

«Talento Tech»

Automation Testing

Clase 15



Clase N.º 15 | Integración Continua (CI/CD) con GitHub Actions

Temario

- Introducción y contexto
- Objetivos de aprendizaje
- Conceptos clave: CI, CD y sus diferencias
- GitHub Actions: runners, eventos, jobs, steps y YAML
- Anatomía de un workflow YAML
- Paso a paso – crear nuestro primer pipeline
- Lectura de resultados
- Buenas prácticas y problemas típicos
- Ejercicio práctico TalentoLab
- Relación con la Entrega Final
- Recursos adicionales

Objetivos de la clase

En esta clase aprenderemos los fundamentos de la Integración Continua (CI) y el Despliegue Continuo (CD), entendiendo sus diferencias y su importancia en proyectos de desarrollo. Exploraremos cómo funciona GitHub Actions, con foco en sus componentes clave: runners, eventos, jobs, steps y la estructura de archivos YAML. A partir de esto, crearemos paso a paso nuestro primer pipeline, aprenderemos a leer sus resultados y reconoceremos buenas prácticas y errores comunes. El ejercicio práctico estará centrado en un caso real de TalentoLab, conectando lo aprendido con los requisitos de la entrega final. Finalmente, contaremos con recursos adicionales para seguir profundizando en el tema.

Introducción

Hasta hoy construimos pruebas UI y API, organizamos el código con Page Object Model, generamos reportes y logs, e incluso escribimos requisitos ejecutables con Behave.

Ahora, el paso lógico es automatizar la ejecución cada vez que el equipo modifica el repositorio. Para eso usaremos GitHub Actions, el motor de CI/CD nativo de GitHub.

¿Cómo funciona la automatización con GitHub Actions?

Imagina este escenario: un desarrollador hace cambios en el código y crea un Pull Request. En lugar de esperar a que alguien ejecute las pruebas manualmente, GitHub Actions detecta automáticamente este evento y:

1. **Se dispara el pipeline** → Al hacer push o crear un PR hacia main/develop
2. **Ejecuta todas las pruebas** → UI, API y BDD sin intervención humana
3. **Valida la calidad** → Si algún test falla, el PR se marca como "no aprobado"
4. **Protege el código principal** → Evita que código defectuoso llegue a producción

¿Por qué GitHub Actions?

GitHub Actions es gratuito para repositorios públicos y ofrece 2,000 minutos mensuales para repositorios privados. Nos permite ejecutar nuestros tests automáticamente en la nube sin configurar servidores propios, garantizando que cada cambio sea validado antes de integrarse al proyecto principal.

Requisito: tener una cuenta gratuita en github.com y subir tu proyecto como repositorio.





Integración Continua (CI) y Entrega Continua (CD)

¿Qué es Integración Continua (CI)?

Proceso mediante el cual cada cambio en el código se compila, prueba y valida de forma automática. El objetivo es detectar errores cuanto antes y garantizar que la base de código se mantenga siempre en un estado deployable.

Ejemplo de CI en acción:

- María modifica el código de login en SauceDemo y hace push
- GitHub Actions se dispara automáticamente
- Ejecuta todas las pruebas: UI (login, checkout), API (usuarios, productos) y BDD
- Si algún test falla → el commit se marca como  y María recibe notificación inmediata
- Si todos pasan → el código se integra exitosamente 

¿Qué es Entrega Continua (CD)?

Práctica que toma lo generado por la CI y crea artefactos listos para desplegar en cualquier momento. Puede, o no, incluir el despliegue automático. El foco está en la rapidez y confiabilidad para poner el software en manos de los usuarios.

Ejemplo de CD en acción:

- Después de que la CI de María pasa exitosamente
- El pipeline genera automáticamente: reportes HTML, documentación actualizada, package deployable
- Estos artefactos se suben a un servidor de staging o se preparan para producción
- Con un clic, el equipo puede desplegar la nueva versión en cualquier momento

Diferencias y relación CI vs CD

Fase	¿Qué hace?	Cuándo falla	Valor que aporta
CI	Compila y ejecuta tests	Cuando las pruebas no pasan o el código no compila	Feedback inmediato al desarrollador; evita merge de código roto
CD	Empaqueta y (opcionalmente) despliega	Cuando no puede generar un artefacto o el despliegue se interrumpe	Reduce el tiempo de llegada a producción y acelera el time-to-market

¿Qué es GitHub y cómo cargar un proyecto?

GitHub es una plataforma que permite almacenar y gestionar proyectos de software utilizando el sistema de control de versiones **Git**. Además de facilitar el trabajo colaborativo entre desarrolladores, GitHub permite automatizar tareas como pruebas y despliegues mediante GitHub Actions, que es el foco principal de esta clase.

Pero antes de configurar pipelines, es importante asegurarnos de que el proyecto esté correctamente versionado y subido a GitHub. A continuación, te mostramos los **comandos base** que necesitas conocer para iniciar tu repositorio y cargar tu proyecto:



Comandos básicos para subir un proyecto a GitHub

1. Inicializar un repositorio Git en tu carpeta de proyecto

```
git init
```

2. Agregar todos los archivos al repositorio

```
git add .
```

3. Crear el primer commit (registro de cambios)

```
git commit -m "Primer commit - Subida de proyecto base"
```

4. Crear la rama principal (**main**)

```
git branch -M main
```

5. Conectar tu repositorio local con uno remoto en GitHub

Previamente debes haber creado un repositorio vacío en [GitHub.com](https://github.com)

```
git remote add origin https://github.com/TU_USUARIO/NOMBRE_REPOSITORIO.git
```

6. Subir tu proyecto a GitHub

```
git push -u origin main
```

✓ **Tip útil:** Recordá que si trabajás con un equipo, GitHub también te permite colaborar a través de *Pull Requests*, comentarios y revisiones de código.

GitHub Actions

Ahora que entendemos qué es CI/CD, necesitamos conocer cómo GitHub Actions implementa estos conceptos. GitHub Actions funciona con cuatro componentes clave que trabajan juntos para ejecutar nuestro pipeline de pruebas automáticas.

Runners

Un runner es la máquina virtual que GitHub proporciona para ejecutar tu pipeline. Piénsalo como una computadora limpia en la nube que se enciende cada vez que necesitas correr tus tests.

Tipos de runners disponibles:

- **Runners hospedados por GitHub** (lo que usaremos):
 - `ubuntu-latest` → Máquina Linux (más rápida y económica)
 - `windows-latest` → Máquina Windows
 - `macos-latest` → Máquina macOS
- **Runners auto-gestionados:** Tu propia máquina configurada como runner (para casos avanzados)

¿Por qué ubuntu-latest es nuestra mejor opción?

- **Velocidad:** Inicia más rápido que Windows/macOS
- **Costo:** Consume menos minutos de tu cuota mensual
- **Compatibilidad:** Python, Selenium y nuestras dependencias funcionan perfectamente
- **Herramientas preinstaladas:** Git, Docker, navegadores web ya están listos

Eventos

Los eventos son los "disparadores" que inician tu pipeline. Definen **cuándo** se ejecutará la CI/CD.

Eventos más comunes:

- `push` → Cada vez que subas código al repositorio
- `pull_request` → Cuando alguien crea o actualiza un PR
- `schedule` → Ejecución programada (ej: todas las noches a las 2 AM)
- `workflow_dispatch` → Ejecución manual desde la interfaz de GitHub

En nuestro framework usaremos:

- `push` y `pull_request` para **CI** → validar cambios inmediatamente
- `schedule` para **regresiones nocturnas** → ejecutar todos los tests diariamente

Jobs y Steps

Job: Es una unidad de trabajo completa que corre en un runner aislado. Puede haber múltiples jobs ejecutándose en paralelo.

Step: Es un comando individual dentro de un job, como "instalar dependencias" o "ejecutar tests".

Ejemplo de estructura:

```
None
jobs:

  test:          # Job principal

    runs-on: ubuntu-latest

    steps:

      - name: Checkout código   # Step 1

      - name: Instalar Python   # Step 2

      - name: Ejecutar tests    # Step 3
```

Los steps pueden usar **acciones del Marketplace** (como [actions/checkout@v4](#)) o comandos personalizados (`pip install -r requirements.txt`).

Variables y secretos

Variables de entorno ([env](#)): Definen configuración accesible en todos los steps

```
None
env:

  PYTHON_VERSION: '3.12'

  BROWSER: 'chrome'
```

Secrets: Información sensible (tokens, passwords) que se configura en Settings → Secrets → Actions

None

steps:

- name: Deploy

env:

API_TOKEN: \${{ secrets.DEPLOY_TOKEN }}

Estos cuatro componentes trabajan juntos para crear nuestro pipeline de CI que detectará problemas automáticamente cada vez que modifiquemos el código.

Anatomía de un archivo YAML de workflow

Ahora que conocemos los cuatro componentes de GitHub Actions (runners, eventos, jobs/steps, variables), vamos a ver cómo se plasman en la práctica dentro de un archivo YAML. Este será el "esqueleto" de nuestro pipeline:

```
None
name: CI TalentoLab           # Nombre del workflow

on: [push, pull_request]     # Eventos que lo disparan

jobs:

  tests:                     # Nombre del job

    runs-on: ubuntu-latest    # Runner que utilizaremos

    steps:

      - name: Checkout código  # Step descriptivo

        uses: actions/checkout@v4  # Acción del Marketplace
```

Identificando los componentes del punto 4:

- **Eventos** → `on: [push, pull_request]` define cuándo se ejecuta
- **Runner** → `runs-on: ubuntu-latest` especifica la máquina virtual
- **Job** → `tests:` es nuestra unidad de trabajo principal
- **Steps** → Cada `-name:` representa un comando individual

Este ejemplo básico muestra la estructura mínima, pero en el punto 6 lo expandiremos para incluir instalación de dependencias, ejecución de tests y generación de reportes.

Reglas básicas de YAML

- Indentación con dos espacios, nunca tabs
- Dos puntos (:) separan clave y valor
- `-` indica un elemento de lista

Importante: YAML es muy estricto con la indentación. Un espacio mal colocado puede romper todo el pipeline, así que presta atención a los espacios cuando copies el código.

Paso a paso – crear nuestro primer pipeline

Perfecto. Ya entendemos la teoría y vimos la estructura básica. Ahora es momento de **pasar de la teoría a la práctica** y construir nuestro pipeline completo que integre todo el framework que hemos desarrollado a lo largo del curso.

En esta sección crearemos un pipeline mínimo viable que — al recibir un push o pull request a main/develop — instalará dependencias, ejecutará todos los tests UI y API sobre SauceDemo/JSONPlaceholder, y subirá los reportes como artefactos.

1. Preparación: Subir proyecto a GitHub

Antes de crear el pipeline, necesitas:

1. Crear repositorio en GitHub:

- Ve a github.com → "New repository"
- Nombre sugerido: talentolab-automation-framework
- Selecciona "Public" (para GitHub Actions gratuito)

Subir tu proyecto:

```
Shell
# En tu carpeta del proyecto
git init
git add .
git commit -m "Framework completo UI + API + BDD"
git branch -M main
git remote add origin https://github.com/TU_USUARIO/talentolab-automation-framework.
git push -u origin main
```

2. Estructura de carpetas

```
None
.
├── .github
│   └── workflows
│       └── ci.yml
```

Importante: La carpeta `.github/workflows/` debe estar en la raíz de tu repositorio, al mismo nivel que `pages/`, `tests/`, etc.

3. ci.yml explicado línea a línea

Aquí es donde **aplicamos todos los conceptos aprendidos** en un archivo YAML real. Cada sección corresponde a los componentes que estudiamos:

```
None

name: CI TalentoLab  # Nombre descriptivo del workflow
on:                  # EVENTOS: cuándo se ejecuta
  push:
    branches: [ main, develop ]
  pull_request:
    branches: [ main, develop ]
jobs:
  test:
    runs-on: ubuntu-latest  # JOB principal
                                # RUNNER: máquina Ubuntu

    env:
      PYTHON_VERSION: '3.12'  # VARIABLES de entorno
      PIP_CACHE: ~/.cache/pip

    steps:
      # STEPS: comandos individuales
      # 1 · Checkout
      - name: Checkout código
        uses: actions/checkout@v4  # Acción del Marketplace

      # 2 · Cache de pip
      - name: Cache dependencias
        uses: actions/cache@v4
        with:
          path: ${ env.PIP_CACHE }
          key:  ${ runner.os }}-pip-${
hashFiles('**/requirements.txt') }}

      # 3 · Setup Python
      - name: Instalar Python ${ env.PYTHON_VERSION }}
        uses: actions/setup-python@v5
        with:
          python-version: ${ env.PYTHON_VERSION }}
```

```

# 4 · Instalar dependencias
- name: Install dependencies
  run: pip install -r requirements.txt

# 5 · Ejecutar tests UI + API + BDD
- name: Run Pytest & Behave
  run: |
                                pytest tests/ tests_api/ -v
--html=reports/pytest_report.html --self-contained-html
    behave -f json -o reports/behave.json -f pretty

# 6 · Subir artefactos (reportes + logs)
- name: Upload HTML report
  uses: actions/upload-artifact@v4
  with:
    name: html-report
    path: reports/

- name: Upload Suite Log
  uses: actions/upload-artifact@v4
  with:
    name: suite-log
    path: logs/

```

¿Qué logramos? Un pipeline funcional que valida el proyecto en menos de 10 minutos y deja evidencia descargable, integrando **todos los elementos** construidos en las clases anteriores: Page Object Model, tests API, reportes HTML, logging y BDD.

Lectura de resultados

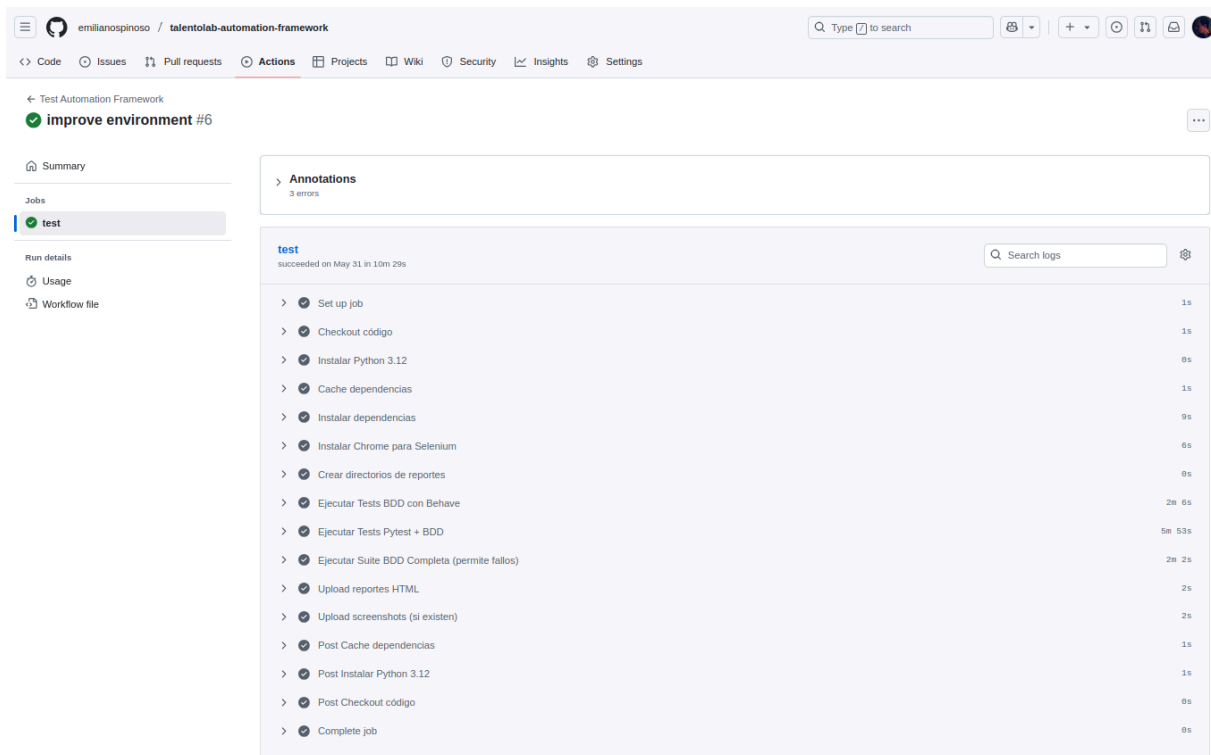
Perfecto, ya creamos nuestro pipeline en el punto 6. Ahora viene la parte crucial: **saber interpretar qué nos dice GitHub Actions** cuando se ejecuta. Después de hacer push, ¿dónde vemos si nuestros tests pasaron o fallaron?

¿Dónde encontrar los resultados?

Una vez que hagas push a tu repositorio, GitHub ejecutará automáticamente el workflow. Ahora te explicamos dónde ver cada cosa:

Paso 1: Ir a la pestaña Actions

1. En tu repositorio de GitHub, haz clic en **"Actions"** (junto a Code, Issues, Pull requests)
2. Verás una lista de todas las ejecuciones del workflow
3. Cada línea muestra: commit, rama, tiempo de ejecución y estado (✅ éxito o ❌ fallo)



Paso 2: Ver detalles de una ejecución









1. Haz clic en cualquier ejecución (ej: "CI TalentoLab")
2. Verás el **Summary** con:
 - Estado general del workflow
 - Lista de jobs ejecutados
 - Sección **"Artifacts"** con archivos descargables

Paso 3: Inspeccionar logs detallados





1. Haz clic en el job **"test"**
2. Se despliega cada step con sus logs:
 - **Checkout código** ✓
 - **Install dependencies** ✓
 - **Run Pytest & Behave** ✗ (si falla, aquí verás el error exacto)
3. Cada step es expandible para ver la salida completa

Paso 4: Descargar artefactos

1. En el Summary, busca la sección **"Artifacts"**
2. Encontrarás:
 - **html-report** → contiene pytest_report.html y behave.json
 - **suite-log** → contiene suite.log con logs detallados
3. Haz clic para descargar (archivo .zip)

Artifacts			
Produced during runtime			
Name	Size	Digest	
 screenshots	2.79 MB	sha256:7e92e7e98b58dc7be25d9161dba9edb7e414f5d6e674b0a8...	  
 test-reports	2.84 MB	sha256:ae4e74d945d97459b213cca96d126f3cc1b97412769a1889...	  

Interpretando el estado del pipeline

-  **Verde:** Todos los tests pasaron, el código está listo para integrar
-  **Rojo:** Al menos un test falló, revisar logs para encontrar el problema
-  **Amarillo:** Pipeline ejecutándose, esperar a que termine
-  **Gris:** Pipeline cancelado o no ejecutado

Badge en el README

Para mostrar el estado del pipeline directamente en tu repositorio, agrega este código a tu README.md:

markdown

None

```
![CI  
Status](https://github.com/TU_USUARIO/TU_REPOSITORIO/actions/workflows/ci.yml/badge.svg)
```

Tip importante: Reemplaza TU_USUARIO y TU_REPOSITORIO con los nombres reales de tu cuenta y proyecto.

Ahora que sabemos interpretar resultados, en el siguiente punto veremos las mejores prácticas para evitar problemas comunes y optimizar nuestro pipeline.

Buenas prácticas + problemas típicos

✓ Haz esto	✗ Evita esto
Fija python-version, evita la versión por defecto del runner	Usar sudo apt-get en runners hospedados (rompe cache)
Sube solo artefactos útiles (HTML, logs)	Versionar los artefactos en Git – ¡ocupan MB y cambian en cada run!
Usa --maxfails=1 para abortar rápido en smoke	Poner sleep en lugar de implicit_wait → pipelines lentos
Protege main con branch protection rules que exijan el check verde	Hacer push directo a main sin PR

Problemas comunes y soluciones

✗ Error: "No module named 'selenium'"

Solución: Verifica que tu `requirements.txt` esté en la raíz y contenga todas las dependencias:

```
selenium==4.15.2
pytest==7.4.3
pytest-html==4.1.1
requests==2.31.0
behave==1.2.6
```

✗ Error: "ChromeDriver not found"

Solución: Agrega este step antes de ejecutar tests:

```
- name: Install Chrome
  run: |
    sudo apt-get update
    sudo apt-get install -y google-chrome-stable
```

✗ Error: "No artifacts found"

Solución: Asegúrate de que las carpetas existen antes de subirlas:

```
- name: Create reports directory
  run: mkdir -p reports logs
```

✗ Tests pasan localmente pero fallan en CI

Causas comunes:

- Timeouts muy cortos para CI (usar `implicitly_wait(10)`)
- Rutas de archivos incorrectas (usar `pathlib` en lugar de rutas hardcoded)
- Tests que dependen de estado específico (hacer tests independientes)

CI/CD en TalentoLab



Durante la daily del equipo, **Silvia** plantea un problema:

"Necesitamos que nuestras pruebas corran solas cada vez que alguien sube código; de lo contrario no llegaremos a la demo del viernes. Debemos implementar un pipeline en GitHub Actions."

Ejercicio Práctico

Ticket QA-199 – Pipeline de pruebas automáticas

Descripción breve

Como Product Owner necesito que cada push al repositorio ejecute automáticamente todas las pruebas y adjunte el reporte, para garantizar calidad continua sin intervención manual.

Alcance

1. Crear `.github/workflows/ci.yml` basado en el ejemplo de la sección 6
2. Ejecutar pruebas UI + API + BDD del framework completo
3. Subir reportes HTML y logs como artefactos
4. Opcional: Si trabajás en equipo -> Configurar branch-protection: exigir que "CI TalentoLab" pase antes de mergear a main

Criterios de Aceptación

- [] El workflow se ejecuta en < 8 min
- [] Falla si cualquier test marcado `@smoke` falla
- [] Los artefactos están disponibles 90 días
- [] El badge cambia a rojo en caso de fallo
- [] Incluye tests de SauceDemo (UI) y JSONPlaceholder (API)

Conexión con clases anteriores

Este pipeline integrará todo lo construido:

- **Tests UI** (Clases 7-10): Page Object Model con SauceDemo
- **Tests API** (Clases 11-12): Requests con JSONPlaceholder
- **Reportes** (Clase 13): HTML reports y logging

- **BDD** (Clase 14): Features en Gherkin con Behave

Pasos específicos para completar el ejercicio



Matías se acerca y te guía con los pasos que debés realizar para completar la tarea de Jira.

Paso 1: Configurar el repositorio

1. Sube tu framework completo a GitHub (si no lo hiciste en la sección 6.1)
2. Verifica que tienes: `pages/`, `tests/`, `tests_api/`, `features/`, `requirements.txt`

Paso 2: Crear el workflow

1. Crea la carpeta `.github/workflows/` en la raíz
2. Crea el archivo `ci.yml` con el contenido de la sección 6.3

Haz commit y push:

```
git add .github/workflows/ci.yml
git commit -m "Add GitHub Actions CI pipeline"
git push
```

Paso 3: Verificar la primera ejecución

1. Ve a GitHub → tu repositorio → pestaña "Actions"
2. Deberías ver el workflow ejecutándose automáticamente
3. Espera 5-8 minutos a que termine

Paso 4: Configurar branch protection (opcional)

1. Ve a Settings → Branches → Add rule
2. Aplica a `main`, marca "Require status checks to pass"
3. Selecciona "CI TalentoLab" como check requerido

Si podés lograr realizar esta tarea: ¡Felicidades!:

- ✓ Se ejecuta localmente con Pytest + Behave
- ✓ Genera evidencia (reportes y logs)
- ✓ Corre de forma autónoma en GitHub con cada cambio de código
- ✓ Integra UI, API y BDD en un solo pipeline
- ✓ Produce artefactos profesionales para stakeholders

Con esto culmina el contenido del curso

En la próxima (y última) sesión presentarán el **Proyecto Final Integrador** que incluye:

- Framework completo UI + API + BDD
- Page Object Model bien estructurado
- Reportes HTML y logging centralizado
- Pipeline de CI/CD funcional
- Documentación profesional

Entrega Final de Proyecto

Framework de Automatización de Pruebas

El objetivo del Trabajo Final Integrador es que desarrolles un framework de automatización de pruebas completo, que combine todos los conocimientos adquiridos a lo largo del curso. Este proyecto consistirá en la creación de un framework de testing automatizado en Python, utilizando Selenium WebDriver para pruebas de UI, la biblioteca Requests para pruebas de API, y aplicando patrones de diseño como Page Object Model para estructurar el código de manera eficiente y mantenible. Tu proyecto deberá demostrar una sólida comprensión de las técnicas de automatización, incluir pruebas tanto de UI como de API, y generar reportes visuales que faciliten la interpretación de los resultados.

Si estuviste realizando los ejercicios prácticos obligatorios tendras casi cerrado este proyecto.

Requerimientos Específicos:

Tecnologías a Utilizar:

- Python como lenguaje de programación principal
- Pytest como framework de testing
- Selenium WebDriver para automatización de interfaces web
- Biblioteca Requests para pruebas de API
- Git para control de versiones
- GitHub como repositorio de código

Organización del Código:

- Estructura tu proyecto con una organización clara de directorios (pages, tests, utils, etc.)
- Implementa el patrón Page Object Model para las pruebas de UI
- Aplica buenas prácticas de programación y comentarios descriptivos
- Nombra variables, métodos y clases de manera significativa

Funcionalidades Específicas:

1. Pruebas de UI (Selenium WebDriver):

- **Automatización de Casos de Prueba:**
 - Implementa al menos 5 casos de prueba para un sitio web demo (puede ser saucedemo.com, automationpractice.com, o cualquier otro sitio de práctica)
 - Los casos de prueba deben cubrir flujos completos (ejemplo: login, navegación, búsqueda, añadir producto al carrito, checkout)

- Incluye al menos un escenario negativo (ejemplo: login con credenciales inválidas)
- **Implementación de Page Object Model:**
 - Crea clases para cada página que vayas a automatizar
 - Implementa métodos que representen acciones en cada página
 - Separa la lógica de las pruebas de la lógica de interacción con la página
- **Gestión de Capturas de Pantalla:**
 - Implementa una funcionalidad que capture screenshots automáticamente cuando una prueba falla
 - Almacena las capturas con nombres descriptivos que incluyan fecha/hora y nombre del test
- **Manejo de Datos de Prueba:**
 - Implementa alguna forma de parametrización para ejecutar las pruebas con diferentes conjuntos de datos
 - Utiliza fuentes externas (archivos CSV, JSON, etc.) para leer los datos de prueba

2. Pruebas de API (Requests):

- **Automatización de Endpoints:**
 - Implementa al menos 3 casos de prueba para una API pública (puede ser ReqRes, JSONPlaceholder, o cualquier otra API pública)
 - Cubre diferentes métodos HTTP (GET, POST, DELETE)
- **Validación de Respuestas:**
 - Verifica códigos de estado HTTP
 - Valida estructura y contenido de las respuestas JSON
 - Implementa assertions para diferentes escenarios (éxito, error, etc.)
- **Encadenamiento de Peticiones (Opcional):**
 - Implementa un flujo donde una petición dependa del resultado de otra (ejemplo: crear un recurso y luego obtenerlo)

3. Generación de Reportes:

- **Reportes HTML:**
 - Configura pytest para generar reportes HTML detallados
 - Los reportes deben mostrar claramente los tests ejecutados, su estado (pasado/fallado) y duración
 - Incluye capturas de pantalla en los reportes para las pruebas fallidas

- **Logging:**
 - Implementa un sistema de logging que registre pasos clave durante la ejecución
 - El log debe ser lo suficientemente detallado para facilitar la depuración

4. Integración con CI/CD (Opcional):

- Configura GitHub Actions para ejecutar tus pruebas automatizadas cuando se realice un push al repositorio
- Genera y almacena los reportes como artefactos de la ejecución

Control de Versiones y Documentación:

Repositorio en GitHub:

- Sube el proyecto a un repositorio en GitHub
- Mantén un historial de commits que documente el progreso del proyecto
- Usa ramas para desarrollar funcionalidades y luego fusiónalas con la rama principal

README.md:

- Incluye un archivo README.md que explique:
 - El propósito del proyecto
 - Las tecnologías utilizadas
 - La estructura del proyecto
 - Cómo instalar las dependencias
 - Cómo ejecutar las pruebas
 - Cómo interpretar los reportes generados

Funcionalidad Esperada:

- El framework debe ser capaz de ejecutar todas las pruebas de manera consistente
- Las pruebas deben ser independientes entre sí (la falla de una no debe afectar a las demás)
- Los reportes generados deben proporcionar información clara sobre los resultados
- La estructura del código debe facilitar la incorporación de nuevas pruebas

Formato de Entrega:

- El proyecto debe estar subido a un repositorio público en GitHub
- Comparte el enlace del repositorio en el aula virtual antes de la fecha límite

Nombre del Repositorio:

`proyecto-final-automation-testing-[nombre-apellido]`

Presentación:

- Prepara una presentación breve (5-10 minutos) donde demuestres la ejecución de tu framework
- La presentación debe incluir:
 - Explicación de la estructura del proyecto
 - Demostración de la ejecución de pruebas
 - Muestra de los reportes generados
 - Conclusiones y desafíos encontrados



Buenos Aires
aprende
Agencia de Actividades para el Futuro

BA Buenos
Aires
Ciudad