CSS Manager

Eshaq Jamdar

May 9, 2024

Abstract

This paper breaks down the main feature use of the plugin in, along with a general discussion of the use of WebExtensionAPI. From there, a discussion about how this program uses the programming features of Firefox plugins, such as the use of Content Scripts and background scripts and how both of them work together. Lastly, a discussion of what future improvements and or features to add to the plugin.

1 Introduction

CSS Manager is a Firefox plugin, developed in JavaScript, that allows the user to inspect a particular element on a web page and edit the styles, add onto the styles, save edited styles to local storage, load from storage, toggle back to the original and edited styles, and clear saved style edits from a page.

Bellow are some annotated screenshots of the overall process of editing the styles of an existing web page. Note that this only shows the main "Inspect Element" feature.

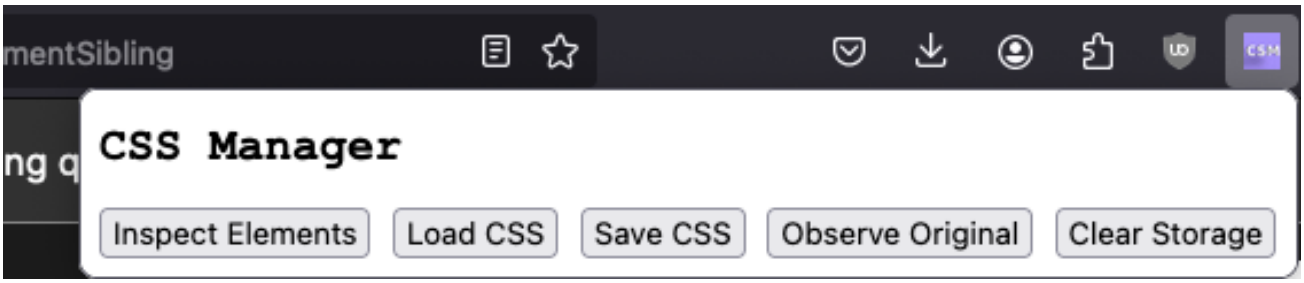

Figure 1: Main menu of the plugin when pressed. Inspect Elements will toggle the panel that allows the user to click and see elements that were applied, as well as add or turn off styles. Load and Save CSS buttons will store applied inline styles in local storage. Observe original will toggle back and forth between the original style of the page and the edited style. Lastly, Clear Storage button will clear the custom style stored in localstorage.

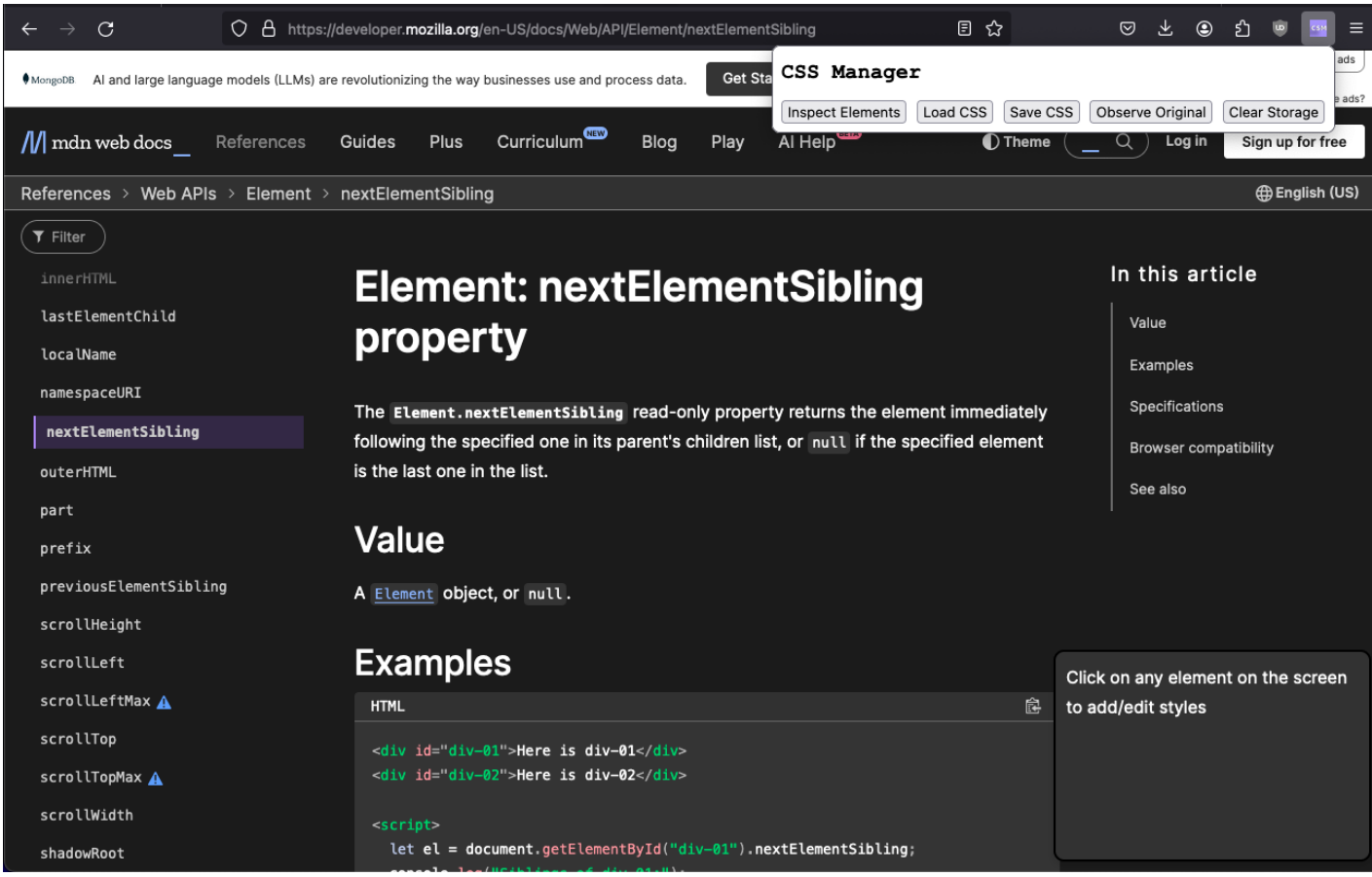

Figure 2: Toggling of the panel on MDN Docs. By default, the user will be instructed to click on an element to add or edit styles.

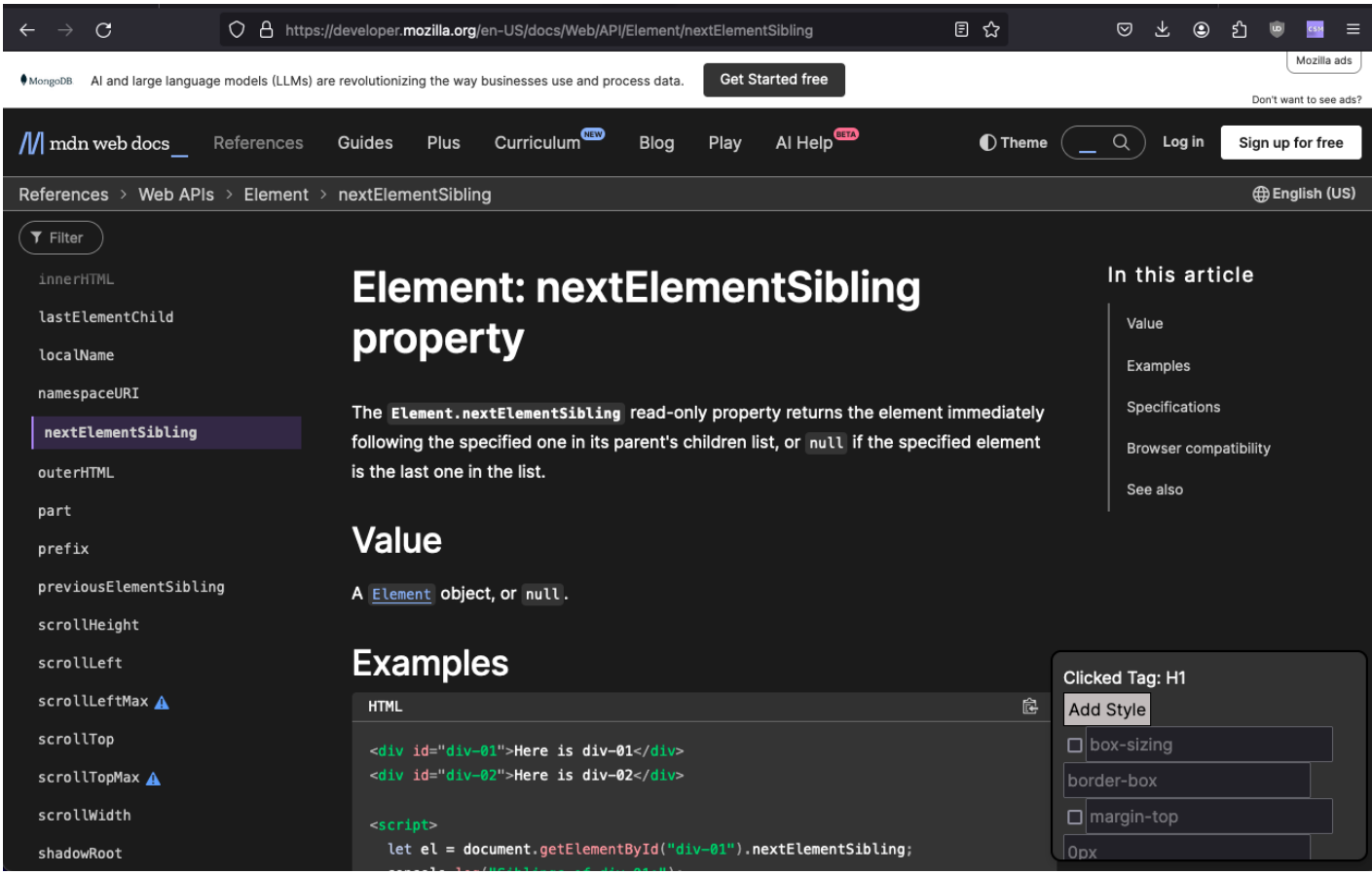

Figure 3: Panel showing the selection of H1 article title). The panel shows off applied styles obtained from the style sheets for that particular element. By default, all of the styles are disabled and if the user wanted to edit the values, they would simply toggle the check mark and edit either the specific property applied, the property value or both. They can also add an additional style if they want to add an extra style or if a particular element does not have any styles applied from the style sheets.

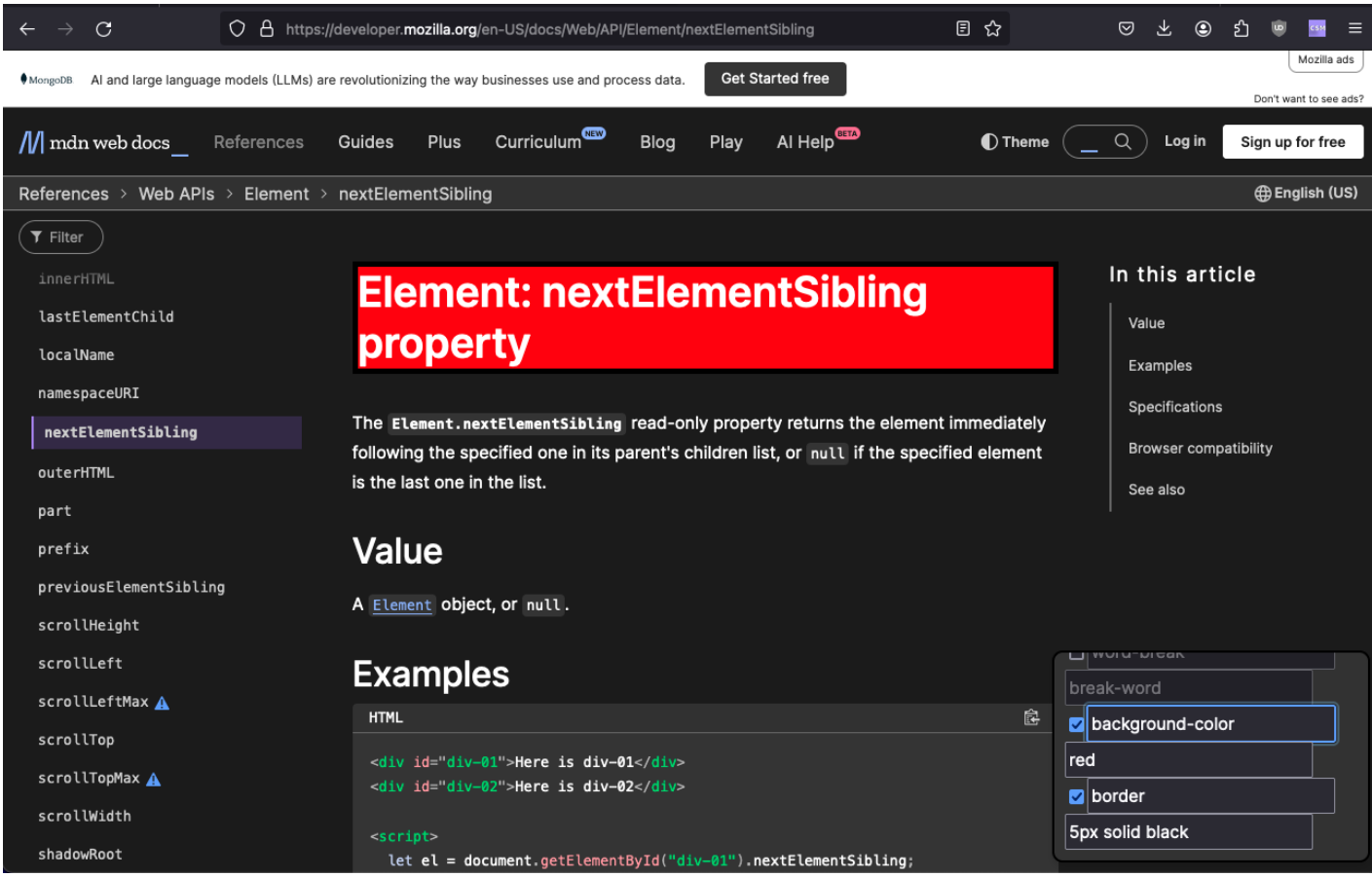

Figure 4: Adding of additional two styles onto the page: background-color of red and a 5px solid black border. All styles are applied as inline styles.

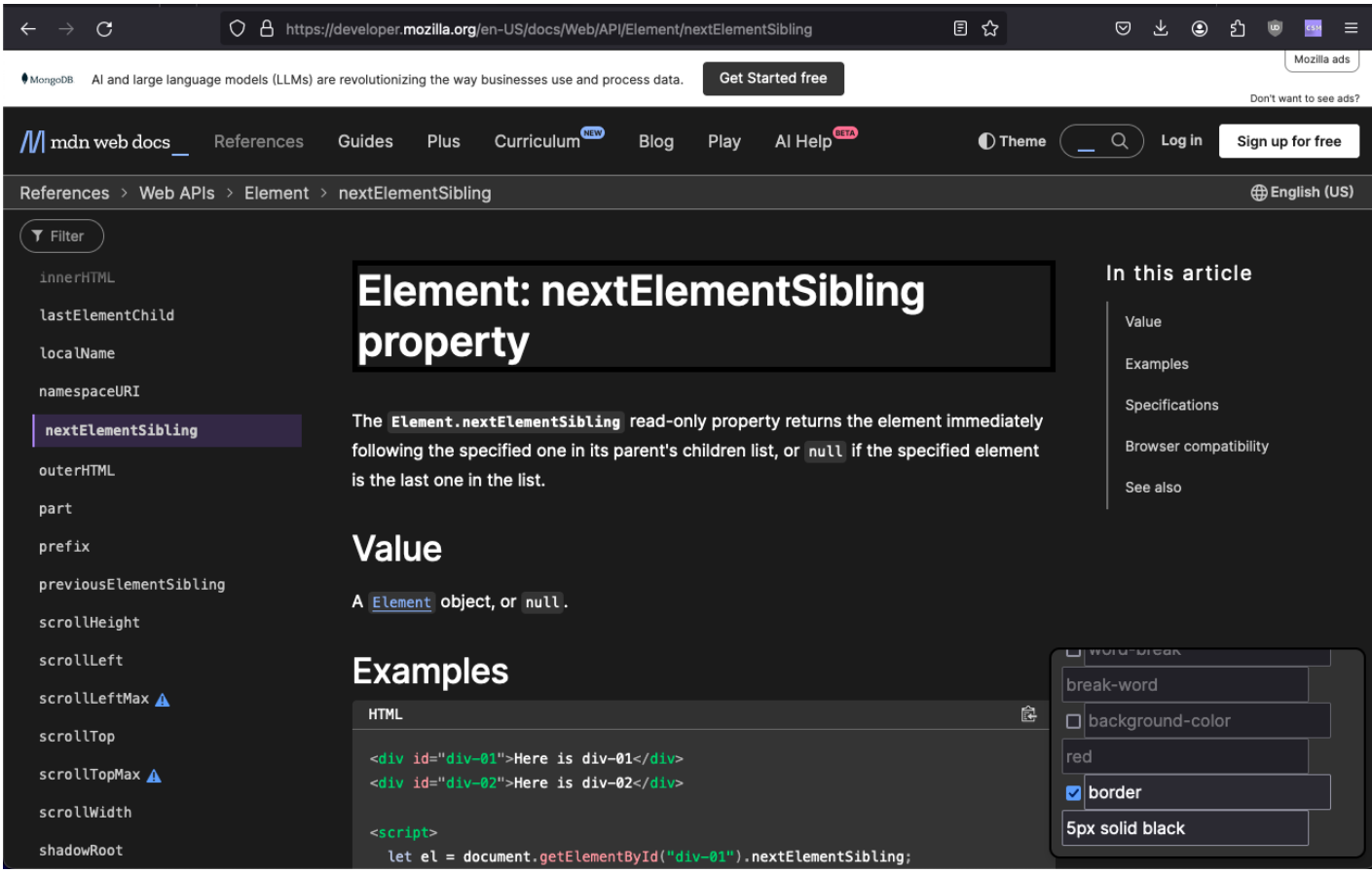

Figure 5: Continuation of the previous screenshot, here we can see the ability to further disable a style on the fly.

2 Organization of the Project

The code for this project was built on top of the structure provided from [Mozilla(2023a)].

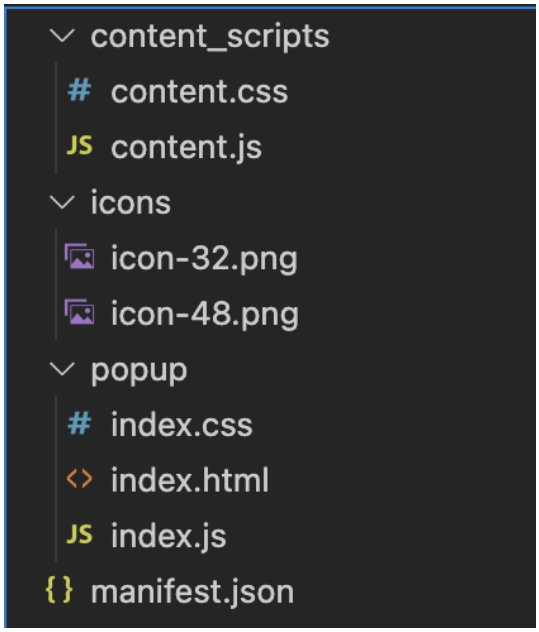

Figure 6: Organization of the project folder. The popup folder is the front end of the plugin, hosting the buttons attached with their event listeners that communicate with content.js within the content$_s$*cripts*.

2.1 Manifest File

```
{
    "manifest_version": 2,
    "name": "CSSManager",
    "version": "1.0",
    "description": "Inspect, edit, save and load custom CSS.",
    "icons": {
      "48": "icons/icon.png"
    },

    "permissions": ["activeTab", "storage"],

    "content_scripts": [
        {
          "matches": ["<all_urls>"],
          "css": ["content_scripts/content.css"],
          "js": ["content_scripts/content.js"]
        }
    ],
    "browser_action": {
      "default_icon": "icons/icon-32.png",
      "default_title": "CSSManager",
      "default_popup": "popup/index.html"
    }
}
})
```

Listing 1: Manifest file listing general metadata and permissions of the plugin.

The most important aspects of the manifest.json file come down to the permissions, content scripts and browser action. The permissions attribute indicate what permissions are needed for the plugin to work. In the context of this app, we need access to the DOM content of the focused tab, as well as local storage to store our saved results. Secondly, we also have content scripts, where we specify the scripts that will inject code into the DOM. We must specify where we want the code to be injected, in this case we want this to work with every URL. In addition, we also specify the script that will be used along with the CSS (in this case the CSS is meant to stylize the panel only not the website).

2.2 Communication with Content Scripts

In terms of how the front end of the plug in communicates with the content script sandbox environment, it is triggered by a event listener from index.js that will communicate with the content script and send over a message that triggers certain actions. The code used here acts like a background script that communicates with the sandboxed environment.

```
document.getElementById("save").addEventListener("click", ()=>{
    browser.tabs.query({active: true, currentWindow: true}, (tabs)=>{
        //send the save command to the active tab
        browser.tabs.sendMessage(tabs[0].id, "save");
    })
})
```

Listing 2: Event Listener to trigger the save action.

Every other button action previously listed works about the same as this example. Here, we look for the "save" id associated with the plugin DOM and attach a click listener with a callback function that utilizes the WebExtensionAPI.

The WebExtensionAPI, as described by [Mozilla(2023b)], is primarily used with background scripts, and one usage is to allow a plugin to monitor for browser actions. One aspect of the browser action that can be accessed are the tabs attribute, which can help manage and observe active tabs, provided that permissions are set properly within manifest.json.

From there, we start to use the WebExtensionAPI via the use of querying the tabs and sending a message to the active tab. When we query the tabs, we are attempting to find a tab that is both active and also the one that is in focused and if we find the focused tab, we go to our callback function, with the queried tab array as a parameter. Within our callback function, we want to send the content script the command that we wish to perform and we do so via the browser.tabs.sendMessage command, which takes in the focused tab (the first element in the queried tab array) and the message (a string that represents the command).

```
browser.runtime.onMessage.addListener((request)=>{
    //toggle: toggles the empty element plane on the current webpage
    if(request === "toggle"){
        togglePanel();
    //save: save applied in-line styles
    }else if(request === "save"){
        saveStylesToLocalStorage();
    //load: load in-line styles from local storage
    }else if(request === "load"){
        loadStylesFromLocalStorage();
    //clear: clear the entry currently in localstorage
    }else if(request === "clear"){
        clearLocalStorage();
    //switch: toggle to original style on the fly
    }else if (request === "switch"){
        disableInLine();
    }
})
```

Listing 3: Content Script code that receives the message.

Within the content script, we have a listener that obtains the message from the background script and from there, we route the message to the appropriate method. It should be noted that [Mozilla(2023b)] mentions that content scripts can access some of the WebExtensionAPI, in this case the ability to receive messages from the background script via the browser.runtime name space.

3 Content Script Breakdown

Content scripts, as described in [Mozilla(2023c)], are scripts that run primarily in the context of the web page it runs on. Unlike the background script, the content scripts can read and modify dom elements with the standard DOM APIs so long as proper permissions have been added to use the script on a particular URL. However, [Mozilla(2023c)] further goes on to state that while the content script does have access to the DOM, it cannot see DOM content modified by other scripts and neither does it run in the same environment as the normal page scripts.

Since we have access to the normal APIs that allow us to manipulate DOM elements, this makes it easy to select DOM elements and apply inline styles to these elements.

```
document.addEventListener("click", (event)=>{
    //only dispay if the panel has been toggled
    const panel = document.getElementById("panel")
    if(panel.contains(event.target) || !panel){
        return;
    }

    //get the clicked element and save the applied styles
    const clickedElement = event.target;
    const cssProperties = returnCssElements(clickedElement);

    //clear panel of other css attributes
    panel.innerHTML = "";
    //div to contain the applied elements
    const elementSection = document.createElement("div");
    //display the tag name selected
    elementSection.textContent = "Clicked Tag: "+event.target.tagName;
    //append the div to the panel
    panel.appendChild(elementSection);
    //unordered list for applied styles
    const listContainer = document.createElement("ul");

    //loop through the applied css styles
    for(const [key, value] of Object.entries(cssProperties)){
        //create the list element
        let listElement = document.createElement("li");
        //make sure css property has a value attached
        if(value != ""){
            //create the checkbox and append to the unordered list
            let checkBox = createCheckbox(key, value, false, clickedElement);
            listElement.append(checkBox)
            listContainer.appendChild(listElement);
        }
    }
    ...
    panel.appendChild(listContainer);
    //prevent normal browser behavior when element is clicked
    event.preventDefault();
    event.stopPropagation();
})
```

Listing 4: Event listener that lisytens for clicks on the page and displays the styles of that element.

Focusing on the aspect that allows us to display the applied styles of a particular element, we see pretty standard use of the DOM API that is used by every website to add functionality to a page. In our case, we use the DOM manipulation to get info about the DOM, in this case the clicked element, obtain the saved element styles via the custom returnCssElements function, append these styles to the panel using a ul and then append the list to the panel.

One thing of note here is the use of preventDefault() and stopPropagation() being called on the event that was being listened on. [Mozilla(2023d)] mentions the use of preventDefault() comes down to implementing custom logic by preventing normal behavior from running. For example, if we wanted to observe the style of an a tag without having the normal behavior of going to another link, we can call upon preventDefault() to block this behavior. In addition to preventDefault(), [Mozilla(2023e)] describes stopPropagation() as a way to prevent the event handler from going further up or down to other elements.

```
function saveStylesToLocalStorage(){
    try{
        //get all elements of the page
        const elements = document.getElementsByTagName("*");
        //object to contain all elements with the style attribute
        const styles = {};

        //loop through the elements
        for(let element of elements){
            ..............
```

```
        }

        //convert styles object to JSON
        const data = JSON.stringify(styles);
        //save data to localStorage via customStyles
        localStorage.setItem("customStyles", data);
        alert("Custom styles saved to localstorage");
    }catch(error){
        console.log("Could not save element: "+error);
    }
}
```

Listing 5: Custom method saveStylesToLocalStorage() that shows off the use of localStorage method provided from the WebExtensionAPI. Abbriviated the method to show off the localStorage concept.

When the user wants to save the applied styles on the page, the code simply goes through all applied inline attributes on the DOM and saves them to a JavaScript object and then converts the object into a JSON object. From there, we can then properly use localStorage from the WebExtensionsAPI to save the JSON object with a denoted name along with the associated data.

To retrieve the data, it is as simple as:

```
const styles = JSON.parse(localStorage.getItem("customStyles"));
```

Listing 6: Line of code from loadStyleFromLocalStorage() method showing how we can retrive the saved item from localStorage with the associated name (key) and converting that JSON object back to a JavaScript object.

To delete something from localstorage, it is as simple as calling

```
localStorage.removeItem("customStyles");
```

Listing 7: Using the removeItem method from localStorage to delete the entry from localStorage by passing the key name.

4 Future Usage/Improvements

In terms of what should be added in the future, the biggest feature is to move away from the use of localstorage and move towards the building of CSS documents to give better control to the user and so they can easily access the customized style sheets they created. Another feature is to fix the styling of the CSS of the panel as the style of the panel is inherited from the current web page and is inconsistent from site to site. Lastly, the use of two way communication between the content script and the plugin script would be a feature worth implementing.

References

[Mozilla(2023a)] Mozilla. Your second extension. *MDN WebDocs*, 2023a. URL https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/Your_second_WebExtension.

[Mozilla(2023b)] Mozilla. Javascript apis. *MDN WebDocs*, 2023b. URL https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/API.

[Mozilla(2023c)] Mozilla. Content scripts. *MDN WebDocs*, 2023c. URL https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/Content_scripts.

[Mozilla(2023d)] Mozilla. Event: preventdefault() method. *MDN WebDocs*, 2023d. URL https://developer.mozilla.org/en-US/docs/Web/API/Event/preventDefault.

[Mozilla(2023e)] Mozilla. Event: stoppropagation() method. *MDN WebDocs*, 2023e. URL https://developer.mozilla.org/en-US/docs/Web/API/Event/stopPropagation.