Locating Elements on a Webpage

# Comparing a DOM and AOM use case

Objective: on [Element.getBoundingClientRect() - Web APIs | MDN (mozilla.org)](https://developer.mozilla.org/en-US/docs/Web/API/Element/getBoundingClientRect), save a reference of the button “Get MDN Plus” to a database object. Client-side application should be able to use this reference to locate the button on different clients.

Graphical user interface, text, application

Description automatically generated

## DOM Use Case Review: Stonly

### Steps

1. User chose to build a tutorial on site
2. When creating a new step with element highlight, user is prompt to select an element on the page. In this case the button “Get MDN Plus”.
3. User may now go back to tutorial Editor and check the UI element that is selected.Graphical user interface, text, application, email

   Description automatically generated
4. As can be seen from the above screenshot, the CSS selector “.primary > .button-wrap” is used to identify the UI element user selected.
5. We may check the validity of this selector by going to browser devtools and search with the CSS selector and see that the button is indeed uniquely identified. Graphical user interface, text, application

   Description automatically generated

### Problems

Theoretically, if the page of topic remains static, the client-side application will be able to locate the same element as with the tutorial creator. However, obvious problems exist in this scheme:

#### Vulnerable to changes on the target website

Let’s take a look into the CSS selector Stonly created for us: “.primary > .button-wrap”. To understand this selector, check [CSS Selectors Reference (w3schools.com)](https://www.w3schools.com/cssref/css_selectors.php). For convenience, my translation of this selector in English is: “**All** child (elements that have the class ‘button-wrap’) of **all** (elements that have the class ‘primary’)”. Here’s an explanation for the functions of these classes: **button-wrap** is used for styling the text content (making the text centered in the button), **primary** is a styling class for buttons of primary functions on this page (prior to class secondary). As you might have observed, the CSS selector could return multiple elements. Even though the selector is unique for the page at the specific time when this doc is created, it is easy to for MDN to add another primary button “Get MDN Pro”. The new button is highly likely to reuse the code from the existing “Get MDN Plus” button, in which case, the selector will return both the two buttons, and the client-application will be confused – which should it highlight?

#### Maintaining the tutorials

Let’s say the above-described scenario happens. What will happen is frustrated users come to us to complain about our application does not work on the MDN page or it is giving wrong information. A most common way to fix it is for the tutorial creator or a developer in the team to manually inspect the problem and fix the CSS selector, so it is pointing to the right document again. This way of solving the problem is neither efficient or scalable in the sense that we need as much of content as possible to cover all webpages for a basic user experience.

#### Letting user know what is going on

Error messages in any webpages are important. They expose information to users about how to find a solution to the problem they are facing. For example, a useful message in the case when the application cannot find the button “Get MDN Plus” is “The system failed to locate the button ‘Get MDN Plus’ due to internal errors”. Looking at this message, a user will instantly know that the tutorial is directing to the button. However, if we are only storing the CSS selector, the only thing we could expose is that the CSS selector being used is “.primary > .button-wrap”, which makes zero sense to general users.

## AOM (Accessibility Object Model)

Docs on AOM and web accessibility:

* [Accessibility Object Model | aom (wicg.github.io)](https://wicg.github.io/aom/explainer.html)
* [Accessibility Object Model (wicg.github.io)](https://wicg.github.io/aom/spec/)
* [ARIA - Accessibility | MDN (mozilla.org)](https://developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA)
* [WICG/aom: Accessibility Object Model (github.com)](https://github.com/WICG/aom)

The DOM is a tree of html nodes, we use css selectors to locate the elements inside of this tree. The css selector contains identifiers and structural information in corresponding html doc for locating the node in the tree. However, due to the business nature of our platform and current goals, it is impossible/inefficient to provide the structural/identity information of a target node (e.g., scanning existing resources online and transform the resources into our model, existing resources are for human and do not provide such information).

Considering the limited information density the existing resources can provide, we could only retrieve descriptions and keywords in human languages. Thus, we are provided with the situation that we need to be able to locate the target node with a human readable-description and make the machine understand and locate the specific description.

Luckily, we are not in a brand-new field of study. After some research, the AOM came into our vision. The core concept of AOM, accessibility, has been there since the early years of the computer history. It was aimed to help people with disability to access modern technologies. Looking at the behavior of some accessibility tools, for example a screen reader, we were able to understand the powers and potentials behind accessibility. A typical screen reader from a developers’ point of view, provides the following functions:

* Highlight and circle a specific UI element of any windows application.
* Provide a description of the highlighted element (read it out) in human language.

With a deeper look, we understand that the description and element locating are specified by developers or built-in with the operating system. The screen reader receives a system window as an input and iterate through all the UI elements to produce a UI/audio effect to the end user. For example, the button “Get MDN Plus” in a screen reader’s point of view is “Button Get MDN Plus”. Looking back at the problem we were given, this is exactly/closely how one would describe with the button in human language. Could we reversely use this property to input the description into the system and get the element we want?

With this initial idea, we went into the research of utilization of such api.

The desktop part seems promising with existing apis available: [Windows Automation API overview - Win32 apps | Microsoft Learn](https://learn.microsoft.com/en-us/windows/win32/winauto/windows-automation-api-portal).

Graphical user interface, text, application

Description automatically generated

The AOM has been exposed to developers as shown in the above screenshot. However, the expected api usage has not been exposed to developers, since the main function of the current accessibility provided by chrome is to allow developers to ensure the accessibility of specific webpages.

An ongoing implementation of the AOM exposure has been found here: [chrome.automation - Chrome Developers](https://developer.chrome.com/docs/extensions/reference/automation/).