:

None

3 features passed, 0 failed, 6 scenarios passed in 4.12s

1. **HTML:** abre **reports/behave.html** con tu navegador o la extensión Live Server en VS Code.
  - Cada fila de la tabla indica Feature → Scenario → Step con fondo verde o rojo
  - Al hacer clic sobre un step rojo, se expande la sección y—gracias al hook **after\_step**—muestra la screenshot capturada
2. **JSON:** útil para integrarlo con Allure, cucumber-reports o para parsearlo y generar métricas

**@smoke + pretty** para feedback instantáneo; **HTML + JSON** para evidencias que viajarán al artefacto de CI.

# Integrar a Pytest: una sinfonía única

La verdadera potencia surge cuando combinamos Pytest y Behave en un solo pipeline de ejecución. **Creamos un wrapper que permite ejecutar las historias BDD como si fueran tests unitarios regulares, unificando reportes y simplificando la experiencia del desarrollador.**

**Creamos un test wrapper:**

```
Python
import subprocess
import pathlib

def test_behave_suite():
    """Ejecuta la suite de Behave desde Pytest"""
    result = subprocess.run([
        'behave', '-q', '-f', 'json', '-o',
        'reports/behave.json'
    ])
    assert result.returncode == 0, "La suite de Behave falló"
```

**¿Qué ganamos?** — Ahora `pytest -v` correrá sus propias pruebas unitarias y las historias Behave en secuencia; el reporte HTML final mostrará todo junto, y los desarrolladores solo necesitan recordar un comando para ejecutar toda la suite de calidad.

# Creanto el framework de TalentoLab – Ticket Jira QA-158



**Silvia** nos envía un mensaje por Slack:

"Necesito los tests de **ver login** y **añadir al carrito** escritos en Gherkin para la demo con el cliente. El Ticket QA-158 ya fue creado y te lo asignamos."



**Matías** abre la videollamada y detalla:

"Vas a escribir dos archivos **.feature** y sus steps. Usa los Page Objects que ya hiciste, captura screenshots en fallos y sube el reporte HTML. Todo esto va directo al framework final, así que cuida los nombres de tags: **@smoke** para login y **@regression** para el carrito."

## Objetivo específico

Crear una suite BDD completa que permita a stakeholders no técnicos entender qué está siendo probado, mientras que los desarrolladores pueden ejecutar y mantener los tests automáticamente.

## Desglose detallado de tareas

### Tarea 1: **login.feature** - Escenarios de autenticación

- Ubicación: **features/login.feature**
- Tags requeridos: **@ui**, **@smoke** para el escenario principal
- Contenido mínimo:
  - Un Background que abra la página de login
  - Un Scenario exitoso con credenciales válidas (**standard\_user** / **secret\_sauce**)
  - Un Scenario Outline con al menos 3 casos de error (credenciales inválidas, usuario bloqueado, campos vacíos)
- Validaciones: Redirección exitosa al inventario vs. mensaje de error visible

### Tarea 2: **cart.feature** - Funcionalidad de carrito

- Ubicación: **features/cart.feature**

- Tags requeridos: `@ui`, `@regression`
- Contenido mínimo:
  - Background que haga login automático con `standard_user`
  - Scenario principal: agregar "Sauce Labs Backpack" y verificar contador en "1"
  - Extra opcional: Scenario con múltiples productos
- Validaciones: Contador del carrito se incrementa correctamente

### Tarea 3: Step definitions - Conectores Python

- Ubicación: `features/steps/login_steps.py` y `features/steps/cart_steps.py`
- Requisito clave: Reutilizar las clases `LoginPage` e `InventoryPage` que ya creaste
- Patrón: Cada step debe importar y usar los métodos de Page Object Model
- Logging: Incluir logs informativos usando `logger.info()` en cada step

### Tarea 4: Environment.py - Configuración global

- Ubicación: `features/environment.py`
- Hooks requeridos:
  - `before_all()`: Configurar WebDriver una vez
  - `after_step()`: Capturar screenshot automático en fallos
  - `after_all()`: Cerrar WebDriver y limpiar recursos
- Screenshots: Guardar en `reports/screens/` con nombres descriptivos

### Tarea 5: Integración con Pytest

- Ubicación: `tests_behave/test_behave_suite.py`
- Función: Wrapper que ejecute Behave desde Pytest usando `subprocess.run()`
- Formatos: Generar tanto JSON como reportes pretty para diferentes audiencias
- Filtros: Tests separados para `@smoke` y `@regression`

### Tarea 6: Configuración y reportes

- `behave.ini`: Configurar idioma, tags por defecto y rutas
- `pytest.ini`: Actualizar para incluir markers BDD
- Reportes: Ejecutar y verificar que se generan `behave.json` y reportes HTML

## Checklist de entrega

Antes de hacer commit, verifica que tienes:

- `features/login.feature` con al menos 4 escenarios (1 exitoso + 3 de error)
- `features/cart.feature` con Background de login y escenario de "Sauce Labs Backpack"
- Steps que importan y usan `LoginPage` e `InventoryPage`
- Hook de screenshots funcionando (prueba haciendo fallar un test)
- Wrapper de Pytest que ejecuta Behave y genera reportes

- Comando `behave -t @smoke` funciona sin errores
- Comando `pytest tests_behave/` ejecuta la suite BDD desde Pytest

## Comandos de verificación

```
# Verificar que Behave reconoce los features
behave --dry-run
# Ejecutar solo smoke tests
behave -t @smoke
# Ejecutar desde Pytest
pytest tests_behave/ -v
# Generar reportes completos
behave -f json -o reports/behave.json -f pretty
```

💡 Este trabajo nutre directamente la sección BDD del Framework Final.

## Conexión con el proyecto final

Los archivos `.feature` que escribas hoy serán fundamentales para tu entrega final porque:

1. **Demuestran comunicación clara:** Los stakeholders pueden leer y entender las pruebas sin conocimiento técnico
2. **Documentación viva:** Las features sirven como especificaciones ejecutables que siempre están actualizadas
3. **Cobertura completa:** BDD + Pytest + API tests muestran un framework robusto y profesional
4. **Preparación para CI/CD:** Los reportes de Behave se integrarán perfectamente en el pipeline final

## Próximos pasos

En la **Clase 15** conectaremos todo a GitHub Actions: instalaremos dependencias, correremos Pytest + Behave y subiremos automáticamente los reportes y las capturas para que Silvia los revise sin salir de la plataforma.

¡Tu framework está casi completo! 🎯



**Buenos Aires**  
*aprende*  
Agencia de Políticas para el Futuro

**BA** Buenos  
Aires  
Ciudad