**Chrome Scripting API**

The chrome.scripting API allows Chrome extensions to inject and execute JavaScript and CSS dynamically within web pages. Introduced in Chrome 88 (Manifest V3), this API provides more controlled and flexible script injection compared to traditional content scripts.

**Permissions and Manifest Setup**

To use the chrome.scripting API, you must declare the scripting permission in the manifest file and specify either host\_permissions for required domains or activeTab for temporary permissions.

**Example Manifest**

{

"name": "Scripting Extension",

"manifest\_version": 3,

"permissions": ["scripting", "activeTab"],

"host\_permissions": ["https://example.com/\*"]

}

**Concepts and Usage**

The chrome.scripting API enables dynamic script and CSS injection. Unlike content scripts, it allows runtime decision-making about which scripts to inject and where.

**Injection Targets**

The target parameter defines where a script or CSS should be injected. The only required field is tabId.

**Injecting in the Main Frame**

chrome.scripting.executeScript({

target: { tabId: tabId },

files: ["script.js"]

}).then(() => console.log("Script injected"));

**Injecting in All Frames**

chrome.scripting.executeScript({

target: { tabId: tabId, allFrames: true },

files: ["script.js"]

}).then(() => console.log("Injected in all frames"));

**Injecting in Specific Frames**

chrome.scripting.executeScript({

target: { tabId: tabId, frameIds: [frameId1, frameId2] },

files: ["script.js"]

}).then(() => console.log("Injected in specific frames"));

**Note:** frameIds and allFrames cannot be used together.

**Injecting Code**

**Injecting JavaScript Files**

chrome.scripting.executeScript({

target: { tabId: tabId },

files: ["script.js"]

}).then(() => console.log("Injected script file"));

**Injecting Inline Functions**

You can directly inject a function instead of a file.

function changeBackgroundColor() {

document.body.style.backgroundColor = "yellow";

}

chrome.scripting.executeScript({

target: { tabId: tabId },

func: changeBackgroundColor

}).then(() => console.log("Injected function"));

**Passing Arguments to Functions**

function changeBackgroundColor(color) {

document.body.style.backgroundColor = color;

}

chrome.scripting.executeScript({

target: { tabId: tabId },

func: changeBackgroundColor,

args: ["blue"]

}).then(() => console.log("Injected function with argument"));

**Injecting CSS**

**Injecting CSS Files**

chrome.scripting.insertCSS({

target: { tabId: tabId },

files: ["styles.css"]

}).then(() => console.log("Injected CSS file"));

**Injecting CSS as a String**

chrome.scripting.insertCSS({

target: { tabId: tabId },

css: "body { background-color: red; }"

}).then(() => console.log("Injected CSS string"));

**Handling Results**

When executing scripts, the results of the execution are returned as an array, containing results from each frame in the order of execution.

function getTitle() {

return document.title;

}

chrome.scripting.executeScript({

target: { tabId: tabId, allFrames: true },

func: getTitle

}).then(injectionResults => {

for (const { frameId, result } of injectionResults) {

console.log(`Frame ${frameId} result:`, result);

}

});

**Note:** scripting.insertCSS() does not return any results.

**Handling Promises**

If the injected function returns a Promise, Chrome waits for it to resolve.

async function fetchTitle() {

const response = await fetch("https://example.com");

const text = await response.text();

return text.substring(0, 50); // Return first 50 characters

}

chrome.scripting.executeScript({

target: { tabId: tabId },

func: fetchTitle

}).then(results => {

console.log("Fetched Title: ", results[0].result);

});

**Unregister Dynamic Content Scripts**

To unregister all dynamic content scripts:

async function unregisterAllScripts() {

const scripts = await chrome.scripting.getRegisteredContentScripts();

const scriptIds = scripts.map(script => script.id);

return chrome.scripting.unregisterContentScripts(scriptIds);

}

**ExecutionWorld Enum**

**ISOLATED**

Runs in an isolated environment unique to the extension.

**Example:** If an extension injects a script in ISOLATED mode, it does not have direct access to the page's JavaScript variables.

chrome.scripting.executeScript({

target: { tabId: tabId },

func: () => {

window.pageVariable = "Injected Value";

},

world: "ISOLATED"

});

**Result:** window.pageVariable will not affect the actual webpage.

**MAIN**

Runs in the main DOM context of the page.

**Example:**

chrome.scripting.executeScript({

target: { tabId: tabId },

func: () => {

window.pageVariable = "Injected Value";

},

world: "MAIN"

});

**Result:** window.pageVariable is accessible by the webpage.

**InjectionTarget Properties**

* tabId (number, required): Tab ID where the script should be injected.
* allFrames (boolean, optional): Injects into all frames if true.
* frameIds (number[], optional): Specific frames to inject into.
* documentIds (string[], optional): Specific documents to inject into (Chrome 106+).

**ScriptInjection Properties**

* files (string[], optional): Path to JS or CSS files.
* func (function, optional): Function to execute.
* args (any[], optional): Arguments for the function.
* world (ExecutionWorld, optional): Execution environment.

**Conclusion**

The chrome.scripting API provides a robust way to inject scripts and styles dynamically into web pages, offering flexibility over traditional content scripts. By leveraging this API, developers can modify and interact with web pages in real-time, ensuring a powerful extension experience while maintaining security and efficiency.