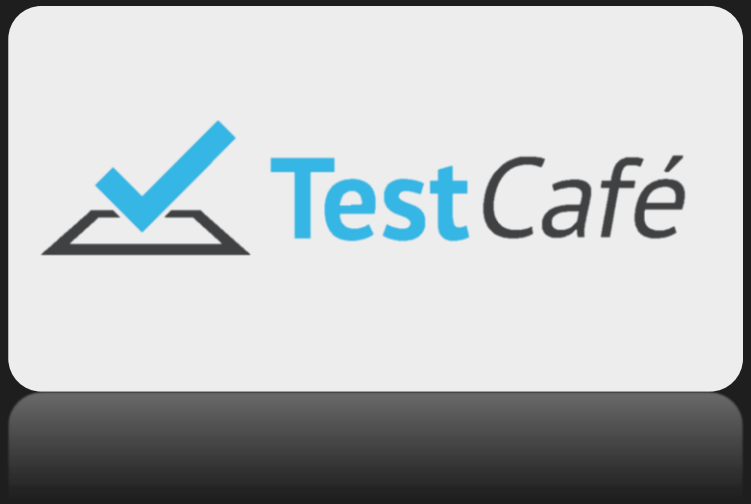
Taller de Automatización y testing web con TestCafe



Por: Luis Manuel Arellano Muñoz

QR Repositorio

Agenda del Taller

1. Repaso y Resolución de Dudas
2. Selectores y Aserciones
3. Patrones y Buenas Prácticas
4. Proyecto Final Guiado
5. Referencias

1.1 Repaso sesión 1

1

Introducción al Testing E2E

Conceptos y fundamentos

2

Herramientas de Automatización

Comparativa y TestCafe

3

Instalación y Configuración

Node.js, NPM y TestCafe

4

Primera Práctica Guiada

Hola Mundo y Login

☑ **Fundamentos completados**

¡Listos para profundizar en la Sesión 2!

2.1 Selectores

MAL - Frágil (dependiente de estructura)

```
Selector('div > span:nth-child(3) > button');
```

REGULAR - Por tag o clase CSS

```
Selector('button');
```

```
Selector('.btn-primary');
```

BUENO - Por ID (específico)

```
Selector('#submit-button');
```

MUY BUENO - Por atributo personalizado

```
Selector('[data-qa="submit-button"]');
```

EXCELENTE - data-testid (estándar)

```
Selector('[data-testid="login-submit"]');
```

Recomendación:
Usa data-testid para tests estables que no cambien con el diseño

2.2 Aserciones

.ok()

Verifica que una condición sea verdadera

```
await t.expect(Selector('.button').exists).ok();
```

.notOk()

Verifica que una condición NO sea verdadera (negación)

```
await t.expect(Selector('.error').exists).notOk();
```

.contains()

Verifica que un texto contenga una subcadena específica

```
await t.expect(Selector('.title').innerText).contains('Bienvenido');
```

.eql()

Compara valores exactos (igualdad estricta)

```
await t.expect(Selector('.count').innerText).eql('10');
```

.exists

Propiedad que verifica si un elemento existe en el DOM

```
await t.expect(Selector('.welcome').exists).ok();
```

.count

Cuenta cuántos elementos coinciden con el selector

```
await t.expect(Selector('.item').count).eql(5);
```

💡 **Las aserciones son el corazón de tus tests: validan que todo funcione correctamente**

2.3 Acciones

.click()

Hace clic en un elemento (botón, enlace, etc.)

```
await t.  click  (Selector(        '#submit-button'    ));
```

.typeText()

Escribe texto en un campo de entrada

```
await t.  typeText    (Selector(        '#username'    ),   'usuario123'    );
```

.selectText()

Selecciona texto en un campo (útil para reemplazar)

```
await t.  selectText    (Selector(        '#email'   ));
```

.hover()

Mueve el cursor sobre un elemento (simula mouse over)

```
await t.  hover  (Selector(        '.menu-item'     ));
```

.doubleClick()

Hace doble clic en un elemento

```
await t.  doubleClick    (Selector(        '.file' ));
```

.pressKey()

Simula pulsaciones de teclas (Enter, Tab, Esc, etc.)

```
await t.  pressKey   ( 'enter'  );
```

💡 **Las acciones simulan interacciones reales del usuario con tu aplicación**

2.4 Otros comandos útiles

.wait()

Pausa la ejecución por un tiempo específico (en milisegundos)

```
await t.wait(3000);              // Espera 3 segundos
```

.navigateTo()

Navega a una URL específica durante el test

```
await t.navigateTo('https://ejemplo.com/login');
```

.takeScreenshot()

Captura una imagen de la página actual (útil para evidencia)

```
await t.takeScreenshot('pantalla-error.png');
```

.setFilesToUpload()

Sube archivos a un input de tipo file

```
await t.setFilesToUpload('#upload', ['./file.pdf']);
```

.switchToIframe()

Cambia el contexto a un iframe para interactuar con él

```
await t.switchToIframe('#payment-iframe');
```

.rightClick()

Hace clic derecho en un elemento (menús contextuales)

```
await t.rightClick(Selector('.menu-item'));
```

↔ .drag()

Arrastra un elemento a una posición específica (drag & drop)

```
await t.drag('.draggable', 100, 50);
```

Nota:

Estos comandos te permiten manejar escenarios más complejos en tus pruebas

3.1 Patrones

Page Object Model (POM)

Encapsula elementos y acciones de páginas.
Mejora mantenibilidad y reutilización.

AAA (Arrange-Act-Assert)

Preparar → Ejecutar → Verificar.
Tests legibles y bien estructurados.

Data-Driven Testing

Mismo test con diferentes datos.
Valida múltiples escenarios sin duplicar.

Fixtures y Hooks

Setup y cleanup automáticos para tests.
Prepara y limpia estado entre pruebas.

DRY (Don't Repeat Yourself)

Crea funciones auxiliares reutilizables.
Reduce mantenimiento y errores en tests.

Estos patrones mejoran la calidad, mantenibilidad y escalabilidad

3.2 Buenas Practicas

✅ Pruebas End-to-End

Simula escenarios reales de usuario, no componentes individuales

⚠ NO uses await en Selectors

Usa el Selector sin await para mantener la query reactiva

Implementa Page Models

Agrupa Selectors y acciones en objetos reutilizables

Usa Roles para autenticación

Define usuarios una vez, cambia con: await t.useRole(admin)

Tests cortos y concisos

Más fáciles de depurar y pueden ejecutarse en paralelo

Estructura organizada

Carpetas: /tests, /page_model, /helpers, /roles, /data

Selectores confiables

Usa data-testid, no muy específicos ni genéricos

Hooks para setup/teardown

before/beforeEach para preparar, after/afterEach para limpiar

3.3 Problemas Comunes

❌ Problema: Usar await con Selectors en assertions

El valor queda congelado y no se actualiza

✅ Solución:

No uses await, guarda el Selector sin ejecutarlo

❌ Problema: Selectores frágiles que se rompen fácilmente

Muy específicos (.nth(5)) o muy genéricos (div > button)

✅ Solución:

Usa atributos data-testid y selectores con significado semántico

❌ Problema: Tests largos difíciles de mantener

Un test que hace muchas cosas es difícil de depurar

✅ Solución:

Divide en tests cortos, cada uno con un propósito claro

❌ Problema: Código duplicado en múltiples tests

Repetir los mismos selectores y acciones en cada test

✅ Solución:

Implementa Page Models para reutilizar selectores y métodos

❌ Problema: Estado persistente entre tests

Datos de un test afectan a otros (cookies, localStorage, archivos)

✅ Solución:

Usa hooks beforeEach/afterEach para limpiar el estado

❌ Problema: Comparar valores estáticos en assertions

Comparar dos valores fijos no valida el comportamiento real

✅ Solución:

Compara el estado actual de la aplicación vs. expectativas

4. Practica Final

El Concepto

Tienes las clases POM ya listas (PaginaLogin.js y PaginaProductos.js)

Tú solo ensamblas los métodos como piezas de LEGO en tu test

Piezas que ya tienes (NO las modifiques)

paginas/PaginaLogin.js

paginas/PaginaProductos.js

Tu trabajo: Completar pruebas/login.test.js

TEST 1: Login exitoso

TEST 2: Login fallido

TEST 3: Agregar productos

Ejemplo: Así ensamblas las piezas

```
import paginaLogin     from   '../paginas/PaginaLogin';
test  ( 'Login exitoso'       , async t => {
   // ARRANGE - Preparar datos
   const   usuario =      'standard_user'       ;
   const   contrasena =       'secret_sauce'       ;
   // ACT - Ejecutar acción
   await    paginaLogin.     hacerLogin    (usuario, contrasena);
```

Credenciales para SauceDemo

Usuario: `standard_user`

Password: `secret_sauce`

Solo ensambla, ¡no escribas desde cero!

5. Referencias

Documentación Oficial

Documentación Principal

testcafe.io/documentation

Guías completas, API y ejemplos

Getting Started

testcafe.io/documentation/getting-started

Primeros pasos y tutoriales básicos

API Reference

testcafe.io/documentation/reference

Referencia completa de APIs y métodos

Recipes and Examples

testcafe.io/documentation/recipes

Ejemplos prácticos y casos de uso

¡Continúa tu viaje en automatización de pruebas!