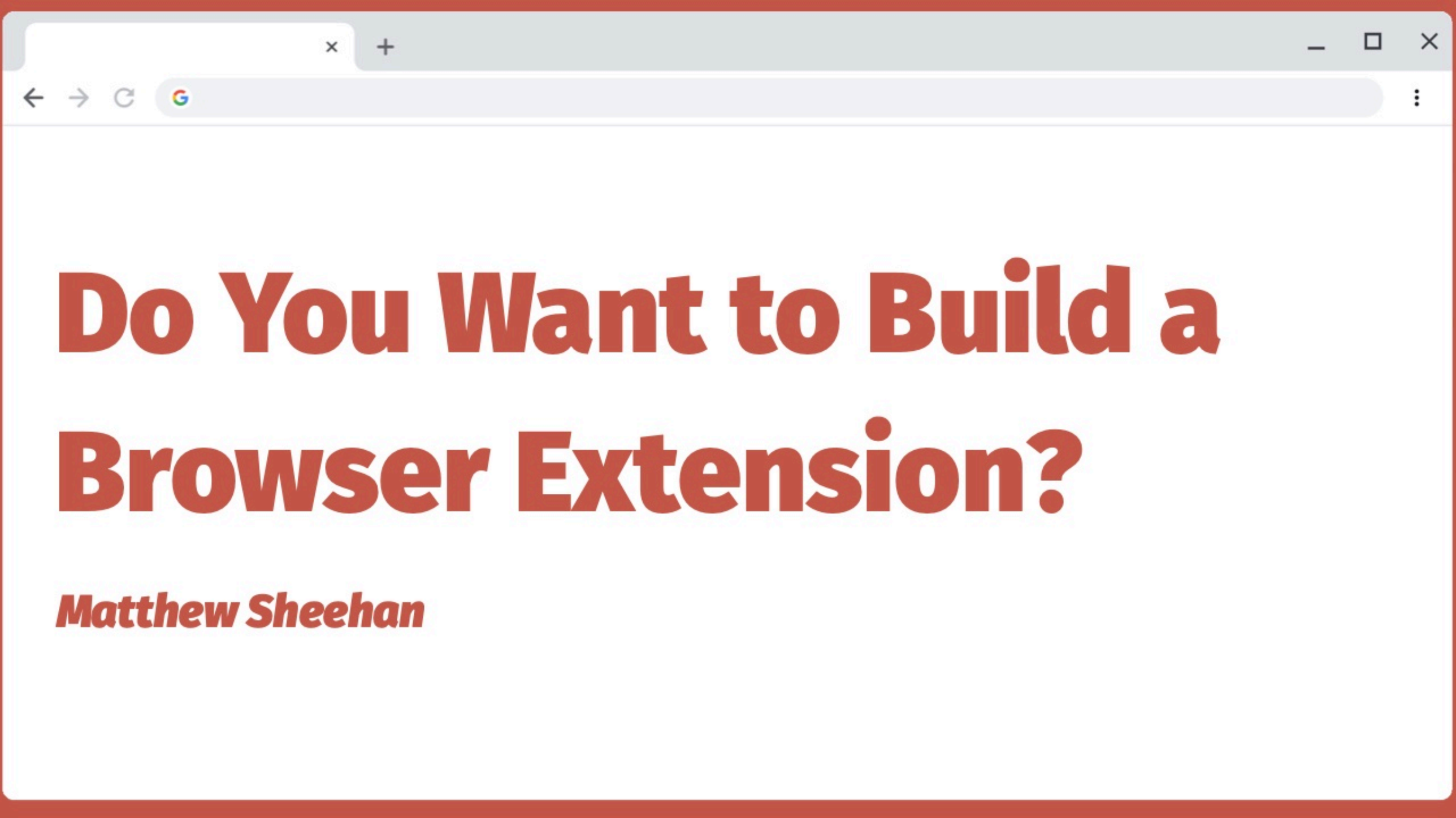

Do You Want to Build a Browser Extension?
Matthew Sheehan

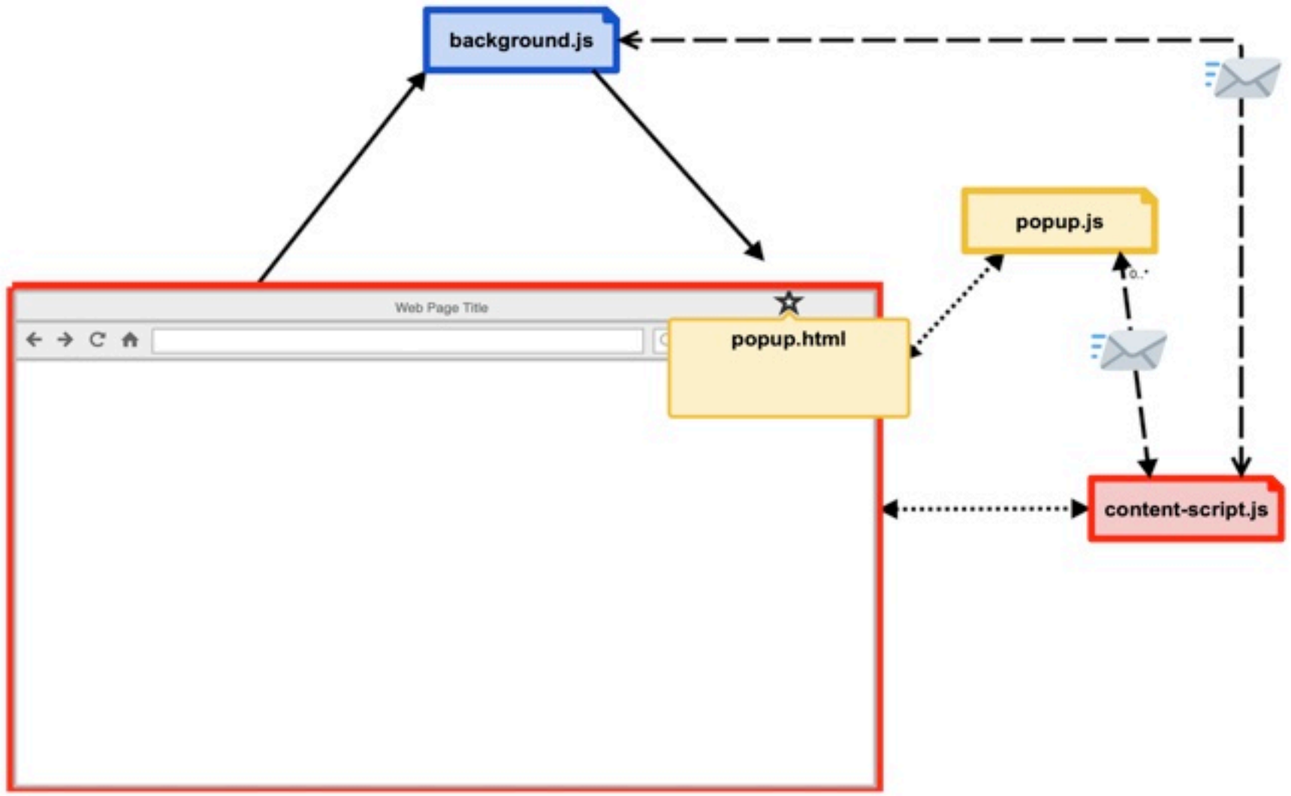
Getting Started

- → Focus on Chromium-based Browsers
  - → Google Chrome, Microsoft Edge
  - → Firefox compatibility: `https://www.extensiontest.com`
- → Focus on Manifest V3 WebExtension API
  - → Service Workers and Promises

Anatomy of a Browser Extension

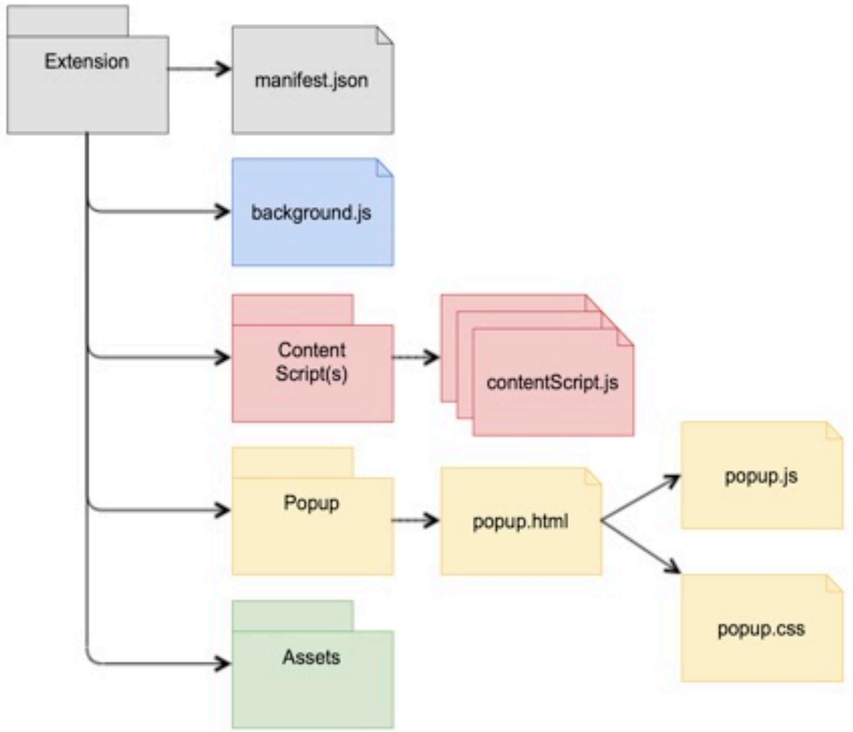
- → Manifest File
- → Service Worker (background page)
- → Content Script
- → Toolbar Icon
- → Popup UI Elements
- → Options Page

Architecture



Identifying the Components

```
{
    "manifest_version": 3,
    "background": {
        "service_worker": "background.js"
    },
    "content_scripts": [
        {
            "matches": ["https://*/*"],
            "js": ["contentScript.js"]
        }
    ],
    "action": {
        "default_popup": "popup.html"
        ...
    },
    "web_accessible_resources": [
        {
            "resources": ["assets/*"],
            "matches": ["https://*/*"]
        }
    ],
    ...
}
```



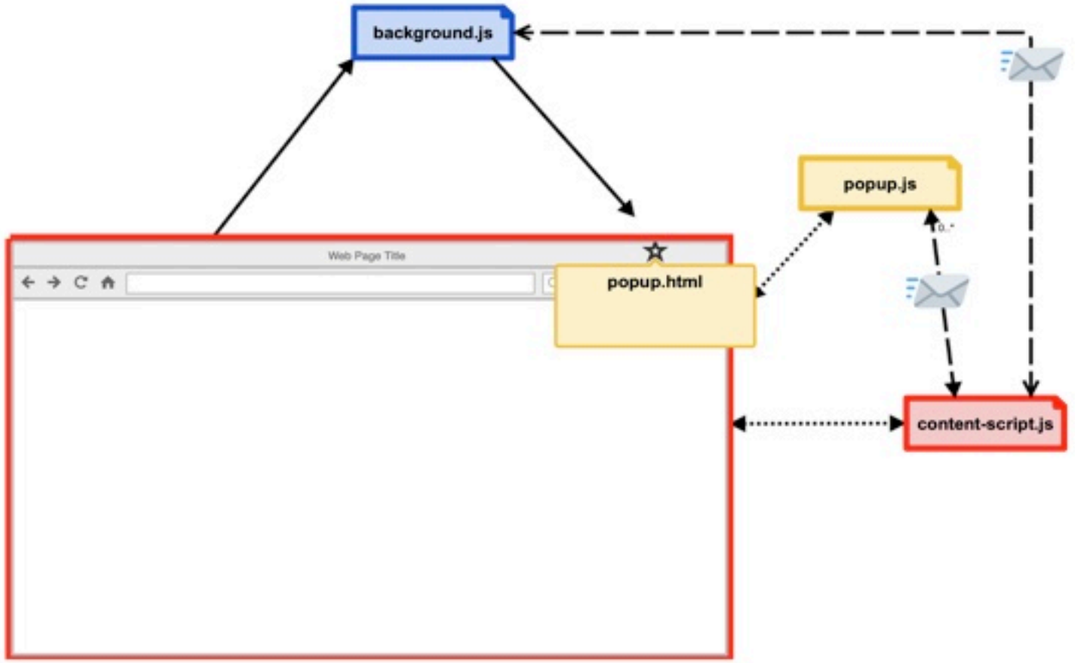
**Condensed Manifest*

Example 1 - Sideloading

- Chrome Developer Mode & Sideloading
- Starter Extension Template
  - `https://github.com/MattSheehanDev/chrome-manifest-v3-starter-template`

Service Workers

- Service workers are more specialized web workers
  - Communicate through messages
  - Events and event handlers
- Don't have access to anything from the global `Window` interface
  - DOM
  - localStorage
  - XMLHttpRequest (`fetch()` Web API available)



Message passing 📨

→ One-time messages

```
// contentScript.js, send message to background service worker
chrome.runtime.sendMessage({message: "hello"}, function(response) {
  console.log(response.message);
});


// background.js, send message to content script
chrome.tabs.sendMessage(tabId, {mesage: "hello"}, function(response) {
  console.log(response.message);
});


// contentScript.js or background.js listener
chrome.runtime.onMessage.addListener((request, sender, sendResponse) => {
  sendResponse({message: "goodbye"});
});
```

Message passing 📨

→ Long-lived connections

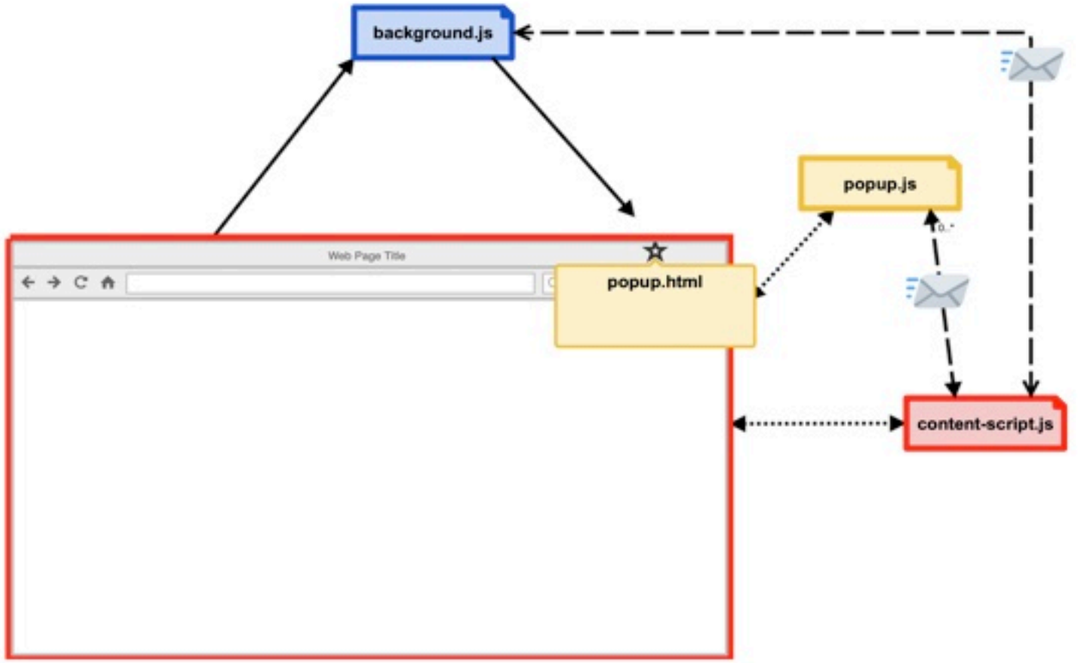
```
// contentScript.js, connect to background service worker
var port = chrome.runtime.connect({name: "greet"});
port.postMessage({message: "hello"});
port.onMessage.addListener((msg) => { });

// background.js, connect to content script
var port = chrome.tabs.connect(tabId, {name: "greet"});
port.postMessage({message: "hello"});
port.onMessage.addListener((msg) => { });

// contentScript.js or background.js add listener
chrome.runtime.onConnect.addListener((port) => {
  port.onMessage.addListener((msg) => {
    port.postMessage({message: "goodbye"});
  });
});
```

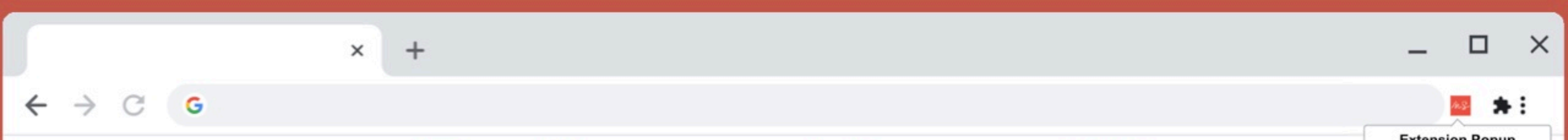
Example 2 - Debugging the pieces

- → Background Script Errors
- → Inspecting the Background Script
- → Content Script Logs
- → Inspecting the Popup UI



APIs and Permissions

- → Chrome exposes a number of browser API's through the `chrome.*` namespace (`browser.*`)
- → Some of these API's require requesting extra permission
  - → Minimize requested permissions
  - → Optional features with permissions can registered as `optional_permissions`
  - → Users trust extensions with less permission warnings



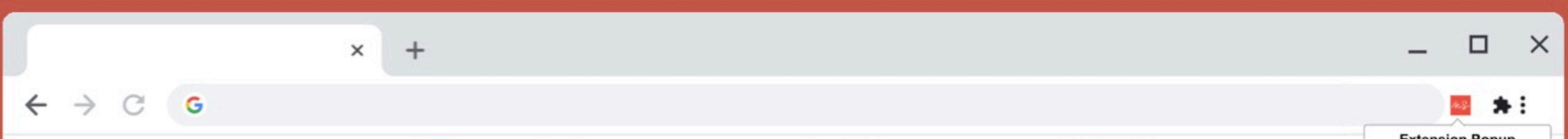
Toolbar and Popup UI

API	Permission
chrome.actions	none

```
{
  "name": "Chrome Extension Action Manifest",
  "action": {
    "default_icon": {
      "16": "images/icon16.png",
      "24": "images/icon24.png",
      "32": "images/icon32.png"
    },
    "default_title": "tooltip text",
    "default_popup": "popup.html"
  }
  ...
}
```

→ `chrome.action.setIcon(...);`

→ `chrome.action.setPopup(...);`



Toolbar and Popup UI

API
chrome.declarativeContent

Permission
"declarativeContent"

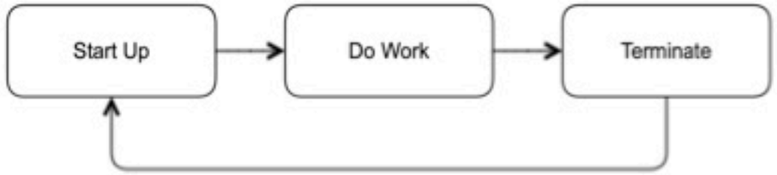
→ Enables or disables the extensions action depending on the properties of a web page.

```
let rule = {
  conditions: [
    new chrome.declarativeContent.PageStateMatcher({
      css: ["input[type='password']"]
    })
  ],
  actions: [ new chrome.declarativeContent.ShowAction() ]
};

chrome.declarativeContent.onPageChanged.addRules([rule]);
```

Service Workers

- Service Workers (background script) are non-persistent



  - **DO** Register event listeners at startup
  - **Do not** store state in global variables
  - **Do not** use `setTimeout` or `setInterval`

Persisting State

API	Permission
chrome.storage	"storage"

- → Similar API to localStorage but more robust
- → Can also sync settings to a user's chrome profile

```
chrome.storage.local.set({key: value}, () => {
  console.log(`${key} is set to ${value}`);
});

chrome.storage.local.get(['key'], (result) => {
  console.log(`Value of key is set to ${result.key}`);
});
```

Example 3 - Full Extension

→ Extension to track Hacker News comments

Security considerations

- Trustworthiness: Content script < Background page
- Content-Security-Policy and Sandboxes
  - `chrome-extension://`
  - Sandboxed extensions are served from a unique origin. Access denied because of same-origin policy.

```
{
  ...
  "content_security_policy": {
    "sandbox": "sandbox allow-scripts; script-src 'self' 'unsafe-inline' 'unsafe-eval'; child-src 'self';"
  },
  "sandbox": {
    "pages": [
      "sandboxedExtensionPage.html"
    ]
  },
  ...
}
```

Publishing your Extension

- → Create a Chrome Web Store Developer Account
  - → https://chrome.google.com/webstore/devconsole/register
- → Zip and upload your extension
- → Submit your extension
  - → Justify any permissions your extension uses
  - → Loading and executing remote code, as of Manifest V3, is not accepted