



Módulo 4: Automatización de pruebas

## Pruebas funcionales

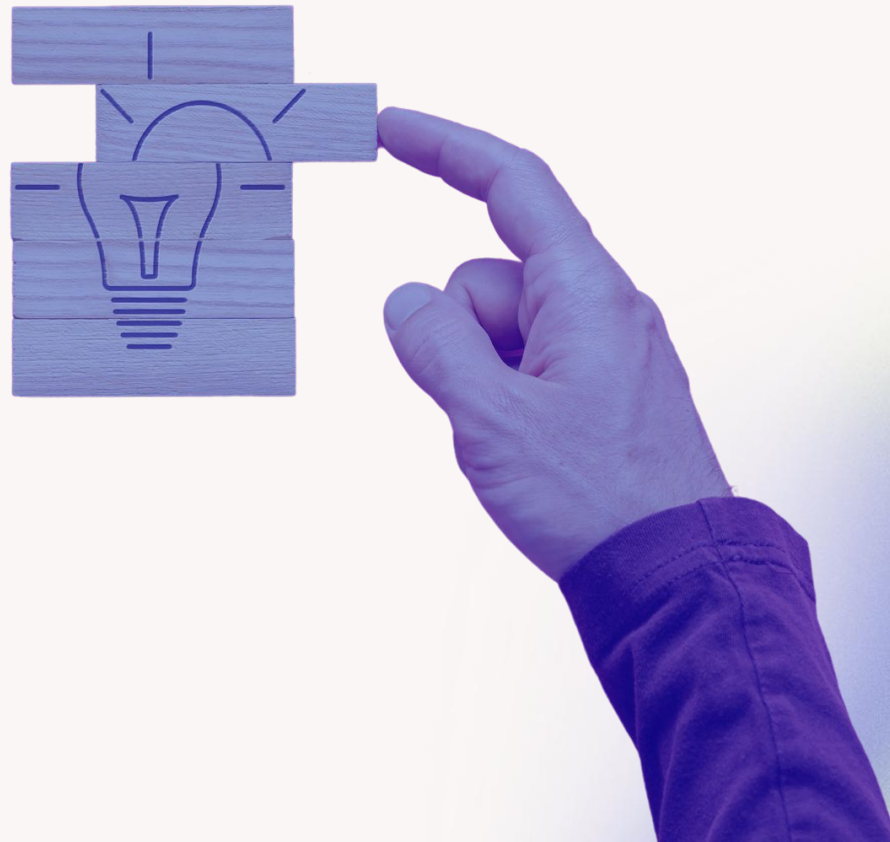


## MÓDULO 4

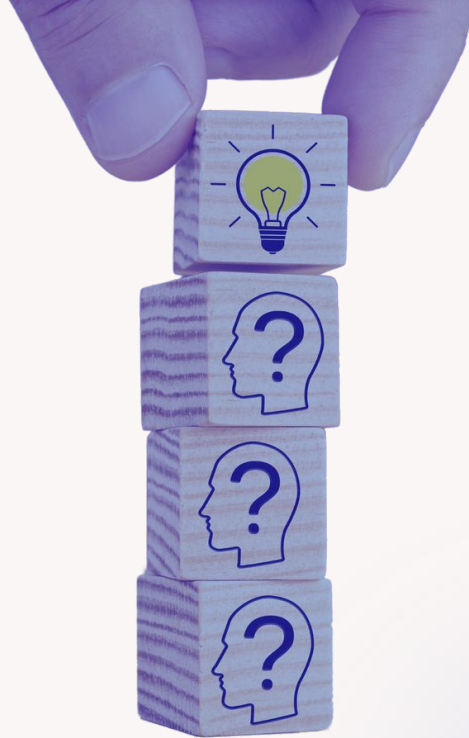
- Testing en un entorno ágil
- El concepto de regresividad en las pruebas
- Pruebas unitarias
- Pruebas unitarias de servicios web
- Pruebas de rendimiento
- **Pruebas funcionales**

# Objetivos

Comprender los fundamentos de las pruebas funcionales, su automatización con Selenium y su integración en entornos ágiles y pipelines de CI/CD.




¿Cómo integrarías  
JMeter en un pipeline  
de CI/CD para  
garantizar que los  
resultados de las  
pruebas de  
rendimiento?



# Fundamentos de Pruebas Funcionales



# ¿Qué es una prueba funcional y cuál es su objetivo?

 Una **prueba funcional** valida que un sistema **cumpla con los requisitos definidos**, verificando **qué hace** el software y no **cómo lo hace**.

## **Objetivos principales:**

- Validar funcionalidades desde la perspectiva del usuario.
- Verificar criterios de aceptación y reglas del negocio.
- Detectar errores en las interacciones del sistema.

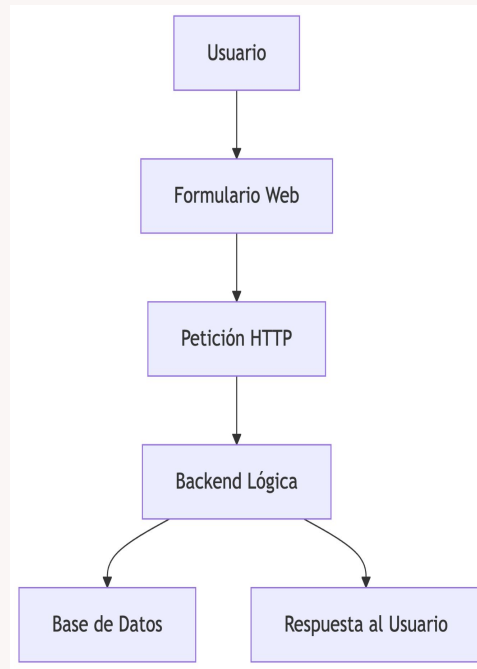
## **Ejemplo de prueba funcional:**

En una app bancaria, verificar que el botón “Transferir” realice correctamente una transferencia cuando el saldo es suficiente.


# Tipos de pruebas funcionales en el desarrollo de software

## ✓ Clasificación común:

Tipo de prueba	Descripción breve
Pruebas de caja negra	Sin conocer la lógica interna del sistema.
Pruebas de aceptación	Valida requerimientos funcionales del negocio.
Pruebas de regresión	Verifica que nuevas funcionalidades no rompan lo anterior.
Pruebas de interfaz	Validan la interacción usuario-sistema.
Pruebas de sistema	Evalúan el sistema completo como un todo.



# Importancia de la regresividad en pruebas funcionales

 Las pruebas de regresión funcional aseguran que el software siga funcionando después de aplicar cambios.

## Situaciones típicas:

- Nuevas funcionalidades agregadas.
- Refactorizaciones o mejoras de rendimiento.
- Cambios en integraciones o librerías externas.

## Ejemplo práctico:

Después de actualizar el sistema de login, se prueba que aún se pueda acceder a “Historial de Compras” sin errores.



# Diferencias entre pruebas manuales y pruebas automatizadas

Característica	Manual	Automatizada
Tiempo de ejecución	Lento	Rápido
Repetitividad	Poco eficiente	Altamente repetible
Escenarios complejos	Fácil de adaptar en tiempo real	Requiere planificación y codificación
Costos iniciales	Bajo	Alto (pero reduce costos a largo plazo)



## **Recomendación:**

Automatiza pruebas repetitivas y de regresión. Haz pruebas manuales para casos exploratorios o nuevos.

# Beneficios de la automatización de pruebas funcionales

## **Ventajas clave:**

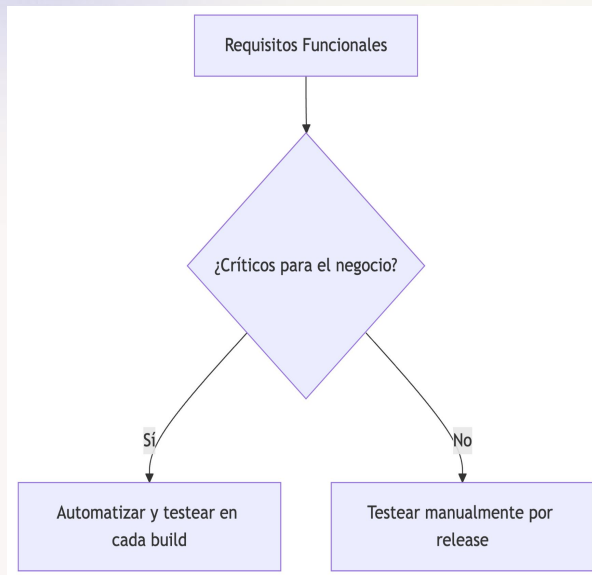
- Reducción de errores humanos.
- Mayor velocidad de validación en ciclos CI/CD.
- Trazabilidad y reportes automáticos.
- Facilidad para ejecutar pruebas cruzadas (navegadores, dispositivos).

## **Caso real:**

Un ecommerce que automatiza sus flujos de login, carrito y checkout reduce el tiempo de validación de 2 días a 1 hora.

# Estrategias para la selección de casos de prueba funcionales

 No todas las funcionalidades deben probarse de la misma forma.



## Estrategias clave:

- **Prueba de funcionalidades críticas** (ej. pagos, login).
- Casos con mayor impacto en negocio.
- Funciones propensas a cambios frecuentes.
- Validaciones ligadas a cumplimiento legal o seguridad.

# Herramientas populares para pruebas funcionales

## ✓ Comparativa de herramientas:

Herramienta	Enfoque	Lenguaje	Ideal para...
<b>Selenium</b>	UI, funcional web	Java, Python, etc.	Web apps con flujos UI complejos
<b>Cypress</b>	End-to-End funcional	JavaScript	SPAs y APIs modernas
<b>TestCafe</b>	Navegador multiplataforma	JS/TS	Pruebas modernas sin drivers externos
<b>Postman</b>	API funcional	Visual/Script	Pruebas de APIs REST

## 📖 Elección recomendada:

- UI web → Selenium o Cypress
- APIs → Postman o REST-assured

# Integración de pruebas funcionales en el ciclo de desarrollo ágil

 Las pruebas funcionales deben ejecutarse desde etapas tempranas.

## ✓ Ciclo sugerido:

Planificación de Sprint → incluir criterios de aceptación.

Desarrollo → pruebas manuales exploratorias.

CI/CD → pruebas automatizadas por feature.

Deploy → validación funcional pre-release.

## Ejemplo de integración CI:

```
- name: Ejecutar pruebas funcionales Selenium  
  run: mvn test -Dsuite=smokeTests
```

# Validación de criterios de aceptación en pruebas funcionales

 Cada historia de usuario en Agile debe tener **criterios de aceptación**.

## Estrategia:

- Convertir criterios de aceptación en **casos de prueba funcionales**.
- Validarlos manual o automáticamente.
- Usar lenguaje Gherkin cuando se integra BDD (Behavior Driven Development).

## Ejemplo de criterio Gherkin:

**Feature:** Transferencia bancaria


**Scenario:** Usuario transfiere monto válido

**Given** el usuario está autenticado

**When** realiza una transferencia de \$100

**Then** la transferencia debe completarse exitosamente

# Reportes y métricas en pruebas funcionales

 Los reportes permiten tomar decisiones rápidas y trazables.

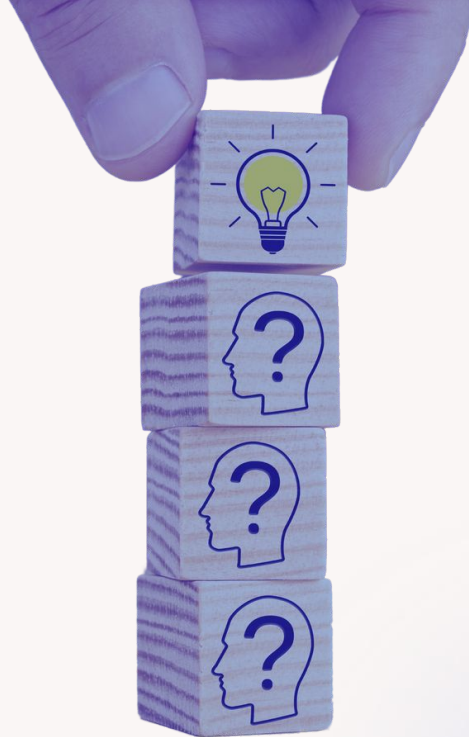
## Métricas comunes:

- % de casos exitosos vs fallidos.
- Tiempo promedio de ejecución.
- Pruebas por funcionalidad/endpoint.
- Regresión: pruebas que fallan luego de cambios recientes.

## Ejemplo con Selenium + Allure Report:

```
mvn clean test  
allure generate target/allure-results --clean -o  
target/allure-report
```


¿Qué criterios  
considerarías para  
decidir entre realizar  
una prueba funcional  
de forma manual?





# Implementación de Pruebas Funcionales con Selenium

# ¿Qué es Selenium y dónde obtenerlo?

 **Selenium** es un conjunto de herramientas de automatización para aplicaciones web. Permite simular interacciones de usuario como clics, escritura y navegación.

## Componentes principales:

- **Selenium WebDriver:** núcleo de automatización.
- **Selenium IDE:** herramienta de grabación rápida.
- **Selenium Grid:** permite pruebas distribuidas.

 Sitio oficial: <https://www.selenium.dev/>

# Instalación y configuración inicial de Selenium

 Puedes usar Selenium con lenguajes como Java, Python, JavaScript, C#.

 Ejemplo en Java con Maven:


**pom.xml**

```
<dependencies>
  <dependency>
    <groupId>org.seleniumhq.selenium</groupId>
    <artifactId>selenium-java</artifactId>
    <version>4.16.1</version>
  </dependency>
</dependencies>
```

 Asegúrate de tener:

- JDK instalado.
- Maven o Gradle para dependencias.
- Un navegador compatible y su **WebDriver** (ej. ChromeDriver para Chrome).

# Creación de scripts de automatización con Selenium

 Simulemos una prueba de login en una página web:

```
import org.openqa.selenium.*;
import org.openqa.selenium.chrome.ChromeDriver;

public class LoginTest {
    public static void main(String[] args) {
        WebDriver driver = new ChromeDriver();
        driver.get("https://ejemplo.com/login");

        driver.findElement(By.id("username")).sendKeys("usuario123");
```

```
        driver.findElement(By.id("password")).sendKeys("clave123");
        driver.findElement(By.id("loginBtn")).click();

        String mensaje =
        driver.findElement(By.id("successMessage")).getText();
        System.out.println("Mensaje recibido: " +
        mensaje);

        driver.quit();
    }
}
```


# Estructura y componentes de una prueba con Selenium

✓ Una prueba típica con Selenium contiene:

Componente	Rol
Setup	Inicializa navegador y WebDriver
Actions	Simula interacciones del usuario
Assertions	Verifica el resultado esperado
Teardown	Cierra navegador y libera recursos

📖 **Patrón recomendado:** Page Object Model (POM) para mantener código mantenible.

# Ejecución de pruebas funcionales con Selenium

 Puedes ejecutar tus pruebas localmente o desde un entorno CI.

## Desde Maven:

```
mvn test
```

## Desde línea de comandos con JUnit o TestNG:

```
java -cp "libs/*" org.junit.runner.JUnitCore LoginTest
```

# Uso de WebDriver para la interacción con aplicaciones web

## ✓ Métodos comunes de WebDriver:


- `findElement(By.id("..."))`
- `click()`
- `sendKeys("texto")`
- `getText()`
- `getTitle()` y `getCurrentUrl()`

## 📖 Ejemplo:

## 📖 Ejemplo:

```
WebElement boton =  
driver.findElement(By.className("btn-primary"));  
boton.click();
```

# Validación de elementos y eventos en Selenium

 Verificamos que ciertos elementos estén presentes o que el sistema reaccione correctamente.

## ✓ Validaciones con assert:

```
String mensaje = driver.findElement(By.id("success")).getText();  
Assert.assertEquals("Login exitoso", mensaje);
```

## ✓ Esperas explícitas (esperar hasta que algo ocurra):

```
WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(10));  
wait.until(ExpectedConditions.visibilityOfElementLocated(By.id("logout")));
```



# Integración de Selenium en pipelines de CI/CD

 Selenium puede integrarse en GitHub Actions, GitLab CI, Jenkins, etc.

## Flujo sugerido:

1. Clonar el repositorio.
2. Instalar dependencias (Maven, WebDriver).
3. Ejecutar pruebas.
4. Publicar resultados.

# Configuración de un pipeline en Jenkins para pruebas funcionales

## Ejemplo básico **Jenkinsfile**:

```
pipeline {
  agent any
  stages {
    stage('Instalar dependencias') {
      steps {
        sh 'mvn clean install'
      }
    }
    stage('Ejecutar pruebas funcionales') {
      steps {
```

```
        sh 'mvn test'
      }
    }
    stage('Publicar Reporte') {
      steps {
        junit 'target/surefire-reports/*.xml'
      }
    }
  }
}
```

# Mejores prácticas y estrategias para pruebas automatizadas con Selenium

## ✓ Recomendaciones:

- Usar Page Object Model (POM) para reutilizar elementos.
- Mantener los datos de prueba en archivos externos o fixtures.
- Usar tags para agrupar pruebas (smoke, regression, full).
- Ejecutar en contenedores (Docker + Selenium Grid).
- Paralelizar pruebas para reducir tiempos.

## 📖 Caso de uso:

Un sistema de reservas hoteleras automatizó con Selenium los flujos de búsqueda, reservas y pagos. Esto redujo el tiempo de QA de 2 días a 30 minutos con Jenkins y Selenium Grid.

¿Qué ventajas ofrece  
la integración de  
Selenium en un  
pipeline de CI/CD y  
cómo contribuye esto  
a la calidad?





# Ejercicio Guiado: Pruebas Funcionales con Selenium



## Ejercicio Guiado: Pruebas Funcionales con Selenium

Las pruebas funcionales permiten verificar si el software cumple con los **requisitos funcionales definidos**, es decir, si "hace lo que se supone que debe hacer". Automatizarlas con herramientas como **Selenium WebDriver** permite mejorar la eficiencia, reducir errores humanos y validar aplicaciones web de forma continua.

## Objetivos

- Instalar y configurar Selenium con Python.
- Crear una prueba funcional sobre un sitio web público.
- Validar eventos, elementos y flujos básicos.
- Ejecutar la prueba localmente y preparar su integración en CI.



## Paso 1: Preparar el entorno

### ¿Qué haremos?

Instalaremos los paquetes necesarios para usar Selenium con Python.

### Instrucciones:

1. Asegúrate de tener Python instalado (`python --version` en terminal).
2. Crea una carpeta de proyecto, por ejemplo: `selenium_test_funcional`





## Paso 1: Preparar el entorno

3. Dentro de la carpeta, ejecuta:

```
python -m venv venv  
source venv/bin/activate    # En Windows: venv\Scripts\activate  
pip install selenium
```

4. Descarga el driver para tu navegador:

- Para Chrome: <https://chromedriver.chromium.org/downloads>

Coloca el ejecutable en la misma carpeta del script o en el PATH del sistema.



## Paso 2: Escribir una prueba funcional



### ¿Qué haremos?

Crearemos un script que navegue al sitio web de DuckDuckGo y valide una búsqueda.

Crea un archivo llamado `test_búsqueda.py`:



## Paso 2: Escribir una prueba funcional

```
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.common.keys import Keys
import time
```

```
driver = webdriver.Chrome()
```

```
driver.get("https://duckduckgo.com/")
```

```
# Buscar campo de texto
```

```
buscador = driver.find_element(By.NAME, "q")
```

```
buscador.send_keys("inmuebles en Bogotá")
```

```
buscador.send_keys(Keys.RETURN)
```

```
# Esperar resultados
```

```
time.sleep(2)
```

```
# Validar que exista algún resultado
```

```
resultados = driver.find_elements(By.CSS_SELECTOR,
".result")
```

```
assert len(resultados) > 0, "No se encontraron
resultados."
```

```
print("✅ Prueba funcional completada con éxito")
```

```
driver.quit()
```






## Paso 3: Validar eventos funcionales



### ¿Qué haremos?

Validaremos la presencia de elementos clave y simularemos interacciones.

### Explicación del script:

-  Simula una búsqueda en un sitio real.
-  Valida que existan resultados en pantalla.
-  Usa `time.sleep()` para esperar la carga (luego puedes usar `WebDriverWait`).

## Preguntas finales

- ¿Qué tipo de errores podrías detectar con esta prueba funcional?
- ¿Por qué es importante automatizar pruebas desde la perspectiva del usuario?
- ¿Qué limitaciones tiene Selenium y cómo las superarías?

## Entregable:

- El script `test_busqueda.py` con su prueba funcional.
- Captura de pantalla con la prueba corriendo y resultado por consola.
- (Opcional) Evidencia de integración en Jenkins o CI.
- Un pequeño documento con:
  - Qué validaron
  - Qué podría fallar si esta prueba no existiera
  - Sugerencias para nuevos casos funcionales



## Resumen de lo aprendido

- **Pruebas Funcionales y su Rol:** Verifican que el sistema cumpla con los requisitos funcionales esperados, validando entradas, procesos y salidas.
- **Automatización con Selenium:** Uso de WebDriver, scripts personalizados y validación de eventos para simular interacciones reales con aplicaciones web.
- **Integración en el Ciclo Ágil:** Inclusión de pruebas funcionales en sprints y pipelines CI/CD, utilizando herramientas como Jenkins para su ejecución.
- **Mejores Prácticas:** Selección eficiente de casos de prueba, uso de criterios de aceptación, métricas de cobertura y mantenimiento de pruebas escalables.

# Próxima clase...

Principios fundamentales de diseño de una arquitectura



