

«Talento Tech»

# Automation Testing

Clase 6



# Clase N° 6: DOM para Automatización

## Temario

- ¿Qué es el DOM y por qué nos importa en QA?
- Recorrido del DOM con selectores CSS
- XPath: otra vía para llegar al elemento correcto
- Estrategias de localización: del `id` al XPath relativo
- Ejercicios

## Objetivos de la clase

En esta clase profundizaremos en la navegación y selección de elementos dentro del DOM (Document Object Model), una habilidad clave para la automatización efectiva. Aprenderemos a visualizar la estructura en árbol que el navegador genera a partir del HTML y a recorrerla utilizando selectores CSS, desplazándonos entre padres, hijos y elementos hermanos. Además, construiremos expresiones XPath más precisas, aplicando filtros por atributos y contenido de texto. Analizaremos cómo elegir el selector más robusto y sostenible según el contexto, evitando errores comunes en entornos dinámicos. Como cierre, comenzaremos a documentar nuestros selectores para facilitar su futura reutilización en scripts de automatización con herramientas como Selenium.

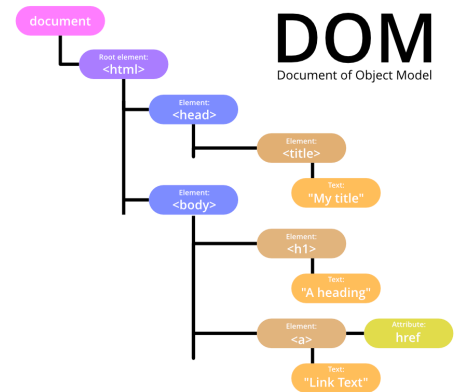
# ¿Qué es el DOM?

Hasta ahora vimos HTML y CSS como si la página fuera un archivo estático. Pero cuando el navegador la procesa, se convierte en algo vivo: el **DOM**. Entender cómo está organizado ese árbol de nodos (y cómo construir selectores fiables) es la piedra angular para automatizar pruebas en la interfaz. Hoy vamos a zambullirnos en el DOM, practicar con selectores CSS y XPath, y terminar escribiendo nuestros propios localizadores para la aplicación de TalentoLab.

El **Document Object Model** es la representación en forma de árbol que el navegador crea a partir del HTML. Cada etiqueta se convierte en un **nodo** con propiedades y métodos. Gracias al DOM, JavaScript (y por extensión, Selenium) puede:

- Recorrer la jerarquía (`parentNode`, `children`).
- Consultar/Modificar contenido (`textContent`, `innerHTML`).
- Escuchar eventos (`addEventListener`).

💡 Imagina el HTML como el plano de una casa y el DOM como la casa construida: ahora puedes tocar puertas, encender luces o derribar muros.



## ¿Por qué el DOM es clave en la automatización de pruebas?

Cuando usás herramientas como **Selenium**, **Playwright** o **Puppeteer**, no estás interactuando directamente con el HTML como archivo, sino con el **DOM**: ese árbol de nodos en tiempo real que el navegador construye a partir del HTML.

## ¿Qué significa esto?

Automatizar una prueba en la interfaz (UI) es, en esencia, decirle al navegador:

“Encontrá este botón, hacé clic. Completá este input con texto. Verificá que aparezca un mensaje”. Y para que eso funcione, **necesitás seleccionar correctamente los elementos del DOM**. Es ahí donde entran:

- **Selectores CSS** → por `id`, `class`, combinaciones jerárquicas (`div > input`, etc.).
- **XPath** → cuando los elementos no tienen atributos claros o estables.

✓ Dominar el DOM y los selectores es lo que convierte a una persona que *automatiza* cosas en alguien que **construye pruebas estables y confiables**

## Mini-mapa de nodos

Ya vimos que el DOM (Document Object Model) es una **representación en árbol** del HTML. Cada etiqueta se convierte en un **nodo**, y la forma en que estos nodos se organizan es lo que determina cómo los vamos a encontrar desde nuestros scripts de automatización.

```
<body>
  <div id="login">
    <label>Correo</label>
    <input name="email" />
    <button class="btn enviar">Enviar</button>
  </div>
</body>
```

- `<body>` → padre raíz visible.
- `<div id="login">` → hijo de `<body>`.
- `<label>`, `<input>`, `<button>` → hermanos, hijos de `div`.

Con DevTools (F12) puedes plegar/desplegar estos nodos y ver la relación padre-hijo en tiempo real.

## Recorrido del DOM con selectores CSS

Los **selectores CSS** no solo se usan para dar estilo. También son una de las herramientas más poderosas en **automatización de pruebas**, porque te permiten **localizar elementos dentro del DOM** de forma clara, rápida y precisa.

### ¿Cómo funcionan?

Los selectores CSS se leen **de izquierda a derecha**, como si estuvieras siguiendo un camino desde un nodo padre hacia sus hijos. Cuanto mejor entiendas la jerarquía del DOM, más preciso será tu selector.

Selector	Significado	Ejemplo práctico
<code>#email</code>	Elemento con <code>id="email"</code>	Único y directo.
<code>.btn.enviar</code>	Elemento con dos clases <code>btn</code> y <code>enviar</code>	<code>&lt;button class="btn enviar"&gt;</code>
<code>form</code> <code>input[name="password"]</code>	<code>input</code> con <code>name="password"</code> dentro de un <code>form</code>	Reduce riesgo de confundir con otro <code>input</code> .



<code>label + input</code>	<code>input hermano inmediato de un label</code>	Útil cuando no hay id.
<code>div &gt; p</code>	<code>p hijo directo de un div</code>	Evita saltar niveles innecesarios.

💡 **Regla de oro:** usa el selector más corto que sea a la vez único y estable. El 80 % de las veces, un buen `id` o `name` basta.

### Ejemplo:

✅ `#email` (claro, directo)

❌ `body > div:nth-child(2) > form > input[type='text']` (frágil y largo)

## Acceso a nodos con JavaScript (métodos clásicos y modernos)

Una vez que comprendiste cómo está estructurado el DOM, el siguiente paso es **acceder a esos nodos desde JavaScript**, algo fundamental tanto para automatizar como para validar el comportamiento de una interfaz.

JavaScript ofrece dos grandes familias de métodos:

### A) Métodos clásicos `get*`

A continuación verás **cuatro métodos tradicionales** para acceder a nodos. Cada ejemplo muestra el **HTML original**, el **JavaScript** que lo captura y modifica, y el **estado final** (comentado) para que visualices el efecto.

#### 1. `getElementById(id)`

**HTML de partida:**

```
<div id="mensaje">Bienvenido</div>
```

**JavaScript:**

Creá un archivo llamado [funcion.js](#) con este contenido:

```
const caja = document.getElementById("mensaje"); // captura el div
caja.textContent = "¡Hola, TalentoLab!"; // lo modifica
```

**Resultado en el DOM:**

```
<div id="mensaje">¡Hola, TalentoLab!</div> <!-- texto cambiado -->
```

✓ Útil cuando el elemento tiene un **id** único y confiable.

## 2. `getElementsByClassName(clase)`

HTML inicial:

```
<body>
  <button class="btn">Guardar</button>
  <button class="btn">Cancelar</button>
</body>
```

JavaScript:

```
const botones = document.getElementsByClassName("btn"); //
HTMLCollection
for (const b of botones) {
  b.style.backgroundColor = "#28a745"; // pinta ambos botones de verde
}
```

Estado final: los dos botones comparten el nuevo color.

✓ Ideal para aplicar cambios a muchos elementos con el mismo estilo o comportamiento.

## 3. `getElementsByName(name)`

HTML:

```
<body>
  <input name="genero" type="radio" value="m" /> Masculino
  <input name="genero" type="radio" value="f" /> Femenino
</body>
```

JavaScript:

```
const radios = document.getElementsByName("genero");
radios[0].checked = true; // marca el primero
```

Resultado: el radio “Masculino” aparece seleccionado. **Útil** en formularios donde el backend se basa en **name** para procesar datos.

✓ Muy usado cuando el backend espera datos con name.

#### 4. getElementsByTagName(tag)

HTML antes:

```
<ul>
  <li>Uno</li>
  <li>Dos</li>
</ul>
```

JavaScript:

```
const items = document.getElementsByTagName('li');
items[1].style.fontWeight = 'bold'; // resalta el segundo ítem
```

DOM después:

```
<li style="font-weight: bold;">Dos</li> <!-- ahora en negrita -->
```

Perfecto para recorrer o contar todos los elementos de un tipo (inputs, filas de tabla, etc.).

**Recuerda:** métodos **get\*** **obtienen** nodos; sus homólogos **set\*** (cuando existen) **asignan** o modifican valores.

✓ Útil para recorrer listas, inputs o filas de tabla.

### B) Métodos modernos basados en selectores CSS basados en selectores CSS

#### 5. querySelector(selector)

Devuelve **el primer** elemento que coincide o **null**.

```
const btn = document.querySelector('.btn.enviar');
```

#### 6. querySelectorAll(selector)

Devuelve **todos** los que coinciden (NodeList estática).

```
const page = document.querySelector('#page'); // id
const info = document.querySelector('.main .info'); // clase
dentro de otra
const inputs = document.querySelectorAll('form input'); // todos los
inputs del formulario
```

**querySelector** simplifica búsquedas complejas que con los métodos clásicos requerirían varias líneas.

## Crear y agregar nodos:

```
// Crear un botón desde cero y añadirlo al body
const nuevoBtn = document.createElement('button');
const txt      = document.createTextNode('Click aquí');

nuevoBtn.appendChild(txt);           // <button>Click aquí</button>
document.body.appendChild(nuevoBtn);
```

## innerHTML vs textContent

- **innerHTML** lee o escribe *HTML* dentro del elemento (interpreta etiquetas).
- **textContent** maneja **solo texto plano** (más seguro contra XSS).

```
div.innerHTML = '<strong>Hola</strong>'; // inserta negrita
div.textContent = '<strong>Hola</strong>'; // muestra los caracteres
tal cual
```

Con estas herramientas puedes capturar, crear o modificar nodos; estos mismos principios se aplicarán cuando tus herramientas de automatización interactúen con la página.

## ¿Cuándo usar cuál?

Necesitás...	Usá este método
Un elemento con <b>id</b> único	<code>getElementById</code> o <code>querySelector('#id')</code>
Todos los inputs de un form	<code>querySelectorAll('form input')</code>
Todos los botones con clase	<code>getElementsByClassName('btn')</code>
Todos los elementos con <code>name="email"</code>	<code>getElementsByName('email')</code>
Algo más flexible y potente	<code>querySelector</code> / <code>querySelectorAll</code>



# XPath: la navaja suiza

Cuando estás automatizando y **no podés usar `id`, `class` o `name` porque no existen o no son únicos**, entra en juego **XPath**: un lenguaje que te permite navegar por el DOM como si fuera una estructura de carpetas.


## Anatomía de una ruta XPath

```
//div[@class='card'][1]/header/h2
```

Parte	Significado
//	Buscar en cualquier parte del documento
div[@class='card'][1]	Primer div con clase "card"
/header/h2	Bajá a <header> y luego a <h2>

## Patrones imprescindibles

Escenario	XPath	Explicación rápida
Por atributo	<code>//input[@type='email']</code>	Todos los input de tipo email.
Por texto exacto	<code>//button[text()='Guardar']</code>	Botón cuyo texto visible es "Guardar".
Por texto parcial	<code>//p[contains(text(),'éxito')]</code>	Cualquier <p> que contenga la palabra "éxito".
Relativo a otro nodo	<code>//label[.='Usuario']/following::input[1]</code>	Primer input que sigue al label con texto "Usuario".
Por posición	<code>((//tr[@class='fila']))[last()]</code>	Última fila con clase fila en una tabla.
Múltiples atributos	<code>//*[@data-id='99' and @data-role='delete']</code>	Elemento con dos atributos específicos.

 **Tip:** Puedes probar tus XPaths en la pestaña *Console* con `$x(' //xpath')` y ver si devuelve el nodo esperado.

## Evita el camino absoluto

Un XPath absoluto (`/html/body/div[2]/div[3]/button`) es como dar direcciones diciendo “sube dos pisos y cuenta tres puertas”: basta mover una pared para perderte. **Prefiere XPaths relativos** que empiecen con `//` y usen atributos o texto que no cambie con el diseño.

## Buenas prácticas al combinar CSS y XPath

Elige siempre el selector más **estable**, **claro** y **corto**. Por ejemplo, si existe `#email`, usa ese CSS; si necesitas ubicar un botón por su texto visible, un XPath como `//button[text()='Enviar']` puede ser la opción.

## Casos de estudio rápidos.

### Error debajo del campo email

```
<input id="email" />
<span class="error">Correo inválido</span>
```

1. *XPath:*  
`//input[@id='email']/following-sibling::span[@class='error']`

### Checkbox “Acepto términos”

```
<label><input type="checkbox" /> Acepto términos</label>
```

2. *XPath:* `//label[contains(., 'Acepto')]/input[@type='checkbox']`

### Última fila de resultados

```
<table id="res">
  <tr class="row">...</tr>
  <tr class="row">...</tr>
</table>
```

3. *XPath:* `//table[@id='res']//tr[@class='row'][last()]`

Con práctica, XPath se convertirá en tu comodín cuando te topes con páginas poco “amigables” para la automatización.

## Estrategias de localización

1. **id único** → `#user-email`.
2. **name** → `input[name='password']`.
3. **Clase + jerarquía** → `.form-login input[type='checkbox']`.
4. **Atributo personalizado** → `[data-testid='signup-btn']`.
5. **XPath** por texto, posición o relación.

Si tu selector sobrevive a un cambio de estilo y no depende de la posición de un elemento, has ganado la mitad de la batalla contra la fragilidad de los tests.

# ¡Manos a la obra en TalentoLab!

Durante el último sprint afinaste tu maqueta HTML/CSS y escribiste los primeros test unitarios en Pytest. Ahora el **equipo Front-End** anuncia una **refactorización profunda del formulario de registro** del portal interno de perfiles. Como QA te corresponde proteger la automatización futura: si mañana cambian colores o posiciones, **tus selectores no deben romperse**.



## Avisos que llegan al canal #qa



**Silvia (PO):** “Mañana rediseñan la pantalla. Necesito una lista de selectores que **no dependan del layout**.”

Los selectores independientes del layout se apoyan en algo que no cambia con el diseño, como un id (`#email`), un name (`input[name="password"]`) o el texto de un botón (`//button[text()='Guardar']`).

En cambio, los que dependen del layout describen posiciones o clases de maquetado: `body > div:nth-child(3) > form > input` o `/html/body/div[2]/button`; basta mover un `<div>` o renombrar una clase para que dejen de funcionar.

## Requisitos funcionales:

Concentrarse en los 4 campos obligatorios (**nombre, apellido, correo, contraseña**) y el **botón Guardar**.

- Si existe un **id** único, úsalo (CSS).
- Si no, construye un CSS o un XPath **robusto**.
- Documenta porqué lo elegiste.”



“Seguí este paso a paso al pie de la letra y en 20 min tendrás la tabla lista.”

# Ejercicio práctico

1. Visita una demo pública: SauceDemo: <https://www.saucedemo.com/> o HerokuApp: <https://the-internet.herokuapp.com/login>
2. Obtén un selector CSS y un XPath relativo para:
  - a. input usuario
  - b. input contraseña
  - c. (extra) botón login/guardar.
3. Prueba cada selector en la consola: `document.querySelector()` y `$x()`.
4. Crea selectores-talento.md con la tabla:

Elemento	Selector elegido	Tipo	Razón

5. Crear un snippet de test masivo para verificar rápidamente si tus **selectores CSS apuntan a los elementos correctos**
6. (Bonus):Añade un snippet JS que cambie el `backgroundColor` de cada campo para demostrar que el selector apunta bien.

## Paso a paso:

### 1. Herramientas listas

- Abre la URL elegida.
- Pulsa **F12** → pestaña **Elements**.
- Activa el ícono del cursor para seleccionar nodos.

### 2. Selector CSS primero

1. Haz clic sobre el **input usuario**.
2. ¿Tiene `id="user-name"`? Perfecto: `#user-name`.



3. ¿No hay `id`? Busca un `name` o clase exclusiva:
  - `input[name="username"]`
  - `.login_input[type="text"]`
4. Repite con **contraseña** y **botón**.

### 3. XPath de respaldo

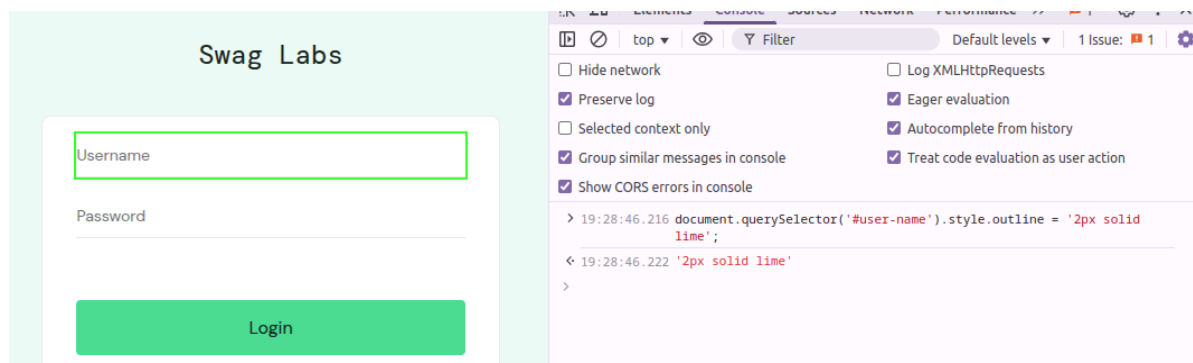
- Por `id`: `//*[@id='user-name']`
- Por `name`: `//input[@name='password']`
- Por texto del botón: `//button[normalize-space()='Login']`

🛑 Nunca uses rutas absolutas largas — si el Front mueve un `<div>` se rompe todo.

### 4. Verifica en Console

```
document.querySelector('#user-name').style.outline = '2px solid lime';
```

Verás que se pone en color lima el borde del input de user-name



Sino también por XPath

```
$x("//*[@id='user-name']")[0].style.outline = '2px solid red';
```

Debe pintar **exactamente un** elemento. Si es `null` o selecciona varios, afina el selector.



## 5. Rellena [selectores-talento.md](#)

Ejemplo:

Elemento	Selector	Tipo	Razón
Usuario	#user-name	CSS	id único y semántico
Usuario	//*[@id='user-name']	XPath	respaldo si cambian clases
Contraseña	input[type="password"]	CSS	atributo único dentro del form
Contraseña	//input[@type='password']	XPath	sencillo y estable
Botón Guardar	.btn_action	CSS	clase exclusiva del botón
Botón Guardar	//input[@type='submit']	XPath	atributo estable, indep. del texto

## 6. Snippet de test masivo

Este snippet permite verificar rápidamente si tus **selectores CSS apuntan a los elementos correctos**. Recorre una lista de selectores y les cambia el fondo a color khaki. Si un campo no se pinta, es señal de que el selector está mal o no es único. Así confirmás visualmente que tu tabla de selectores funciona antes de usarla en Selenium.

```
[ '#user-name', 'input[type="password"]', '.btn_action' ].forEach(sel=>{  
  const el=document.querySelector(sel);  
  if (el) el.style.background='khaki';  
});
```

Ejecuta; si los tres campos se vuelven amarillos, ¡selector validado!



**Buenos Aires**  
*aprende*  
Agencia de Habilidades para el Futuro

**BA** Buenos  
Aires  
Ciudad