Sesión 8: Pruebas con Selenium

Duración: 45 minutos

🎯 Objetivos de aprendizaje

Al finalizar esta sesión, el participante será capaz de:

- Configurar Selenium WebDriver para testing automatizado de UI
- Implementar el patrón Page Object Model para tests mantenibles
- Crear tests end-to-end robustos y confiables
- Manejar elementos dinámicos, waits y sincronización
- Integrar Selenium tests en pipelines de CI/CD
- Implementar testing cross-browser y responsive
- Configurar reporting y debugging avanzado

📋 Contenido

1. Selenium en el Ecosistema de Testing (8 min)

Analogía: Si las pruebas unitarias son como inspeccionar los componentes de un auto por separado, las pruebas de Selenium son como hacer un test drive completo - verifican que todo funcione junto desde la perspectiva del usuario final.

Posición en la pirámide de testing:

```
           🌐 E2E Tests (Selenium)
          ╱‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾╲
         ╱ Lentas pero realistas ╲  ← 5-10% del total
        ╱ User journey completo   ╲   Críticas para confianza
       ╱                   ╲
      ╱_____________________╲
     ╱                     ╲
    ╱ 🔗 Integration Tests      ╲  ← 20-30% del total
   ╱   API, Database, Services    ╲
  ╱_______________________________╲
 ╱                           ╲
╱  🧪 Unit Tests               ╲ ← 60-70% del total
╲_________________________________╱  Base sólida y rápida
```

¿Cuándo usar Selenium?

```
✅ IDEAL PARA:
├── User journeys críticos (login, checkout, registro)
├── Flows complejos multi-página
├── Validación de integración frontend-backend
├── Testing cross-browser compatibility
├── Regression testing de UI changes
└── Smoke tests post-deployment

❌ NO USAR PARA:
├── Validación de lógica de negocio (unit tests)
├── Testing de APIs (integration tests)
├── Performance testing (herramientas específicas)
├── Debugging de CSS/layout (herramientas de dev)
└── Testing de cada input field (demasiado granular)
```

Selenium vs Alternativas:

```
🔧 SELENIUM:
├── Pros: Maduro, multi-lenguaje, gran ecosistema
├── Contras: Lento, complejo setup, frágil
└── Mejor para: Testing exhaustivo, compatibilidad

⚡ PLAYWRIGHT:
├── Pros: Rápido, built-in waits, multi-browser
├── Contras: Relativamente nuevo, menos recursos
└── Mejor para: Testing moderno, CI/CD rápido

🎭 CYPRESS:
├── Pros: DX excelente, debugging visual, time-travel
├── Contras: Solo Chrome, limitado para multi-tab
└── Mejor para: Development-time testing
```

2. Configuración y WebDriver Setup (7 min)

Arquitectura de Selenium:

```
📱 Test Script (Python)
  ↓ WebDriver API calls
🔧 WebDriver (ChromeDriver, FirefoxDriver, etc.)
  ↓ W3C WebDriver Protocol
🌐 Browser (Chrome, Firefox, Safari, Edge)
  ↓ JavaScript execution
📄 Web Application (DOM manipulation)
```

Setup moderno con WebDriver Manager:

python

```python
from selenium import webdriver
from selenium.webdriver.chrome.service import Service
from webdriver_manager.chrome import ChromeDriverManager

# Automatic driver management
def get_driver():
    options = webdriver.ChromeOptions()
    options.add_argument('--headless')  # Para CI
    options.add_argument('--no-sandbox')
    options.add_argument('--disable-dev-shm-usage')

    service = Service(ChromeDriverManager().install())
    return webdriver.Chrome(service=service, options=options)
```

Configuración cross-browser:

python

```
BROWSERS = {
    'chrome': {
        'driver': webdriver.Chrome,
        'options': webdriver.ChromeOptions(),
        'manager': ChromeDriverManager
    },
    'firefox': {
        'driver': webdriver.Firefox,
        'options': webdriver.FirefoxOptions(),
        'manager': GeckoDriverManager
    },
    'edge': {
        'driver': webdriver.Edge,
        'options': webdriver.EdgeOptions(),
        'manager': EdgeChromiumDriverManager
    }
}
```

Docker para testing consistente:

```
yaml
```

```yaml
# docker-compose.selenium.yml
version: '3.8'
services:
  selenium-hub:
    image: selenium/hub:4.15.0
    ports:
      - "4444:4444"

  chrome:
    image: selenium/node-chrome:4.15.0
    environment:
      - HUB_HOST=selenium-hub
    depends_on:
      - selenium-hub

  firefox:
    image: selenium/node-firefox:4.15.0
    environment:
      - HUB_HOST=selenium-hub
    depends_on:
      - selenium-hub
```

3. Page Object Model (POM) - Arquitectura Mantenible (10 min)

¿Por qué Page Object Model?

```
❌ SIN POM (frágil y repetitivo):
def test_login():
    driver.find_element(By.ID, "username").send_keys("user")
    driver.find_element(By.ID, "password").send_keys("pass")
    driver.find_element(By.CSS_SELECTOR, "button[type='submit']").click()

def test_login_error():
    driver.find_element(By.ID, "username").send_keys("wrong")
    driver.find_element(By.ID, "password").send_keys("wrong")
    driver.find_element(By.CSS_SELECTOR, "button[type='submit']").click()
    # Si cambia el selector, hay que actualizar en muchos lugares

✅ CON POM (mantenible y reutilizable):
login_page.enter_credentials("user", "pass")
login_page.click_submit()
# Si cambia el selector, solo se actualiza en la Page Object
```

Estructura de Page Object:

python

```python
class BasePage:
    """Funcionalidad común a todas las páginas"""
    def __init__(self, driver):
        self.driver = driver
        self.wait = WebDriverWait(driver, 10)

    def find_element(self, locator):
        return self.wait.until(EC.presence_of_element_located(locator))

    def click(self, locator):
        element = self.wait.until(EC.element_to_be_clickable(locator))
        element.click()

class LoginPage(BasePage):
    """Page Object para página de login"""
    # Locators (separados de las acciones)
    USERNAME_INPUT = (By.ID, "username")
    PASSWORD_INPUT = (By.ID, "password")
    SUBMIT_BUTTON = (By.CSS_SELECTOR, "button[type='submit']")
    ERROR_MESSAGE = (By.CLASS_NAME, "error-message")

    def enter_username(self, username):
        self.find_element(self.USERNAME_INPUT).send_keys(username)

    def enter_password(self, password):
        self.find_element(self.PASSWORD_INPUT).send_keys(password)

    def click_submit(self):
        self.click(self.SUBMIT_BUTTON)

    def get_error_message(self):
        return self.find_element(self.ERROR_MESSAGE).text
```

Patrón de composición:

python

```
class LoginFlow:
    """Combina múltiples page objects para flows complejos"""
    def __init__(self, driver):
        self.login_page = LoginPage(driver)
        self.dashboard_page = DashboardPage(driver)

    def login_successful(self, username, password):
        self.login_page.enter_username(username)
        self.login_page.enter_password(password)
        self.login_page.click_submit()
        return self.dashboard_page.is_loaded()
```

4. Waits y Sincronización - Clave para Tests Estables (8 min)

Problema de timing en tests UI:

```
🕐 PROBLEMA COMÚN:
Test Script:    |-- Click Button --|-- Check Result --|
Browser:        |-- Loading... -------|-- Render --|
Result:         ❌ Test falla porque verifica antes de que termine la carga
```

Tipos de waits:

```
python

# ❌ Hard Waits (nunca usar)
import time
time.sleep(5)  # Siempre espera 5s, lento e impredecible

# ⚠️ Implicit Waits (usar con cuidado)
driver.implicitly_wait(10)  # Global, puede enmascarar problemas

# ✅ Explicit Waits (recomendado)
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC

wait = WebDriverWait(driver, 10)
element = wait.until(EC.element_to_be_clickable((By.ID, "submit")))
```

Expected Conditions más útiles:

```
python
```

```
# Presencia de elemento
EC.presence_of_element_located((By.ID, "content"))

# Element clickeable
EC.element_to_be_clickable((By.ID, "button"))

# Texto específico presente
EC.text_to_be_present_in_element((By.ID, "status"), "Complete")

# Element visible
EC.visibility_of_element_located((By.CLASS_NAME, "modal"))

# Element NOT visible (para spinners)
EC.invisibility_of_element_located((By.CLASS_NAME, "loading"))

# URL contiene texto
EC.url_contains("dashboard")

# Título de página
EC.title_is("Welcome Page")
```

Custom Expected Conditions:

```python
class ElementHasAttribute:
    """Custom condition: element tiene atributo específico"""
    def __init__(self, locator, attribute):
        self.locator = locator
        self.attribute = attribute

    def __call__(self, driver):
        element = driver.find_element(*self.locator)
        return element.get_attribute(self.attribute) is not None

# Uso
wait.until(ElementHasAttribute((By.ID, "input"), "data-loaded"))
```

5. Testing de Elementos Dinámicos y SPAs (7 min)

Challenges de aplicaciones modernas:

```
🔄 DYNAMIC CONTENT:
├──── Content loaded via AJAX
├──── Elements appearing/disappearing
├──── Changing attributes/properties
└──── Conditional rendering

⚡ SPA CHALLENGES:
├──── URL doesn't change on navigation
├──── Virtual DOM updates
├──── Async state updates
├──── Client-side routing
└──── Progressive loading
```

Estrategias para contenido dinámico:

```python
class DynamicContentPage(BasePage):
    def wait_for_content_loaded(self):
        """Esperar a que contenido dinámico se cargue"""
        # Método 1: Esperar por atributo específico
        self.wait.until(
            EC.presence_of_element_located((By.CSS_SELECTOR, "[data-loaded='true']"))
        )

        # Método 2: Esperar por desaparición de loader
        self.wait.until(
            EC.invisibility_of_element_located((By.CLASS_NAME, "spinner"))
        )

        # Método 3: Esperar por AJAX completion (JavaScript)
        self.wait.until(lambda driver: driver.execute_script(
            "return jQuery.active == 0"  # Si usa jQuery
        ))

    def wait_for_table_data(self, expected_rows):
        """Esperar número específico de filas en tabla"""
        self.wait.until(lambda driver:
            len(driver.find_elements(By.CSS_SELECTOR, "table tbody tr")) >= expected_rows
        )
```

Handling de Shadow DOM:

```python
def find_in_shadow_dom(driver, host_element, css_selector):
    """Buscar elemento dentro de Shadow DOM"""
    shadow_root = driver.execute_script(
        "return arguments[0].shadowRoot", host_element
    )
    return shadow_root.find_element(By.CSS_SELECTOR, css_selector)
```

Testing de drag and drop:

```python
def drag_and_drop_test(self):
    source = self.find_element((By.ID, "draggable"))
    target = self.find_element((By.ID, "droppable"))

    # Method 1: ActionChains
    ActionChains(self.driver).drag_and_drop(source, target).perform()

    # Method 2: HTML5 drag and drop (más compatible)
    self.driver.execute_script("""
        var source = arguments[0];
        var target = arguments[1];
        var event = new DragEvent('dragstart', {bubbles: true});
        source.dispatchEvent(event);

        var dropEvent = new DragEvent('drop', {bubbles: true});
        target.dispatchEvent(dropEvent);
    """, source, target)
```

6. CI/CD Integration y Headless Testing (3 min)

Configuración para CI/CD:

```python
```

```
def get_ci_driver():
    """Driver optimizado para CI/CD"""
    options = webdriver.ChromeOptions()

    # Headless mode
    options.add_argument('--headless')

    # CI optimizations
    options.add_argument('--no-sandbox')
    options.add_argument('--disable-dev-shm-usage')
    options.add_argument('--disable-gpu')
    options.add_argument('--window-size=1920,1080')

    # Performance optimizations
    options.add_argument('--disable-extensions')
    options.add_argument('--disable-plugins')
    options.add_argument('--disable-images')  # Para tests más rápidos

    return webdriver.Chrome(options=options)
```

Docker integration:

```yaml
# En GitHub Actions
- name: Run Selenium tests
  run: |
    docker-compose -f docker-compose.selenium.yml up -d
    pytest tests/selenium/ --browser=remote --hub-url=http://localhost:4444
    docker-compose -f docker-compose.selenium.yml down
```

Parallel testing:

```python
# pytest-xdist para paralelización
pytest tests/selenium/ -n 3  # 3 browsers en paralelo

# Custom parallel setup
@pytest.fixture(scope="session", params=["chrome", "firefox"])
def browser_type(request):
    return request.param
```

7. Debugging y Troubleshooting (2 min)

Screenshots on failure:

```python
@pytest.fixture(autouse=True)
def screenshot_on_failure(request, driver):
    yield
    if request.node.rep_call.failed:
        timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
        screenshot_name = f"failure_{request.node.name}_{timestamp}.png"
        driver.save_screenshot(f"screenshots/{screenshot_name}")
```

Video recording:

```python
# Con pytest-html-reporter
pytest tests/selenium/ --html=report.html --self-contained-html
```

Console logs capture:

```python
def test_with_console_logs(driver):
    # Capturar console logs
    logs = driver.get_log('browser')
    for log in logs:
        if log['level'] == 'SEVERE':
            pytest.fail(f"JavaScript error: {log['message']}")
```

⚡ Best Practices para Selenium

Reliability:

- ✅ Always use explicit waits
- ✅ Implement retry mechanisms for flaky elements
- ✅ Use stable locators (ID > CSS > XPath)
- ✅ Avoid testing implementation details

Performance:

- ✅ Run tests in parallel where possible
- ✅ Use headless mode in CI
- ✅ Disable images/extensions for speed
- ✅ Implement smart test selection

Maintainability:

- ✅ Use Page Object Model consistently
- ✅ Keep tests focused and atomic
- ✅ Extract common functionality to base classes
- ✅ Use data-test attributes for stable selection

Debugging:

- ✅ Take screenshots on failures
- ✅ Capture browser logs
- ✅ Use explicit test names
- ✅ Implement proper error messages

🚫 Common Antipatterns

❌ Testing too much in one test:

```python
def test_entire_user_journey():  # BAD: 50+ steps
    login()
    create_account()
    make_purchase()
    check_email()
    # ... 30 more steps
```

❌ Hard-coded sleeps:

```python
time.sleep(5)  # BAD: Always waits, slow and unreliable
```

❌ Brittle locators:

```python
# BAD: Breaks with any HTML change
driver.find_element(By.XPATH, "/html/body/div[3]/div[2]/span[1]/button")

# GOOD: Semantic and stable
driver.find_element(By.CSS_SELECTOR, "[data-testid='submit-button']")
```

❌ **No abstraction:**

```python
# BAD: Repeated in every test
driver.find_element(By.ID, "username").send_keys("test")
driver.find_element(By.ID, "password").send_keys("pass")
driver.find_element(By.ID, "submit").click()
```

📝 Resumen Final

1. **Selenium** es ideal para testing E2E de user journeys críticos
2. **Page Object Model** es esencial para tests mantenibles
3. **Explicit waits** son clave para tests estables y confiables
4. **Headless testing** optimiza ejecución en CI/CD
5. **Cross-browser testing** asegura compatibilidad
6. **Debugging tools** facilitan troubleshooting
7. **Best practices** previenen tests frágiles y lentos

🏃 Actividad Práctica Sugerida

Ejercicio 1 - Setup básico:

1. Configurar WebDriver con automatic driver management
2. Crear test simple de login con explicit waits
3. Implementar screenshot on failure

Ejercicio 2 - Page Object Model:

1. Refactorizar test de login usando POM
2. Crear flow completo (login → dashboard → action)

3. Añadir cross-browser testing

Ejercicio 3 - CI Integration:

1. Configurar tests para ejecutar en headless mode
2. Integrar con GitHub Actions
3. Implementar parallel testing

Tiempo estimado: 35-40 minutos adicionales