Guía Completa de Pruebas Selenium

ImageProcessor - Pruebas End-to-End

Proyecto: ImageProcessor

Stack: Python Flask + React Vite

Herramienta: Selenium WebDriver

Fecha: 2025

Tabla de Contenidos

1. Introducción a las Pruebas Selenium

2. Estructura del Proyecto

3. Instalación y Configuración

4. Configuración Base (conftest.py)

5. Tests de Navegación

6. Tests de Carga de Archivos

7. Tests de Procesamiento

8. Tests de Descarga

9. Ejecución de Pruebas

10. Mejores Prácticas

11. Solución de Problemas

12. Anexos

1. Introducción a las Pruebas Selenium

¿Qué es Selenium?

Selenium WebDriver es una herramienta de automatización de pruebas que permite controlar navegadores web de forma programática. Es ideal para realizar pruebas end-to-end (E2E) que verifican el funcionamiento completo de una aplicación web desde la perspectiva del usuario.

¿Por qué Selenium para ImageProcessor?

ImageProcessor requiere pruebas E2E porque:

- Involucra interacción compleja: carga de archivos, procesamiento asíncrono, descargas
- Tiene un frontend React y backend Flask que deben funcionar juntos
- Requiere validación de flujos completos: cargar → procesar → descargar
- Necesita verificar elementos visuales y comportamientos dinámicos

Arquitectura de las Pruebas

Las pruebas están organizadas en cuatro categorías principales:

1. **Tests de Navegación:** Verifican que todas las páginas cargan correctamente
2. **Tests de Upload:** Prueban la carga de archivos individual y masiva
3. **Tests de Procesamiento:** Validan switches, opciones y procesamiento
4. **Tests de Descarga:** Comprueban descarga individual y en ZIP

2. Estructura del Proyecto

Estructura de Carpetas Completa

```
PRUEBA_PRACTICAS/ │ ├── backend/ │ ├── app.py │ ├── uploads/ # Archivos temporales │ ├── tests/ # Tests backend (opcional) │ │ ├── __init__.py │ │ ├── test_app.py │ │ └── test_integration.py │ └── requirements.txt │ ├── frontend/ │ ├── src/ │ │ ├── components/ │ │ ├── hooks/ │ │ └── App.jsx │ ├── tests/ # Tests frontend │ │ ├── selenium/ # TESTS SELENIUM │ │ │ ├── __init__.py │ │ │ ├── conftest.py # Configuración │ │ │ ├── test_navigation.py # Tests navegación │ │ │ ├── test_upload.py # Tests carga │ │ │ ├── test_processing.py # Tests procesamiento │ │ │ ├── test_download.py # Tests descarga │ │ │ └── test_assets/ # Imágenes de prueba │ │ │ └── test_image.png │ │ └── pytest.ini # Config pytest │ └── package.json │ ├── requirements-test.txt # Dependencias tests └── run_tests.sh # Script ejecución
```

Archivos Clave

Archivo	Propósito
`conftest.py`	Configuración compartida de fixtures pytest
`test_*.py`	Archivos de pruebas individuales
`pytest.ini`	Configuración global de pytest
`requirements-test.txt`	Dependencias Python para testing

`run_tests.sh`	Script automatizado de ejecución

3. Instalación y Configuración

Paso 1: Instalar Dependencias

Crear requirements-test.txt

En la raíz del proyecto, crea el archivo `requirements-test.txt` :

```
selenium==4.15.2
pytest==7.4.3
pytest-selenium==4.0.1
webdriver-manager==4.0.1
Pillow==10.1.0
requests==2.31.0
```

Instalar dependencias

```
$ pip install -r requirements-test.txt
```

> **Nota:** webdriver-manager descarga automáticamente los drivers de Chrome/Firefox, eliminando la necesidad de configuración manual.

Paso 2: Crear Estructura de Carpetas

```
$ cd frontend $ mkdir -p tests/selenium/test_assets $ touch tests/selenium/__init__.py $ touch tests/selenium/conftest.py $ touch tests/selenium/test_navigation.py $ touch tests/selenium/test_upload.py $ touch tests/selenium/test_processing.py $ touch tests/selenium/test_download.py $ touch tests/pytest.ini
```

Paso 3: Verificar Instalación

```
$ python -c "import selenium; print(selenium.__version__)" $
pytest --version
```

> Si ambos comandos funcionan sin errores, la instalación fue exitosa.

4. Configuración Base (conftest.py)

¿Qué es conftest.py?

Es un archivo especial de pytest que contiene fixtures compartidas por todos los tests. Los fixtures son funciones que proporcionan recursos o configuraciones reutilizables.

Código Completo

Crea `frontend/tests/selenium/conftest.py` :

```
import pytest
from selenium import webdriver
from selenium.webdriver.chrome.service import Service
from selenium.webdriver.chrome.options import Options
from webdriver_manager.chrome import ChromeDriverManager
import time
import os

@pytest.fixture(scope="session")
def chrome_options():
    """Opciones para Chrome"""
    options = Options()
    # Descomenta para modo headless (sin ventana)
    # options.add_argument("--headless")
    options.add_argument("--no-sandbox")
    options.add_argument("--disable-dev-shm-usage")
    options.add_argument("--window-size=1920,1080")
    return options

@pytest.fixture(scope="function")
def driver(chrome_options):
    """Inicializar driver de Selenium"""
    service = Service(ChromeDriverManager().install())
    driver = webdriver.Chrome(service=service, options=chrome_options)
    driver.implicitly_wait(10)

    yield driver

    # Cleanup
    driver.quit()

@pytest.fixture(scope="session")
```

```
def base_url():
    """URL base de la aplicación"""
    return "http://localhost:5173"  # Vite default port

@pytest.fixture(scope="session")
def api_url():
    """URL base del API"""
    return "http://localhost:5000/api"

@pytest.fixture(scope="function")
def test_image_path():
    """Ruta a imagen de prueba"""
    test_assets = os.path.join(os.path.dirname(__file__), "test_assets")
    os.makedirs(test_assets, exist_ok=True)

    from PIL import Image
    img_path = os.path.join(test_assets, "test_image.png")

    if not os.path.exists(img_path):
        img = Image.new('RGB', (800, 600), color='red')
        img.save(img_path)

    return img_path
```

Explicación de Fixtures

Fixture	Scope	Descripción
`chrome_options`	session	Configuración del navegador Chrome
`driver`	function	Instancia de WebDriver (se crea por cada test)
`base_url`	session	URL del frontend
`api_url`	session	URL del backend API
`test_image_path`	function	Genera imagen de prueba automáticamente

5. Tests de Navegación

Objetivo

Verificar que todas las páginas cargan correctamente y la navegación funciona.

Código Completo

Crea `frontend/tests/selenium/test_navigation.py` :

```
import pytest
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
import time

class TestNavigation:
    """Tests de navegación entre páginas"""

    def test_landing_page_loads(self, driver, base_url):
        """Verificar que la página de inicio carga correctamente"""
        driver.get(base_url)

        WebDriverWait(driver, 10).until(
            EC.presence_of_element_located((By.CLASS_NAME, "hero-section"))
        )

        assert "Procesamiento" in driver.page_source
        assert "inteligente" in driver.page_source

    def test_navigate_to_processor(self, driver, base_url):
        """Navegar desde inicio a procesador"""
        driver.get(base_url)

        processor_link = WebDriverWait(driver, 10).until(
            EC.element_to_be_clickable((By.XPATH,
                "//button[contains(text(), 'Procesador')]"))
        )
        processor_link.click()

        WebDriverWait(driver, 10).until(
            EC.presence_of_element_located((By.CLASS_NAME, "processor-main"))
        )
```

```
        assert "Procesador de Imágenes" in driver.page_source

    def test_navigate_to_help(self, driver, base_url):
        """Navegar a página de ayuda"""
        driver.get(base_url)

        help_link = WebDriverWait(driver, 10).until(
            EC.element_to_be_clickable((By.XPATH,
                "//button[contains(text(), 'Ayuda')]"))
        )
        help_link.click()

        WebDriverWait(driver, 10).until(
            EC.presence_of_element_located((By.CLASS_NAME, "help-page-main"))
        )

        assert "Centro de Ayuda" in driver.page_source

    def test_all_navigation_links_work(self, driver, base_url):
        """Verificar que todos los links de navegación funcionan"""
        driver.get(base_url)

        pages = [
            ("Inicio", "hero-section"),
            ("Procesador", "processor-main"),
            ("Ayuda", "help-page-main")
        ]

        for page_name, expected_class in pages:
            link = WebDriverWait(driver, 10).until(
                EC.element_to_be_clickable((By.XPATH,
                    f"//button[contains(text(), '{page_name}')]"))
            )
            link.click()
            time.sleep(1)

            WebDriverWait(driver, 10).until(
                EC.presence_of_element_located((By.CLASS_NAME, expected_class))
            )
```

Conceptos Clave

- **WebDriverWait:** Espera hasta que un elemento esté disponible (máx 10 segundos)
- **By.CLASS_NAME:** Busca elementos por su clase CSS
- **By.XPATH:** Busca elementos usando expresiones XPath
- **EC.presence_of_element_located:** Condición de espera para existencia del elemento
- **EC.element_to_be_clickable:** Condición de espera para clickeabilidad

6. Tests de Carga de Archivos

Objetivo

Verificar la funcionalidad de carga de imágenes individuales y múltiples.

Código Completo

Crea `frontend/tests/selenium/test_upload.py` :

```
import pytest
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
import time

class TestUpload:
    """Tests de carga de archivos"""

    @pytest.fixture(autouse=True)
    def setup(self, driver, base_url):
        """Navegar al procesador antes de cada test"""
        driver.get(base_url)
        processor_link = WebDriverWait(driver, 10).until(
            EC.element_to_be_clickable((By.XPATH,
                "//button[contains(text(), 'Procesador')]"))
        )
        processor_link.click()
        time.sleep(1)

    def test_file_input_exists(self, driver):
        """Verificar que existe el input de archivos"""
        file_input = driver.find_element(By.CSS_SELECTOR,
            "input[type='file']")
        assert file_input is not None
        assert file_input.get_attribute("accept") == "image/*,.zip"

    def test_upload_single_image(self, driver, test_image_path):
        """Cargar una sola imagen"""
        file_input = driver.find_element(By.CSS_SELECTOR,
            "input[type='file']")
        file_input.send_keys(test_image_path)

        WebDriverWait(driver, 10).until(
```

```
            EC.text_to_be_present_in_element(
                (By.CLASS_NAME, "processor-upload-title"),
                "1 archivos cargados"
            )
        )

        process_button = WebDriverWait(driver, 10).until(
            EC.presence_of_element_located((By.CLASS_NAME, "process-btn"))
        )
        assert process_button.is_displayed()

    def test_upload_area_drag_active(self, driver):
        """Verificar que el área de drag & drop existe"""
        upload_area = driver.find_element(By.CLASS_NAME,
            "processor-upload-area")
        assert upload_area is not None

    def test_clear_files_button_appears(self, driver, test_image_path):
        """Verificar que aparece el botón de resetear"""
        file_input = driver.find_element(By.CSS_SELECTOR,
            "input[type='file']")
        file_input.send_keys(test_image_path)

        time.sleep(2)

        clear_button = WebDriverWait(driver, 10).until(
            EC.presence_of_element_located(
                (By.CLASS_NAME, "processor-clear-btn"))
        )
        assert clear_button.is_displayed()
        assert "Resetear" in clear_button.text
```

Características Importantes

- **autouse=True:** El fixture setup se ejecuta automáticamente antes de cada test
- **send_keys():** Método para enviar rutas de archivo al input
- **get_attribute():** Obtiene atributos HTML del elemento
- **is_displayed():** Verifica si el elemento es visible

7. Tests de Procesamiento

Objetivo

Validar switches, inputs de dimensiones y el proceso completo de procesamiento.

Código Completo (Parte 1)

Crea `frontend/tests/selenium/test_processing.py` :

```
import pytest
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
import time

class TestProcessing:
    """Tests de procesamiento de imágenes"""

    @pytest.fixture(autouse=True)
    def setup_and_upload(self, driver, base_url, test_image_path):
        """Setup: navegar y cargar imagen"""
        driver.get(base_url)

        processor_link = WebDriverWait(driver, 10).until(
            EC.element_to_be_clickable((By.XPATH,
                "//button[contains(text(), 'Procesador')]"))
        )
        processor_link.click()
        time.sleep(1)

        file_input = driver.find_element(By.CSS_SELECTOR,
            "input[type='file']")
        file_input.send_keys(test_image_path)
        time.sleep(2)

    def test_switches_are_present(self, driver):
        """Verificar que existen los switches de opciones"""
        bg_removal_card = driver.find_element(By.XPATH,
            "//h4[contains(text(), 'Eliminar Fondo')]"
            "/ancestor::div[contains(@class, 'processor-option-card')]"
        )
        assert bg_removal_card is not None
```

```
        resize_card = driver.find_element(By.XPATH,
            "//h4[contains(text(), 'Redimensionar')]"
            "/ancestor::div[contains(@class, 'processor-option-card')]"
        )
        assert resize_card is not None

    def test_toggle_background_removal(self, driver):
        """Activar/desactivar eliminación de fondo"""
        toggle = driver.find_element(By.XPATH,
            "//h4[contains(text(), 'Eliminar Fondo')]"
            "/ancestor::div[contains(@class, 'processor-option-card')]"
            "//div[contains(@class, 'processor-toggle')]"
        )

        initial_classes = toggle.get_attribute("class")
        toggle.click()
        time.sleep(0.5)

        after_classes = toggle.get_attribute("class")
        assert initial_classes != after_classes
```

Código Completo (Parte 2)

```
    def test_resize_shows_dimensions_panel(self, driver):
        """Verificar que al activar resize aparece el panel"""
        resize_toggle = driver.find_element(By.XPATH,
            "//h4[contains(text(), 'Redimensionar')]"
            "/ancestor::div[contains(@class, 'processor-option-card')]"
            "//div[contains(@class, 'processor-toggle')]"
        )
        resize_toggle.click()
        time.sleep(1)

        dimensions_panel = WebDriverWait(driver, 10).until(
            EC.presence_of_element_located(
                (By.CLASS_NAME, "dimensions-panel"))
        )
        assert dimensions_panel.is_displayed()

    def test_dimension_inputs_appear(self, driver):
        """Verificar inputs de ancho y alto"""
        resize_toggle = driver.find_element(By.XPATH,
            "//h4[contains(text(), 'Redimensionar')]"
            "/ancestor::div[contains(@class, 'processor-option-card')]"
            "//div[contains(@class, 'processor-toggle')]"
        )
        resize_toggle.click()
        time.sleep(1)

        width_input = driver.find_element(By.CSS_SELECTOR,
```

```
            "input[type='number']")
        assert width_input is not None

    def test_process_button_is_clickable(self, driver):
        """Verificar que el botón de procesar es clickeable"""
        process_button = WebDriverWait(driver, 10).until(
            EC.element_to_be_clickable((By.CLASS_NAME, "process-btn"))
        )
        assert process_button.is_enabled()

    def test_full_processing_flow(self, driver):
        """Test completo: activar opciones y procesar"""
        bg_toggle = driver.find_element(By.XPATH,
            "//h4[contains(text(), 'Eliminar Fondo')]"
            "/ancestor::div[contains(@class, 'processor-option-card')]"
            "//div[contains(@class, 'processor-toggle')]"
        )
        bg_toggle.click()
        time.sleep(0.5)

        process_button = driver.find_element(By.CLASS_NAME, "process-btn")
        process_button.click()

        results_section = WebDriverWait(driver, 30).until(
            EC.presence_of_element_located(
                (By.CLASS_NAME, "processor-results-section"))
        )
        assert results_section.is_displayed()

        time.sleep(5)

        processing_items = driver.find_elements(By.CLASS_NAME,
            "processing-item")
        assert len(processing_items) > 0
```

8. Tests de Descarga

Objetivo

Verificar que las descargas funcionan correctamente (imagen individual o ZIP múltiple).

Código Completo

Crea `frontend/tests/selenium/test_download.py` :

```
import pytest
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
import time

class TestDownload:
    """Tests de descarga de imágenes procesadas"""

    @pytest.fixture(autouse=True)
    def setup_process_image(self, driver, base_url, test_image_path):
        """Setup: procesar una imagen"""
        driver.get(base_url)

        processor_link = WebDriverWait(driver, 10).until(
            EC.element_to_be_clickable((By.XPATH,
                "//button[contains(text(), 'Procesador')]"))
        )
        processor_link.click()
        time.sleep(1)

        file_input = driver.find_element(By.CSS_SELECTOR,
            "input[type='file']")
        file_input.send_keys(test_image_path)
        time.sleep(2)

        process_button = driver.find_element(By.CLASS_NAME, "process-btn")
        process_button.click()

        WebDriverWait(driver, 30).until(
            EC.presence_of_element_located(
                (By.CLASS_NAME, "processor-download-section"))
        )
        time.sleep(2)
```

```
    def test_download_button_appears(self, driver):
        """Verificar que aparece el botón de descarga"""
        download_button = driver.find_element(By.CLASS_NAME,
            "processor-download-btn")
        assert download_button.is_displayed()
        assert download_button.is_enabled()

    def test_download_summary_shows_stats(self, driver):
        """Verificar que aparecen estadísticas"""
        summary = driver.find_element(By.CLASS_NAME, "download-summary")
        assert "procesada" in summary.text.lower()

    def test_download_button_text(self, driver):
        """Verificar texto del botón de descarga"""
        download_button = driver.find_element(By.CLASS_NAME,
            "processor-download-btn")
        button_text = download_button.text
        assert "Descargar" in button_text
```

9. Ejecución de Pruebas

Configuración pytest.ini

Crea `frontend/tests/pytest.ini` :

```
[pytest]
minversion = 7.0
testpaths = selenium
python_files = test_*.py
python_classes = Test*
python_functions = test_*
addopts = -v --tb=short --strict-markers
markers =
    slow: marks tests as slow
    integration: marks tests as integration tests
```

Script de Ejecución Automatizada

Crea `run_tests.sh` en la raíz del proyecto:

```
#!/bin/bash

echo "🧪 Ejecutando Tests de ImageProcessor"
echo "======================================"

# Verificar backend
if ! curl -s http://localhost:5000/api/health > /dev/null; then
    echo "❌ Backend no está corriendo"
    echo "Inicia: python backend/app.py"
    exit 1
fi

# Verificar frontend
if ! curl -s http://localhost:5173 > /dev/null; then
    echo "❌ Frontend no está corriendo"
    echo "Inicia: npm run dev"
    exit 1
fi

echo "✅ Backend y Frontend activos"
echo ""
```

```
# Ejecutar tests
cd frontend/tests/selenium
pytest -v --tb=short

echo ""
echo "✅ Tests completados"
```

Hacer ejecutable:

```
$ chmod +x run_tests.sh
```

Comandos de Ejecución

1. Iniciar Backend (Terminal 1)

```
$ cd backend $ python app.py
```

2. Iniciar Frontend (Terminal 2)

```
$ cd frontend $ npm run dev
```

3. Ejecutar Tests (Terminal 3)

Ejecutar todos los tests:

```
$ ./run_tests.sh
```

O manualmente:

```
$ cd frontend/tests/selenium $ pytest -v
```

Test específico:

```
$ pytest test_navigation.py -v
```

Test individual dentro de clase:

```
$ pytest test_upload.py::TestUpload::test_upload_single_image -v
```

Con output detallado:

```
$ pytest test_processing.py -v -s
```

Con marcadores:

```
$ pytest -m slow -v
```

Interpretación de Resultados

Salida Exitosa:

```
test_navigation.py::TestNavigation::test_landing_page_loads PASSED    [ 25%]
test_navigation.py::TestNavigation::test_navigate_to_processor PASSED [ 50%]
test_upload.py::TestUpload::test_file_input_exists PASSED             [ 75%]
test_upload.py::TestUpload::test_upload_single_image PASSED           [100%]

========================== 4 passed in 15.23s ==========================
```

Salida con Errores:

```
test_navigation.py::TestNavigation::test_landing_page_loads FAILED    [ 25%]

FAILED test_navigation.py::TestNavigation::test_landing_page_loads
AssertionError: assert 'Procesamiento' in ''

========================== 1 failed, 3 passed in 12.45s ================
```

10. Mejores Prácticas

Principios Fundamentales

1. Tests Independientes

Cada test debe poder ejecutarse de forma aislada sin depender del orden de ejecución.

Bueno:

```
def test_upload(self, driver, base_url):
    driver.get(base_url)
    # Navegar al procesador
    # Cargar archivo
    # Verificar
```

Malo:

```
def test_upload(self, driver):
    # Asume que ya estás en el procesador
    # Cargar archivo
```

2. Esperas Explícitas

Usar `WebDriverWait` en lugar de `time.sleep()` .

Bueno:

```
WebDriverWait(driver, 10).until(
    EC.presence_of_element_located((By.CLASS_NAME, "result"))
)
```

Malo:

```
time.sleep(5)  # Puede ser muy corto o muy largo
```

3. Selectores Robustos

Preferir selectores por ID o clases estables sobre XPath complejos.

Tipo	Robustez	Ejemplo
ID	Muy alta	`By.ID, "upload-btn"`
Clase	Alta	`By.CLASS_NAME, "process-btn"`
CSS	Media	`By.CSS_SELECTOR, "button.primary"`
XPath	Baja	`By.XPATH, "//div[3]/button[2]"`

4. Limpieza de Recursos

Siempre cerrar el driver después de cada test.

```
@pytest.fixture
def driver(chrome_options):
    driver = webdriver.Chrome(options=chrome_options)
    yield driver
    driver.quit()  # Siempre se ejecuta
```

5. Nombres Descriptivos

Los nombres de tests deben describir exactamente qué verifican.

Bueno:

- `test_upload_single_image_shows_counter`
- `test_background_removal_toggle_changes_state`

Malo:

- `test_1`

- `test_button`

Organización de Tests

Patrón AAA (Arrange-Act-Assert)

```
def test_example(self, driver, base_url):
    # ARRANGE: Preparar
    driver.get(base_url)
    button = driver.find_element(By.ID, "submit")

    # ACT: Actuar
    button.click()

    # ASSERT: Verificar
    result = driver.find_element(By.CLASS_NAME, "result")
    assert "Success" in result.text
```

Manejo de Errores Comunes

TimeoutException

```
try:
    element = WebDriverWait(driver, 10).until(
        EC.presence_of_element_located((By.ID, "result"))
    )
except TimeoutException:
    pytest.fail("Elemento no encontrado después de 10 segundos")
```

StaleElementReferenceException

```
# Re-localizar el elemento si la página cambió
def find_element_safe(driver, locator):
    for _ in range(3):
        try:
            return driver.find_element(*locator)
        except StaleElementReferenceException:
            time.sleep(0.5)
    raise
```

11. Solución de Problemas

Problemas Comunes y Soluciones

1. "WebDriver executable needs to be in PATH"

Error: No se encuentra el driver de Chrome.

Solución:

```
pip install webdriver-manager
```

Y usar en conftest.py:

```
from webdriver_manager.chrome import ChromeDriverManager
service = Service(ChromeDriverManager().install())
```

2. "Unable to locate element"

Posibles Causas:

- Elemento no ha cargado todavía
- Selector incorrecto
- Elemento en iframe
- Elemento oculto

Solución:

```
WebDriverWait(driver, 10).until(
    EC.visibility_of_element_located((By.ID, "element"))
)
```

3. Tests Lentos

Optimizaciones:

- Usar modo headless para Chrome
- Reducir implicit_wait si es muy alto
- Ejecutar tests en paralelo con pytest-xdist

Modo headless en conftest.py:

```
options.add_argument("--headless")
```

Ejecución paralela:

```
pip install pytest-xdist
pytest -n 4  # 4 workers en paralelo
```

4. "Connection refused" al ejecutar tests

Causa: Backend o Frontend no están corriendo.

Solución:

1. Verificar backend: `curl http://localhost:5000/api/health`
2. Verificar frontend: `curl http://localhost:5173`
3. Iniciar servicios faltantes

5. Tests fallan esporádicamente

> **Causa:** Race conditions o timing issues.
>
> **Solución:**
>
> - Aumentar timeouts
> - Usar esperas explícitas
> - Verificar que elementos estén realmente visibles
>
> ```
> WebDriverWait(driver, 15).until(
>     EC.visibility_of_element_located((By.CLASS_NAME, "result"))
> )
> ```

Debugging de Tests

Screenshots en Fallos

```
@pytest.fixture
def driver(chrome_options, request):
    driver = webdriver.Chrome(options=chrome_options)
    yield driver

    # Capturar screenshot si el test falló
    if request.node.rep_call.failed:
        driver.save_screenshot(f"screenshots/{request.node.name}.png")

    driver.quit()
```

Modo Interactivo

```
import pdb; pdb.set_trace()  # Pausar ejecución
# O usar pytest con:
pytest --pdb  # Pausa automática en fallos
```

Logs Detallados

```
import logging
logging.basicConfig(level=logging.DEBUG)
```

```
# En el test:
print(f"URL actual: {driver.current_url}")
print(f"Título: {driver.title}")
print(f"Page source: {driver.page_source[:200]}")
```

12. Anexos

A. Checklist de Implementación

Paso	Tarea	Completado
1	Instalar dependencias (requirements-test.txt)	☐
2	Crear estructura de carpetas tests/selenium/	☐
3	Configurar conftest.py	☐
4	Implementar test_navigation.py	☐
5	Implementar test_upload.py	☐
6	Implementar test_processing.py	☐
7	Implementar test_download.py	☐
8	Configurar pytest.ini	☐
9	Crear run_tests.sh	☐
10	Ejecutar y verificar todos los tests	☐

B. Comandos Rápidos de Referencia

Acción	Comando
Instalar dependencias	`pip install -r requirements-test.txt`

Iniciar backend	`python backend/app.py`
Iniciar frontend	`npm run dev`
Ejecutar todos los tests	`pytest -v`
Test específico	`pytest test_navigation.py -v`
Con output detallado	`pytest -v -s`
Pausar en fallos	`pytest --pdb`
Modo headless	Descomentar en conftest.py
Ejecutar en paralelo	`pytest -n 4`

C. Selectores CSS Útiles

Selector	Descripción	Ejemplo
`By.ID`	Por ID único	`"upload-btn"`
`By.CLASS_NAME`	Por clase CSS	`"process-btn"`
`By.CSS_SELECTOR`	Selector CSS complejo	`"input[type='file']"`
`By.XPATH`	Expresión XPath	`"//button[text()='Procesar']"`
`By.TAG_NAME`	Por etiqueta HTML	`"button"`
`By.NAME`	Por atributo name	`"username"`

D. Condiciones de Espera Comunes

```
from selenium.webdriver.support import expected_conditions as EC

# Presencia del elemento
EC.presence_of_element_located((By.ID, "element"))

# Elemento visible
EC.visibility_of_element_located((By.CLASS_NAME, "result"))

# Elemento clickeable
EC.element_to_be_clickable((By.ID, "submit"))

# Texto presente en elemento
EC.text_to_be_present_in_element((By.ID, "msg"), "Success")

# URL contiene
EC.url_contains("processor")

# Título contiene
EC.title_contains("ImageProcessor")
```

E. Recursos Adicionales

- **Documentación Selenium:** https://selenium-python.readthedocs.io/
- **Documentación Pytest:** https://docs.pytest.org/
- **Selenium Expected Conditions:** https://selenium-python.readthedocs.io/waits.html
- **XPath Tutorial:** https://www.w3schools.com/xml/xpath_intro.asp
- **CSS Selectors:** https://www.w3schools.com/cssref/css_selectors.asp

F. Ejemplo de Reporte de Test

```
============================= test session starts ==============================
platform linux -- Python 3.10.0, pytest-7.4.3, pluggy-1.3.0
rootdir: /path/to/frontend/tests
plugins: selenium-4.0.1
collected 12 items

test_navigation.py ....                                                  [ 33%]
test_upload.py ...                                                       [ 58%]
test_processing.py ....                                                  [ 91%]
test_download.py .                                                       [100%]

============================== 12 passed in 45.23s ==============================
```

> **¡Felicitaciones!**
> Has completado la guía de pruebas Selenium para ImageProcessor. Ahora tienes un conjunto completo de tests automatizados que verifican todo el flujo de tu aplicación.

Fin del Documento

ImageProcessor - Pruebas Selenium

Versión 1.0