Contents

Getting Started with Mozilla Webextensions

During 2015 Firefox team has made an exciting very big announcement to Firefox add-ons, they have brought a new extension API's called WebExtension [1]. This update has lot of exciting announcements for developers.

1. Review process will become faster
2. Development is easier, and the Addons will be compatible with Chrome and Opera.

You can find the whole information regarding this update in Add-ons Blog [2]

The next step is to download the Firefox Browser and install them. You can either go for developer Edition [3] or Nightly [4]

So as of now Firefox Will allow only the add-ons which are signed when you start the browser. So for development purpose we have to change the settings.

1. Open the browser
2. in the URL bar type about:config

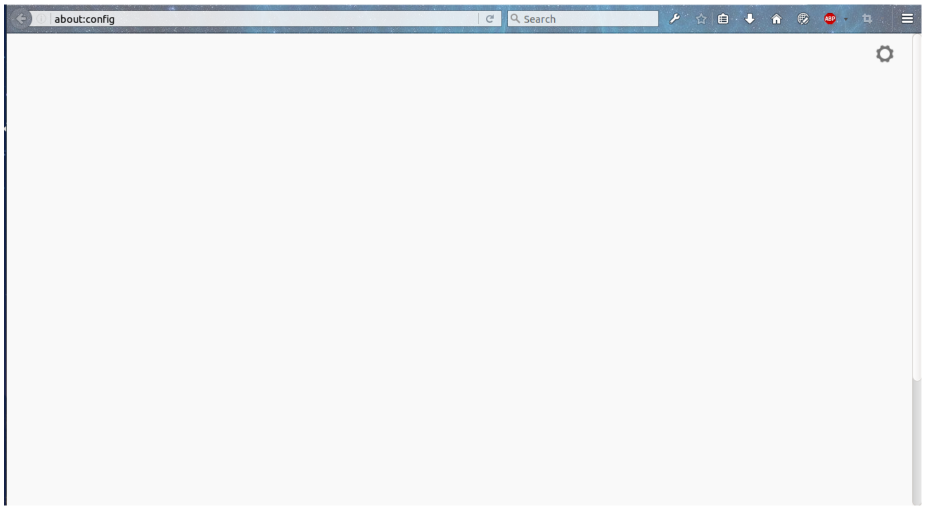
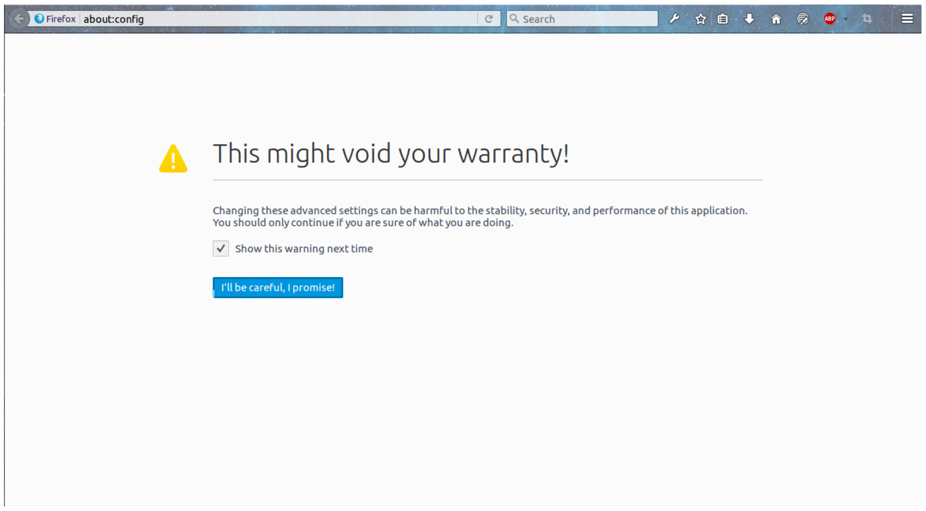


Figure 1: config in URL bar

3. In the search bar of that page type xpinstall.signatures.required , you will see the image as like below. By default it will be true
4. You can double click on it or else right click and select toggle. So it will be set to false

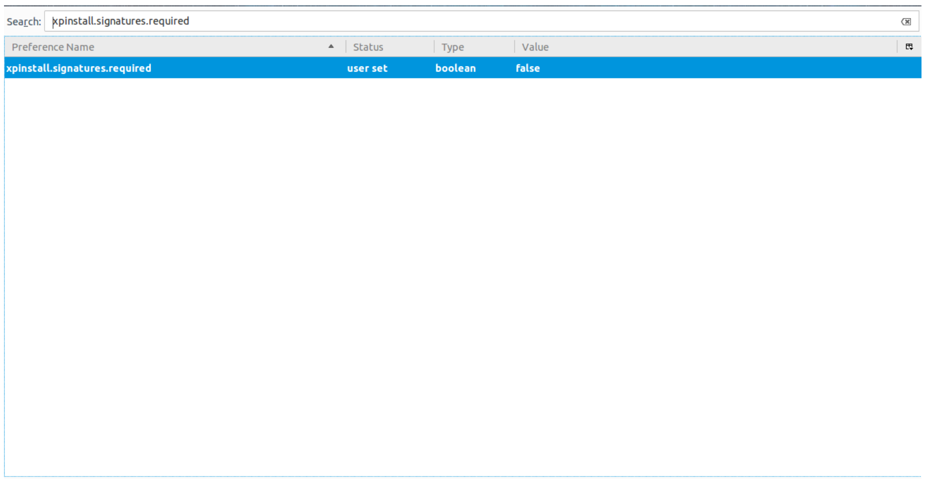


This might void your warranty!

Changing these advanced settings can be harmful to the stability, security, and performance of this application. You should only continue if you are sure of what you are doing.

✓ Show this warning next time

Figure 2: Warning



Search: xpinstall.signatures.required

Preference Name	Status	Type	Value
xpinstall.signatures.required	user set	boolean	false

Figure 3: Signature

Why Web Extensions

One of the Main reason, to get started with Web Extensions is easier to develop. Previously I have tried developing some Chrome Extensions during my free time, so since the same thing is used here, it will be very easier for me to get started and develop well. So I write once for Firefox Addons and it will support multiple platforms.

Very excited for this new journey of learning.

References

- [1] https://wiki.mozilla.org/WebExtensions
- [2] https://blog.mozilla.org/addons/2015/08/21/the-future-of-developing-firefox-add-ons/
- [3] https://www.mozilla.org/en-US/firefox/developer/
- [4] https://nightly.mozilla.org/

Running WebExtension in Firefox

In this section we will be exploring more how to run an Firefox WebExtensions.

I will be showing this demo with the help of the small extension I have written before, feel free to download it from github [1], you can clone whole repo [2]. I personally use web-ext [3] tool for packing my extension, but for basic learning it is not mandatory.

Step by Step procedure for running WebExtension

Step 1: Go to the directory which has contents of Extension

In my case it will look like the below. The content showed below is the contents of Extension, which the link is shared above.

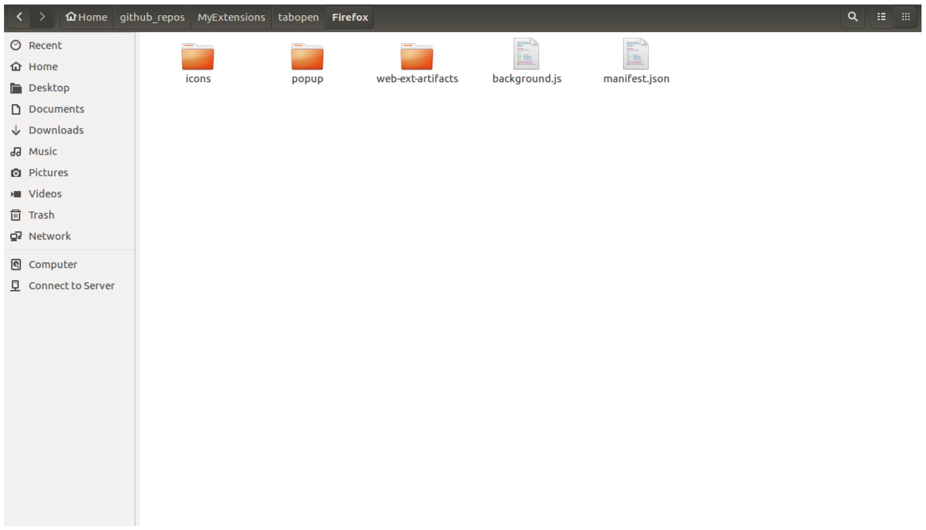


Figure 4: Directory Image

Step 2: We have to compress all the contents into Zip

- Select all the folders and files
- Make it to a zip, anyname.zip

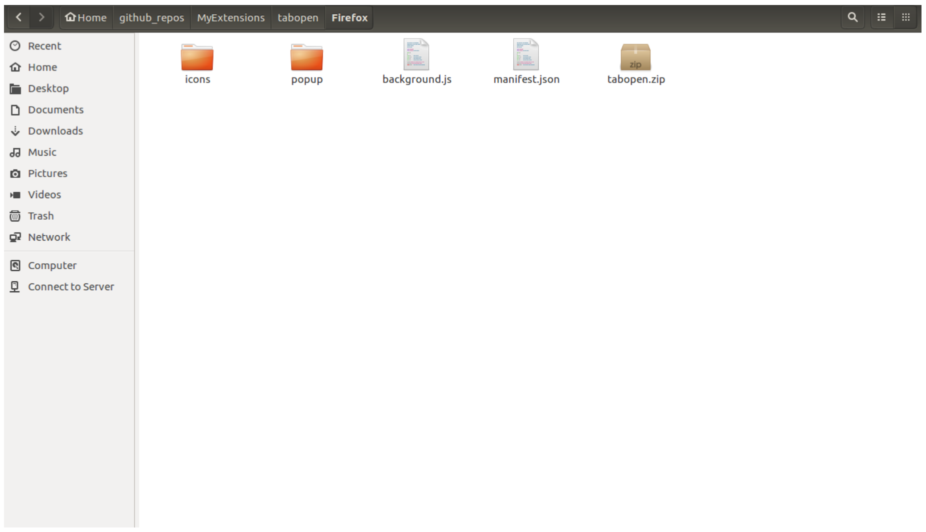


Figure 5: Directory Image

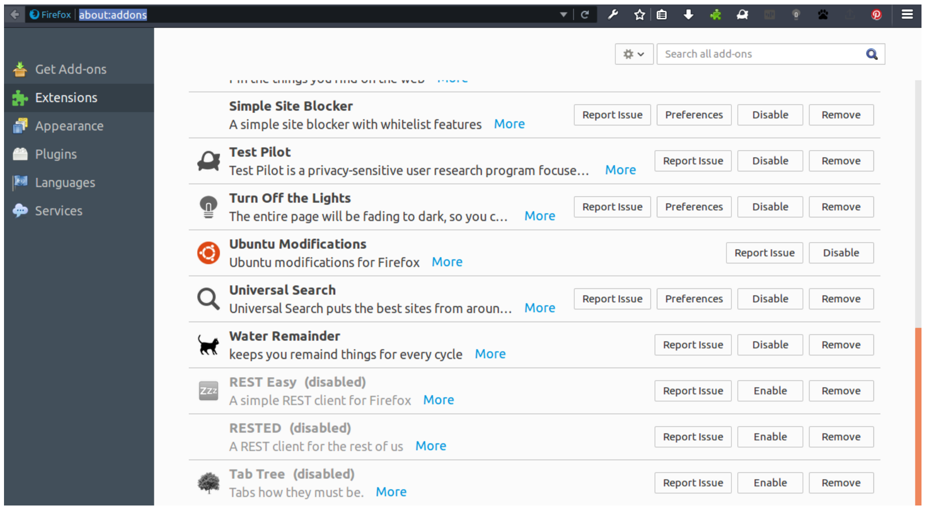

Firefox
about:addons
Search all add-ons
Get Add-ons
Extensions
Appearance
Plugins
Languages
Services
Simple Site Blocker
A simple site blocker with whitelist features More
Report Issue Preferences Disable Remove
Test Pilot
Test Pilot is a privacy-sensitive user research program focuse… More
Report Issue Disable Remove
Turn Off the Lights
The entire page will be fading to dark, so you c… More
Report Issue Preferences Disable Remove
Ubuntu Modifications
Ubuntu modifications for Firefox More
Report Issue Disable
Universal Search
Universal Search puts the best sites from aroun… More
Report Issue Preferences Disable Remove
Water Remainder
keeps you remaind things for every cycle More
Report Issue Disable Remove
REST Easy (disabled)
A simple REST client for Firefox More
Report Issue Enable Remove
RESTED (disabled)
A REST client for the rest of us More
Report Issue Enable Remove
Tab Tree (disabled)
Tabs how they must be. More
Report Issue Enable Remove

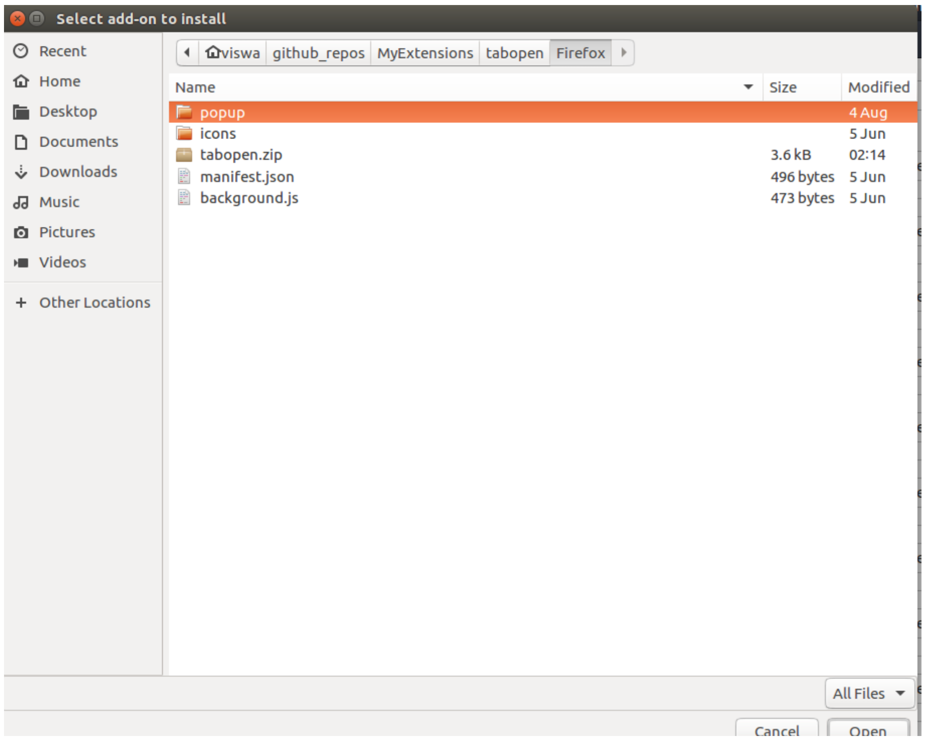
Figure 6: Directory Image

Step 3: Open about:addons or Ctrl+Shift+A

Step 4: Install Addons or Debug Addons

Install Addons

- Click on Tools for all add-ons (the icon - which is like Setting icon)
- Select Install Add-on From File ..
- Make sure you have set the file type to all files
- Choose the Zip which you created in step 2



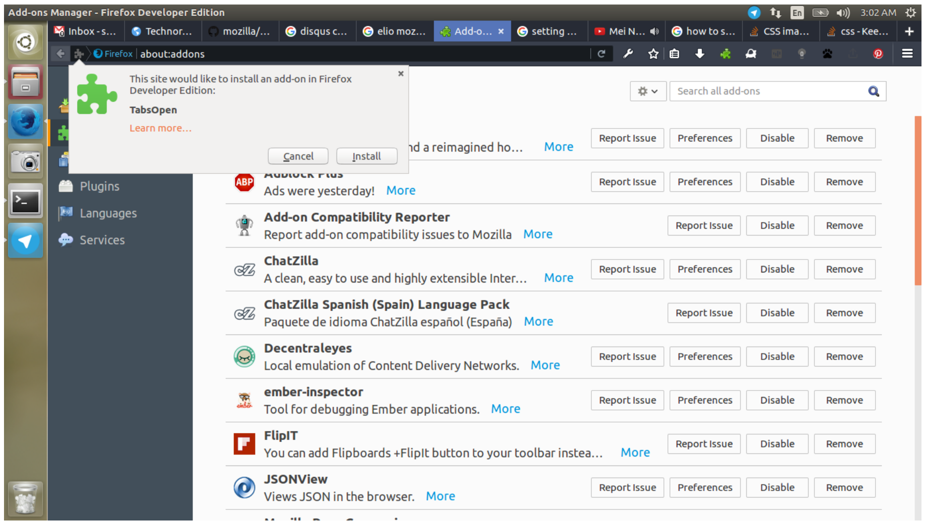
Name	Size	Modified
popup		4 Aug
icons		5 Jun
tabopen.zip	3.6 kB	02:14
manifest.json	496 bytes	5 Jun
background.js	473 bytes	5 Jun

Figure 7: Choose the Zip file

- You will get a pop-up notification after selecting at left croner
- In that pop-up click Install button
- Now your Extension is installed.

Debug Addons

During testing it is enough to go with Debug Add-ons option. Only drawback will be, if we close the Firefox then the Add-on which we installed during our development phase will be uninstalled. Follow steps below.


Add-ons Manager - Firefox Developer Edition

about:addons

This site would like to install an add-on in Firefox Developer Edition:

TabsOpen

Learn more…

Cancel Install

...nd a reimagined ho… More

Adblock Plus
Ads were yesterday! More

Add-on Compatibility Reporter
Report add-on compatibility issues to Mozilla More

ChatZilla
A clean, easy to use and highly extensible Inter… More

ChatZilla Spanish (Spain) Language Pack
Paquete de idioma ChatZilla español (España) More

Decentraleyes
Local emulation of Content Delivery Networks. More

ember-inspector
Tool for debugging Ember applications. More

FlipIT
You can add Flipboards +FlipIt button to your toolbar instea… More

JSONView
Views JSON in the browser. More

Figure 8: Choose the Zip file

- Click on Tools for all add-ons (the icon - which is like Setting icon)
- Debug Add-on.
- Goto the folder where you source code is present.
- Select the manifest.json file, so the Add-on is installed temporarily.

Step 5: Run the Extension

For most of Extensions you will get the icon in the tool bar itself. So for this a page like icon will be coming at toolbar, if every code is run properly. Click on that icon.

So on Selecting any one of the three websites, it will open a new tab and open the site. As of now it is in static mode.

In future, based on user usage we can show top 3 websites there.

Follow for amazing journey of Building WebExtensions.

References

- [1] https://github.com/iamVP7/MyExtensions/tree/master/tabopen/Firefox
- [2] https://github.com/iamVP7/MyExtensions/
- [3] https://github.com/mozilla/web-ext

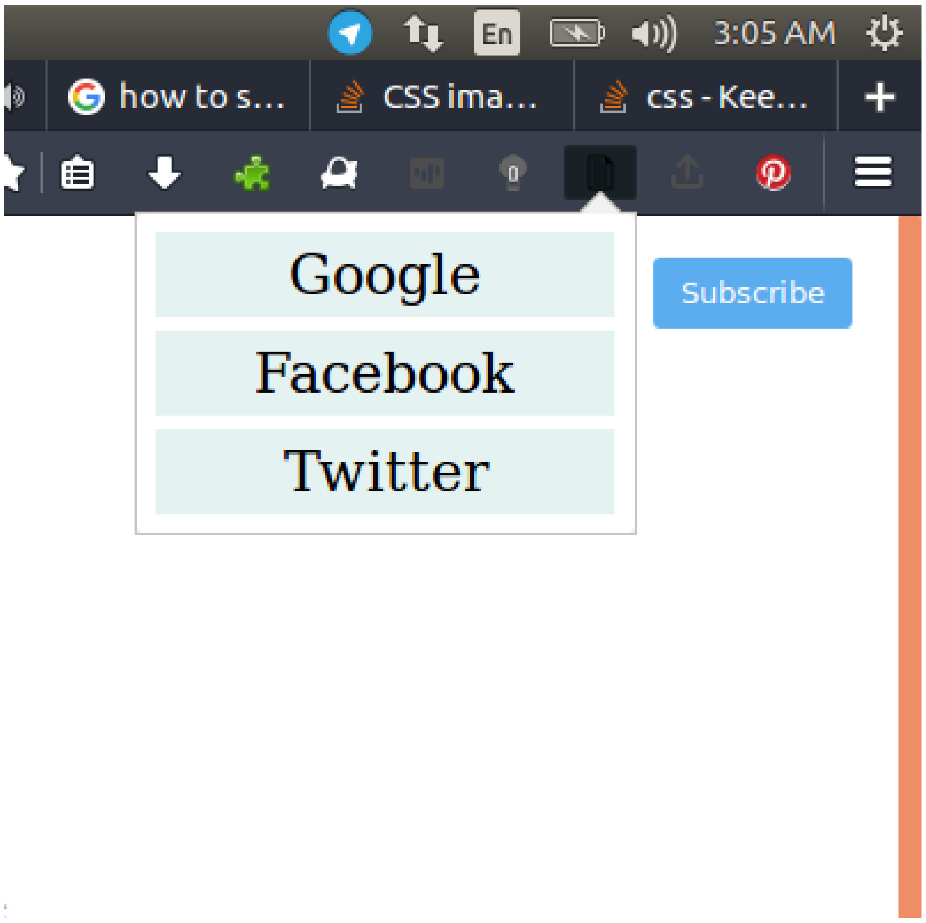


Figure 9: Choose the Zip file

Experiment 1: Building our Motivational Tab

We got to know what is meant by WebExtension and how to install WebExtension from the source code available in our system. Those leasons are very important to us, as we move forward with our development, we will be always using them.

Our Approach is learn the basic code, try them out with little modification and run them in our system, try out the execrise and share about this learning in social medium of community benefit.

Problem statement and Solution

Problem Statement

We have to open different web pages each and every day. Sometime we will be bored to do them. At that time, we may feel to get a cup of coffee. For sure its not possible to count the number of copies we had. Having too much of coffee or other caffeine products is injurious to health.

Solution

So the very first solution is to make sure we are not bored, and we are motivated when we open a new tab. We can be motivated by seeing strong quotes by our favorite personality or a our family pictures whenever we open a new tab.

What is relationship with WebExtension???

Whenever you are opening a new tab, it means we are calling some internal API to open the tab. Each and every action we do is an API, some of them are exposed via WebExtension API. So by default if you check in your browser you will understand you have options to change what should be shown when a new tab is opened. A screenshot from Firefox is shown below, check under the heading *New Windows and Tabs*

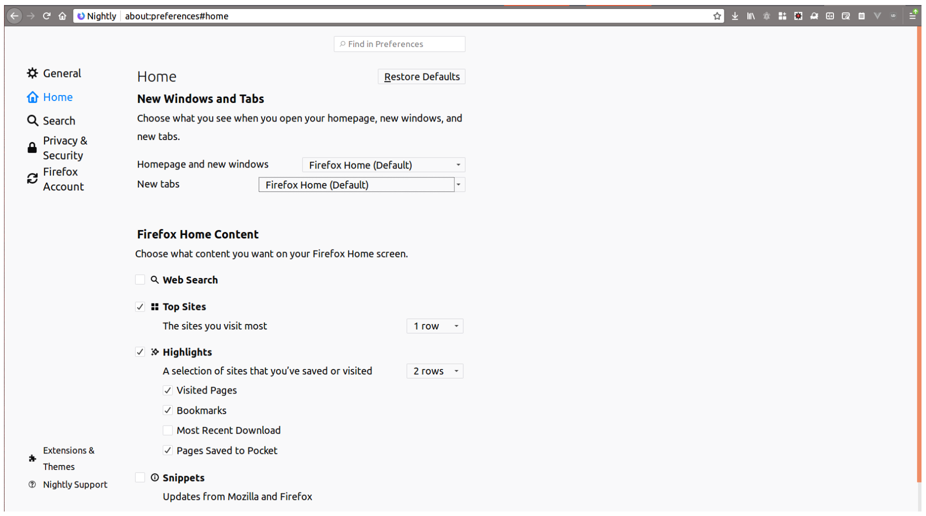
How we can capture this

We need not worry much. It is easy to capture when a new tab is opened. And also it is easy to open our preferred page whenever new tab or window is opened. In this section we will be seeing how to open our own page with some random pre defined quotes using *chrome_url_overrides* WebExtension API.

Building Blocks of Our Motivation Tab WebExtension

Writing manifest.json

For every program we will start by writing the manifest.json, as it will be giving us the clear view what we are going to do and what are all the components like permission, background script, resources and so on, we are going to have.


Nightly about:preferences#home

Find in Preferences

General
Home
Search
Privacy & Security
Firefox Account
Extensions & Themes
Nightly Support

Home
Restore Defaults

New Windows and Tabs

Choose what you see when you open your homepage, new windows, and new tabs.

Homepage and new windows: Firefox Home (Default)

New tabs: Firefox Home (Default)

Firefox Home Content

Choose what content you want on your Firefox Home screen.

☐ Web Search

✓ Top Sites

The sites you visit most: 1 row

✓ Highlights

A selection of sites that you've saved or visited: 2 rows

✓ Visited Pages

✓ Bookmarks

☐ Most Recent Download

✓ Pages Saved to Pocket

☐ Snippets

Updates from Mozilla and Firefox

Figure 10: New tab Configuration

```
{
"manifest_version": 2,
"name" : "Motivation Tab",
"description" : "Shows Random Quotes ",
  "version": "1.0",
  "chrome_url_overrides" : {
    "newtab": "my-new-tab.html"
  }
}
```

The above is our manifest.json required for this program. The very important keys which we have to define for the manifest json are as follows

- name
- description
- manifest_version is always 2 (till Mozilla changes)
- version is the version of this addon, and it should be incremental.

Additionally for this particular Add-on (WebExtension) we are going to use *chrome_url_overrides*

What is chrome_url_overrides ??

We can remember it as URL Overrider, means it will be over riding the default with the value we give in the extension. In our example we are saying to over ride the **newtab** , so whenever there is a new tab created if our extension is installed then, our extension will be over riding the other pages and will show the page which we have defined.

So our definition will be like below,

```
"chrome_url_overrides" : {
    "newtab": "my-new-tab.html"
  }
```

In the chrome_url_overrides defintion, the value of *newtab* is “my-new-tab.html”, which is defined by us. We can define what pages we want.

Our Basic HTML Page

Our next step is to write the HTML page.

```
<!DOCTYPE html>
  <html>
    <head>
      <meta charset="utf-8"/>
      <title>The best Motivation Tab</title>
    </head>
    <body>
      <h1 id="quote" style="text-align: center"></h1>
      <script type="text/javascript" src="script.js"></script>
    </body>
  </html>
```

Our HTML page is very minimal, we will be having only **one heading element** with unique id for that, and we will be including the javascript file.

Our Javascript file,

Our JS file will be having random quotes and the colors for the background.

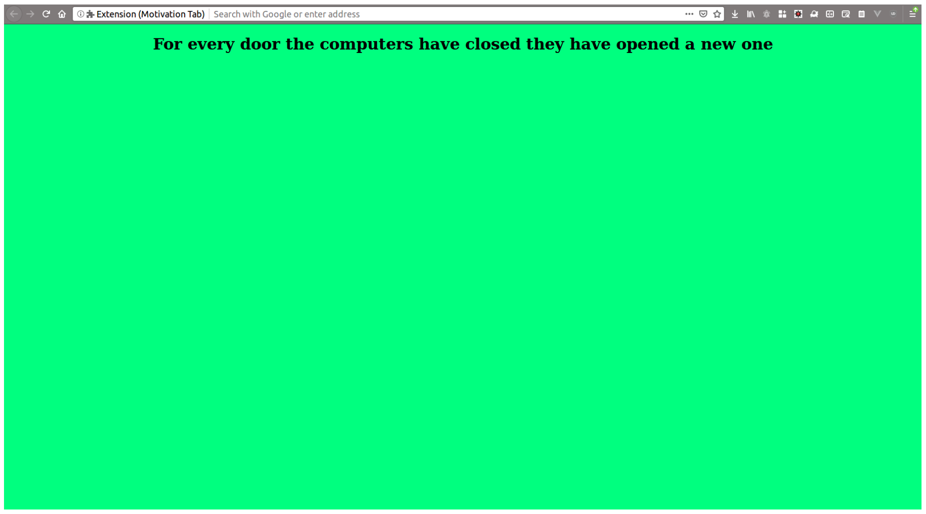
```
let random_quote = [
 "quote1", "quote2", "quote3"
];

let color_hex = ['#778899','#B0C4DE','#E6E6FA','#90EE90','#00FF7F'];
function getRandomQuote() {
  // Generate Random Number for Quote
  var quote_random_num = Math.floor((Math.random() * random_quote.length));
  // Set the content in the div
  document.getElementById("quote").innerText  = random_quote[quote_random_num];
  // Get Random Number for color
  var random_col_num = Math.floor((Math.random() * color_hex.length));
  // Update the background with color
  document.body.style.backgroundColor = color_hex[random_col_num];

}

// Call the method getRandomQuote when this js loads
getRandomQuote();
```

Thats it, our extension (WebExtension or Add-on) is ready, once we do temporary install we can experience it. When we open a new tab our page by default will look like below.


Extension (Motivation Tab)
Search with Google or enter address

For every door the computers have closed they have opened a new one

Figure 11: New tab Addon

Exercise

Make sure to visit Mozilla Developer Network (MDN) [1] to learn more about the above API.

So try out this API in your get started program. Your task is simple you have to load beautiful images whenever a user opens a new tab. We have already done opening Beautiful Quotes, instead of words you make it as image. Remembers visuals have more power compared to basic text words.

Optional: It will be great if you can share your code or blog about this learning in twitter. Our Mozilla Webextension twitter handle is (@MozWebExt) and make sure you use hashtag #WebExtLearn when you are tweeting about this learning.

References

- [1] https://mzl.la/2PBUToi
- [2] https://twitter.com/MozWebExt

Experiment 2: Building our Power Search Add-on

We got to many browser actions can be used by developers via WebExtension API(Application Program Interface), we also built our first WebExtension with which we can get motivational images whenever we open a new tab.

Problem statement and Solution

Problem Statement

When we are surfing a lot there are lot of cases where we need to search on different search engines. We always don't need to use Google or Bing alone. Sometimes we will directly search in Wikipedia or want to do video search in Youtube. We are not always active to make everything using Keyboard(KB) because keyboard follow is something like

- select the word using mouse, right click and copy it or Ctrl+C
- goto the website (if we know the address correctly we will go directly or type in google and go there)
- paste the word which we copied and hit enter.

I understand you might have sipped your coffee more than twice here.

Solution

So our good solution will be to use the mouse. The follow with mouse as follows;

- select the word.
- right click.
- search in our preferred search engine by selecting from list.

When you are selecting a word and doing the right click, you will be poped up with list of options. This small window which is coming near your mouse cursor is called as Context Menu.

How we can capture this

Our proposed solution has totally 3 steps. First is selecting the word, once right click is done we have to populate our options (list of search engines available in browser), then once the preferred search engine is selected then we have to create the new tab. As we have 3 steps here we are going to use 3 simple WebExtension API for this.

- Context Menu API
- Search API
- Tabs.create API (Tabs API has so many sub-api inside them, we will go through them in future)

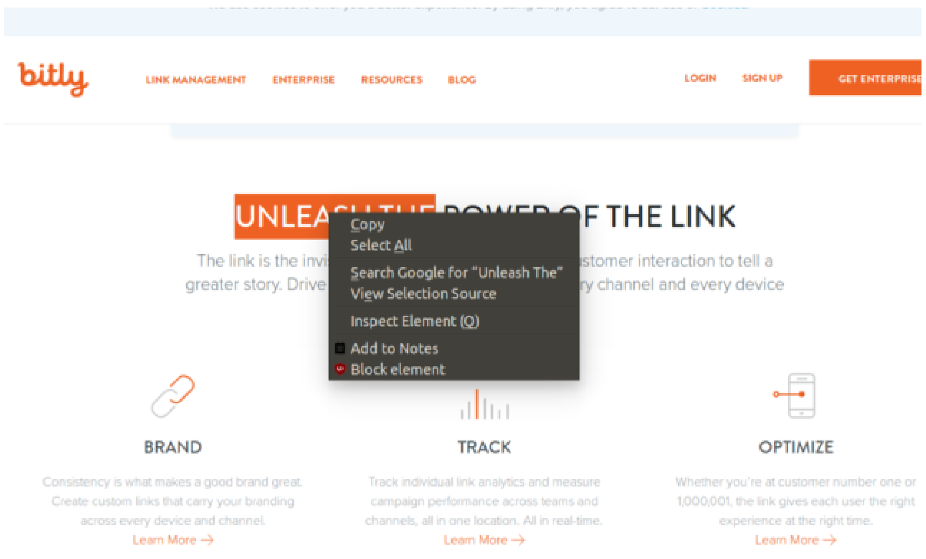


Figure 12: Context Menu

Since our problem statement is clearly defined and we are going to use this three API only, our WebExtension development will be easier and can be finished quickly.

Building Blocks of Our Power Search Add-on

For this extension we will learn the API step by step. So at the end joining them will be very easier.

Create the New Tab

To create the tab, we just need the fully qualified URL which we need to open. We need to give url like “https://www.google.com” instead of “www.google.com”

A sample script is below.

```
browser.tabs.create({
    url:"https://twitter.com/"
  });
```

In the above example, the value of **url** is “https://twitter.com/”, we have to change this if we need to open the different tab.

Context Menu API

Context Menu is nothing but when we click the right click a new pop-up comes near our mouse pointer. We can add our own options in the context menu using Browser Action API. It comes under privileged API, so we need to get **"contextMenus"** permission for using context menu. In our program we will be adding the list of search engines in the context.

To add Options in Context Menu

It is very simple to add options in the Context menu.

```
browser.contextMenus.create({
  id: "id1",
  title: "Display Word",
  contexts: ["selection"]
});
```

For each and every context menu item we need to give unique **id** so based on it we can do different actions. In this the value of **title** is what general users see. In Previous image **Add to Notes** is title. Then we need to mention, when we need to show the particular contextMenu item, whether we need to show always or when we select a word (selection), when we select image alone; this is defined in **contexts**

Note: If we have only one contextMenu for our WebExtension then it will be shown directly, if we have more than 1 then under our Extension Name the list will be available.

Getting the selected word

Whenever we click on the contextMenu item we will be getting two things to the listener method we have defined. First one is **info Object** which will have variety of options based ont he context and Second one is **tab Object** which will have information like tabid, url and so on. When we want to start the search in our desired website, we need to have the word which is selected, so from contextMenu item we will be getting the selected word.

The word which we have click will be available in **info Object** with the key **selectionText**

Search API

Search API will be providing us the details like what are all the search engines installed in the browser, which one is default, what is their fav icon. Search API is also privileged one, we need to get **"search" permission** from user.

Get list of Search Engine

Search API will be giving the list of search engine installed in our browser. As discussed before we may be having more than 1 search engine in our browser. If its not available we can install from addons.mozilla.org it will be easier and helpful.

```
var list_of_searchengine = browser.search.get()
```

The above simple snippet will be fetching and giving us the list of search engine available in our machine. Each search engine will be having the values like name i.e., search engine name , may have favIconUrl i.e., search engine fav icon URL. We may be using this two things.

Searching in Preferred Engine

Once we got the selectedText from contextMenu our next step would be to search in the preferred search engine. To search in our required engine below code snippet will do.

```
browser.search.search({
    query: "word to search",
    engine: "Wikipedia (en)"
  });
```

In this case **query** which will have the word to be searched in mandatory, **engine** is not mandatory, if we are not mentioning it, then default one will be taken. In case we mentioned it wrongly exception is thrown.

Assembling our Power Search Extension Parts

Previously we learned about different parts for building our Power Search Extension. We have to create context menu using the search engines available, get selectionText value and search it in the search engine we have.

Writing manifest.json

As usual lets start with writing our manifest file. Here in this time we are additionally introducing three new keys for manifest.json

```
{
"manifest_version": 2,
"name" : "Power Search Engine",
"description" : "Search the selected text in other search engines in your machine",
"version": "1.0",
"icons": {
    "96": "icons/icon-96.png"
```

```
  },
  "permissions" : [
      "search","contextMenus"
    ],
"background": {
    "scripts": ["background.js"]
  }
}
```

icons

icons is the JSONObject, which can be used to show the icon for particular extension. It is good practice to have beautiful icons for our extension. The normal size we should include is 96px X 96px, 60px X 60px.

permissions

Permission are mandatory if we are going to use API with specific data. In this experiment we are going to use search API and contextMenu API. We have to request permission from user. Only if user grants permission we can run this extension.

background

As the name suggest we will have something running in the backgroud. Here we can define array of scripts so they will be useful and will be listening to everything happens around our extension.

Note: In this program we wont be using any HTML pages, without even HTML page we can run the WebExtension

Our Background script file

In this experiment we are using only one background script namely background.js; we should have included this in manifest.json else we cant use it.

```
createMenu();

function createMenu() {
// Get the list of search Engine
browser.search.get().then(buildContextMenu);
}
```

So our First step we are going to do in the background is to fetch the search engine and once we are done with fetching we are going to build the context menu.

Whenever the script is loaded, the very first method **createMenu()** is called.

In this method we have done steps to get the search engine **browser.search.get()**.

.then(some_function_name) represents once the previous process is done successfully jump to the method *some_function_name*

So in our case once we have got the search engine list using WebExtension API we will be going to the function names **buildContextMenu**

```
function buildContextMenu(searchEngineGot) {

// Iterating Search Engine One by One
for (let searchEngine of searchEngineGot) {

// Create ContextMenu Item
  browser.contextMenus.create({
    id: searchEngine.name, // searchEngine name is set as id.
    title: searchEngine.name,  // searchEngine name is set as title.
    icons: {
      "32": searchEngine.favIconUrl  // searchEngine favicon url is set as icon
    },
    contexts: ["selection"] // context is when we selected a word
  }); // End of contextMenu creation

} // End of iteartion of SearchEngine List

} // End of method
```

Once we got the list of searchEngine successfully we are going to method named **buildContextMenu**. The parameter which we will get here is list of SearchEngine Object. This is defined by WebExtension API and we should follow the params as it is.

In this method we are having so many searchEngine's in **searchEngineGot**, so we are iterating one by one.

In one searchEngine we are having **name** and **favIconUrl** value. Additional values are also present but we are not using it. We are then creating the contextMenu with **name** as the *id* and *title*. Then we have to **favIconUrl** which we are using as icon to dispaly the menuitem.

```
//When Menu Item is clicked we will goto function menuItemClicked
browser.contextMenus.onClicked.addListener(menuItemClicked);
```

After the menu items are created our next step is to listen whenever an menu item is clicked. This is also provided in WebExtension API. Whenever a menu item is clicked we can have a listener and pass the values to listener.

```
function menuItemClicked(menu_info_object, tab_object){
// First we have to check whether there is selectionText and also the menuItemId, both shou
```

```
    if(menu_info_object != null && menu_info_object.hasOwnProperty('menuItemId') && menu_info_
      browser.search.search({
        query: menu_info_object.selectionText, // Selection text is the text which we selecte
        engine: menu_info_object.menuItemId // is the searchEngine.name which we defined prev
      });
    }
  }
```

We have created a Listener named menuItemClicked; so whenever an menu item is clicked our code flow control will be passed here. We will be getting two objects, one is menu item information object and another one is tab object.

First we will be checking whether we have the menuItemId and the selected text. Both of them should be not null. And we can also have a additional check whether menuItemId we got matches with the list of search engine we have in our machine.

Once all the preliminary checks are done we will be at our final stage. Now our only step pending is to use the query text and menuitemId(search engine) and search in the respective Website. *browser.search.search* will accept the JSONObject, where *query* key is mandatory here we will be passing the selectedText and for search engine we have *engine* key.

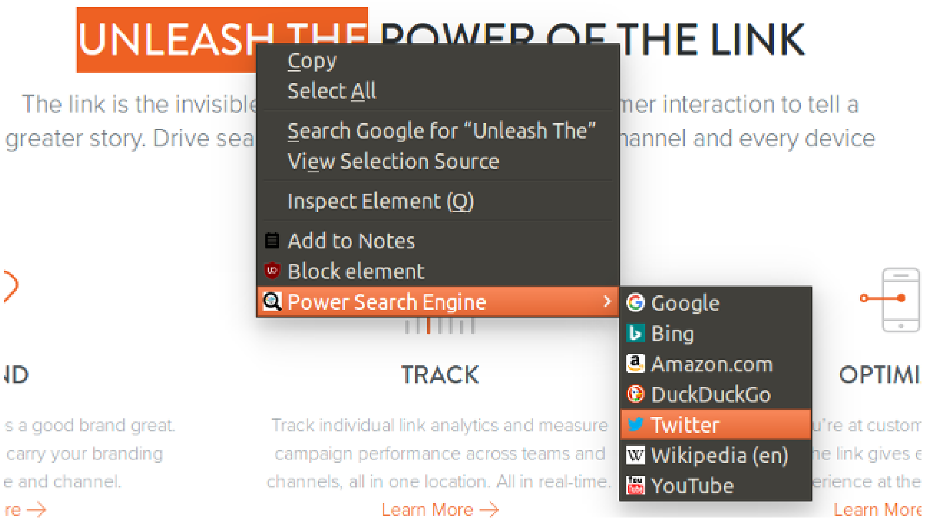


Figure 13: Power Search Addon Demo

Just in case, if you want to search in the current tab itself instead of opening in new tab, you have to make slight changes.

- First in permission you additionally have to get **"activeTab"** Permission

- Second you have to make small modification in the *browser.search.search* API.

```
browser.search.search({
      query: menu_info_object.selectionText, // Selection text is the text which we selecte
      engine: menu_info_object.menuItemId, // is the searchEngine.name which we defined pre
      tabId: tab_object.id // We will be passing the current tab id here.
    });
```

Exercise

To learn More about Search API [1] and ContextMenu API [2], visit MDN link given below.

Your simple task here to find out the various search engines (around 15-20). Make them listed in your ContextMenu. This task will involve you to understand the basic how the search Query params works.

For Example: In Wikipedia if you are search **hello world** you will be getting the URL like below

Wiki URL: https://en.wikipedia.org/w/index.php?search=**hello+world**&title=Special%3ASearch&go=Go

In this only the highlighted part will be changing everytime. Likewise find for other search engines also and create your extension. You may have to decide what title you have to give, id you have to give.

You may also have to do some little parsing.

Optional: It will be great if you can share your code or blog about this learning in twitter. Our Mozilla Webextension twitter handle is (@MozWebExt) and make sure you use hashtag #WebExtLearn when you are tweeting about this learning.

References

- [1] https://mzl.la/2q13lSO
- [2] https://mzl.la/2J3k7tc

Experiment 3: Building our Tabs Closer

We have seen previosly how to search using different search engine available in our browsers. It should interesting and easy one right ? Hope the execrise you tried might have left to finding the URL of different search engines and what are all the extra information they are sending as request params.

Problem statement and Solution

Problem Statement

Now when we are star surfing internet, we end up opening atleast 10-20 tabs, because each and every page have some many hyperlinks which we love to learn and come back, but in reality hyperlinks in that pages leads to different page again there are so many hyper links. In such a case if you are in college or office closing the social media websites (some of us do, we dont love to show it to our boss) will be difficult have to close so many tabs after finding them.

Solution

The simple solution will be to have the list of social websites we visit and with one click, we can close all the tabs. The another solution will be to list all the tabs we have close from an pop-up one by one; this will work great when we are having more than 20 tabs.

How we can capture this

Breaking according to two solutions.

Showing the pre-defined Social Network sites

In this type, we will be pre-defining the one social media website which we visit often. We will be having the icon to close that particular social media website alone. On clicking the icon, all the tabs having that social media website URL should be closed.

Fetching all opened URL

Another approach will be fetching all the opened websites, listing them one by one and when we click on close button we can close them. Just incase we need to visit them we will have another button to visit the particular tab.

In this execrise we will be learning the following list of API

- Tabs.query (To fetch all tabs)
- Tabs.remove (Close)
- Tabs.move (To display the tab clicked)
- Browser Action Icon
- Browser Action pop-up

Building Blocks of Our Social Media Tab Closer

First we will take a look at the API we will be using in this Execrise.

Getting started with Basics of API of Social Tab Closer

Starting the Closing Operation

First step is we need to click on the browserAction icon. As the name suggest browserAction icon will have to scope throughtout the browser and it is not specific to any one particular tab. So whenever we click on it irrespective of tabs we can do the procedure. Once we have click the icon we need to capture the click action.

We have browserAction API for capturing this. *browserAction.onClicked* will be used to listen to click action on that icon.

```
browser.browserAction.onClicked.addListener(action_jump_function_name);
```

Fetching the twitter tabs

In this case we will be knowing the website URL of Twitter, it is nothing but https://twitter.com/ . So our first step will be to get all the tabs where Twitter is loaded. Our WebExtension *tabs.query* have lot of combination to get(or fetch) the tabs loaded, in this case we are going to fetch based on **URL**

```
browser.tabs.query({url: "*://*.twitter.com/*"});
```

Some times we wont know whether its *http* or *https* so we are using the regex (Regular Expression) *://* before twitter.com and also we may be having anything loaded so we have /* at the end of .com

Close the tabs

Once we have got the tabs (or single tab) we need to close them. We have to get the tabid from the tab object we got, based on that only we will be closing the tabs.

```
var removing = browser.tabs.remove([tab_id1, tab_id10, tab_id15]);
```

Assembling parts of our Extension

So now our task is to just write all the above code in manifest.json and background script file. Mos to fhte things are coverd.

Writing manifest.json

```
{
  "manifest_version": 2,
  "name" : "Social Tab Closer",
  "description" : "Close the pre-defined Social Media Tabs",
    "version": "1.0",
  "icons": {
      "96": "icons/icon-96.png"
    },
    "permissions" : [
      "tabs"
    ],
  "background": {
      "scripts": ["background.js"]
    },
    "browser_action": {
      "default_icon": {
        "96": "icons/icon-96.png"
      }
    }
  }
```

Here changes we made will be getting the tabs permission. And also we have to define the brower_action icon alone. We wont make any other drastic changes compared to previous manifest.json.

Writing our background.js

We will be splitting the background javascript into parts.

- Listen to browser Action Icon
- Fetch the twitter tabs.
- close the tabs.

Listen to browser Action Icon

```
browser.browserAction.onClicked.addListener(icon_clicked);
```

Our first step will be to listen whenever our browserAction icon is clicked. Once it is clicked we have to pass the control to the method (icon_clicked) which we have defined.

Fetch the twitter tabs

```
function icon_clicked(tab){
  var querying_tabs = browser.tabs.query({url: "*://*.twitter.com/*"});
  querying_tabs.then(close_tabs);
}
```

Our next step will be to fetch(query) the tabs where twitter.com is loaded, so we will be using the regex which we discussed before and fetch the tabs with the url pattern.

Close the tabs

```
function close_tabs(twitter_tabs){
var array_of_twitter_tabid = [];
  for(let single_twitter_tab of twitter_tabs){
    array_of_twitter_tabid.push(single_twitter_tab.id);
  }

  browser.tabs.remove(array_of_twitter_tabid);
}
```

Our Next step is to get the tab id with the tabs we have fetched. We are first iterating each and every tab object which we are getting in tab array as a result of query using URL. Then while iterating we are getting the **id** and inserting in the array. After we finish the iteration we are just using the close (remove) API and closing the tabs.

Building Blocks of All Tabs Closer

We are going to almost use same set of API which we used for our social tab close. Here we will additionaly have a very small change in manifest.json.

Getting started with Basics of API of All Tab Closer

Adding popup html page

Previously we have mentioned the browserAction icon and just left. Now in manifest.json we are additionally going to add the html page which has to be shown when browserAction icon is clicked and also the title which has to be shown when we move our mouse above browserAction icon.

Note:We can either Click on the browser Action icon or we can show the HTML page

Fetching the all the tabs

Previously we have passed the url and fetched the tabs where twitter.com is loded. Now simply we dont need to pass anything so we will be getting all the tabs.

```
browser.tabs.query();
```

Close the single tab

We are going to close the tabs one by one, so instead of passing the array of tabIDs we can pass only one tabID also.

```
var removing = browser.tabs.remove(tab_id1);
```

Assembling parts of our All Closer Extension

Lets start with manifest.json, for now will share the whole content, in future we can slowly start sharing only the parts which are having difference.

manifest.json look

```
{
  "manifest_version": 2,
  "name" : "All Tab Closer",
  "description" : "Close All Tabs",
    "version": "1.0",
  "icons": {
      "96": "icons/icon-96.png"
    },
    "permissions" : [
    "tabs"
    ],
  "background": {
      "scripts": ["popup/script.js"]
    },
    "browser_action": {
      "default_icon": {
        "96": "icons/icon-96.png"
      },
      "default_title": "List all tabs",
    "default_popup": "popup/tabs_list.html"
    }
  }
```

So our change will be mostly in the **browser__action** part. We have additionally added two more keys for it. Most of the parts remains quite familier and its based on this Add-on.

Building our tiny HTML page.

We need a HTML page which will be acting as a popup when we are going to click on **browserAction** icon. So we need to design very small page.

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8"/>
    <title>List of Tabs</title>
</head>
<style>
    .single_div {
        background-color: lightgrey;
        border: 5px solid rgb(11, 12, 11);
        padding: 25px;
        margin: 25px;
    }
    </style>
<body>
  <div id="tabs_list" style="width: 300px">
    </div>
  <script type="text/javascript" src="script.js"></script>
</body>
</html>
```

Our HTML page is having some css for design and most important part is the div with id *tabs_list*, we will be creating all our tab details and will show here.

And we have also included the Javascript file at the end. This same JS file will be acting as background script file.

Included JS and Background JS

Our first action once the javascript file is loaded is to collect all the tabs which are opened. We are processing this action in **startFetchingTabs** method

```
startFetchingTabs();

// Fetch all the tabs
function startFetchingTabs() {
  var querying = browser.tabs.query({});
  querying.then(create_list_tabs, onError);
}
```

After we have collected the tabs our action will be to list them in the popup browserAction page. So we are itearting tab object one by one. tab object will be having the details like tabID, tabTitle and tabURL. We as of now need tabID and tabTitle.

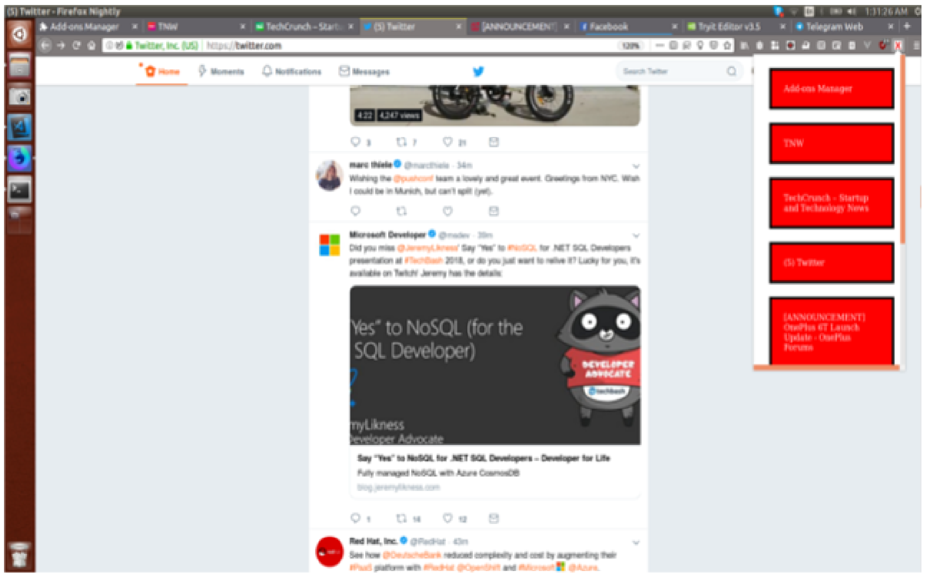

Add-ons Manager
TNW
TechCrunch - Startup and Technology News
(5) Twitter
[ANNOUNCEMENT] OnePlus 6T Launch Update - OnePlus Forums

Figure 14: Popup Before Closing any tab

First we will be deleted all the already listed tabs in the pop-up HTML page, then after that we are creating the div one by one for all the tabs opened.

```
function create_list_tabs(fetched_tabs) {
  // delete already existing child nodes.
  if (document.getElementById("tabs_list").childNodes.length > 0) {
    var myNode = document.getElementById("tabs_list");
    while (myNode.firstChild) {
      myNode.removeChild(myNode.firstChild);
    }
  }

  // Create new child nodes.
  for (let single_tab of fetched_tabs) {

    var created_div = document.createElement("div");
    created_div.style.width = "200px";
    created_div.style.background = "red";
    created_div.style.color = "white";
    created_div.innerHTML = single_tab.title;
    created_div.id = single_tab.id;
    created_div.className = "single_div";
    created_div.addEventListener("click", function () {
      close_tab(single_tab.id);
    }, false);

    document.getElementById("tabs_list").appendChild(created_div);
  }
}
```

We have created the div in such a manner, the tab Title will be displayed to the user. And each and every title is associated with tabID. When we click on any part of the div, the tab ID is passed to **close_tab** method and there we will start the process of closing the tab.

```
// Close the tab and recreate the popup html
function close_tab(tab_id_to_close) {
  browser.tabs.remove(tab_id_to_close);
  startFetchingTabs();
}
```

As discussed before, once the title is clicked, the respective id is passed and we will be closing the tab with that particular ID after that we will again be creating the popup browserAction HTML page.

Exercise

Make sure to visit Mozilla Developer Network (MDN) to learn more about the above API.

- Close API [1]
- Query API [2]
- Capture Tab [3]
- Reload Tab [4]
- Browser Action [5]

There are lot of things related to tabs API to be explored. We have as of now have used the title and showed the list of tabs. But sometimes Title alone wont be enough, we have an API to capture the current instance of the tab as image, use this API instead of title make sure to show the image and on image have two options one for reload and another for close. So based on the button clicks use Reload API or Remove API.

Optional: It will be great if you can share your code or blog about this learning in twitter. Our Mozilla Webextension twitter handle is (@MozWebExt) and make sure you use hashtag #WebExtLearn when you are tweeting about this learning.

References

- [1] https://mzl.la/2QZ1PfB

- [2] https://mzl.la/2PfiZYW
- [3] https://mzl.la/2yLRH2g
- [4] https://mzl.la/2yrb5SZ
- [5] https://mzl.la/2OCdEf1

Experiment 4: Building our PDF WebPage Extension

So we were learning about fetching the tabs Closing them one by one. There are times in our life where we need to do something later. Yes we need to do lot of things later. In this chapter we will be learning how to save so we can check it later. We learnt more about browserAction icon previously now we will be learning opposite to it (not exactly just for fun) PageAction icon, we are seeing it in lot of pages think of closing tab button, bookmark button, what they are doing ? what is their scope ?

Problem statement and Solution

Problem Statement

We read each and every day lot of articles. Think of pages you read in wikipedia, sometimes you will dive deep into it and wanted to spend large amount on that article but you might have closed it and never opened again. This is the most saddest thing can happen when you see that article again and start reading.

Solution

We have very simple solutions for this, we can save it to Pocket app (https://getpocket.com/), or you can save the whole page by selecting all the text copying it and saving it in the text file which is seriously boring thing you can do. One better option is to make the pages as PDF when we have a readable view.

How we can capture this

We have very rich tabs WebExtension API which with which we can know whether that particular tab is in readable mode or not, we can know when there is change in the tab, with simple click we can make the page as PDF.

Building Blocks of PDF WebPage Extension

We will get straight to the API we will be using. Our flow is simple, whenever there is change in tab we will be checking whether current tab have a readable formatted page, if it is available we will be enabling the pageAction icon, whenever there is a click in pageAction icon we will be saving it as PDF in our system.

Capturing change in tab

Changing the tab means activating from one tab to another tab. In Tabs API we are having method to captue this.

```
  browser.tabs.onActivated.addListener(listen_to_tab_change);

  function listen_to_tab_change(activeTabInfo){
    //do Something
  }
```

So whenever there is a change in tab, if we want to capture it we will be having an activeListener method. In this method we will be doing whatever we want.

Similiarly we have to check whenever there is an update in the same tab(current tab itself). Like if the current page is www.google.com and now we moves to wikipedia site where we may some downloaded content, so we have to capture that also.

```
browser.tabs.onUpdated.addListener(listen_to_tab_update);
 function listen_to_tab_update(activeTabInfo){
    //do Something
  }
```

Finally we have to listen when there is a new tab create.

```
browser.tabs.onCreated.addListener(listen_to_tab_create);
function listen_to_tab_create(activeTabInfo){
    //do Something
  }
```

These above 3 are the cases we need to capture. In all the 3 cases the follow will be same. We will first check whether the tab updated or created or changed to is in ReadMode format.

Checking is current Tab reabable format

So the tab object will be having lot of details like id, index,title, url and so on. One of the value is **isArticle**

If the value of isArticle is true, then it means we have readable format (ReadMode is available) tab. In that case we can download as PDF.

Showing the pageAction icon

So we came to know the current page can be in read mode, which means we can have the option to download the page as PDF. At this stage we may need to have user intervention. Remember we learned about browser Action icon during **Building our Tabs Closer** , similar we have pageAction icon (which we can consider limited edition of browser Action icon).

The major difference between browserAction icon and pageAction icon is the scope of availability. browserAction icon is available outside the addressbar icon which means it is not limited to that particular tab alone while the pageAction icon is inside the addressbar which means it is very specifically limited only to that tab alone.

So in our case we will be showing the pageAction icon whenever the current page which we are viewing is having a Reader mode.

Downloading the page

The only final step now is we have to download (save) the page as PDF. This is just one method call, all parameters are optional so we need not pass anything to just save without any modification.

```
browser.tabs.saveAsPDF({});
```

So our next step is to start assembling all the API's into single WebExtension.

Assembling parts of PDF-IT Extension

manifest.json look

Most parts of our manifest.json looks very similar. Only addition to ther permission is **activeTab**. And we are adding a new part of json **page_action** which is similar to browser action.

```
{
"manifest_version": 2,
"name" :"Save as PDF",
"description" : "Saves pages as pdf",

"homepage_url": "http://iamvp7.github.io/",
  "version": "1.0",

"icons": {
    "48": "icons/page-48.png",
    "32": "icons/page-32.png"
  },
  "background": {
    "scripts": ["background.js"]
  },
  "permissions" : ["tabs" , "activeTab"],
  "page_action": {
    "default_icon": "icons/page-32.png",
    "default_title": "Download as PDF"
  }
}
```

Adding tab state changes listeners

Our next step is adding the listeners when there is update in tab, creation in new tab, activation in tab.

```
browser.tabs.onUpdated.addListener(listen_tab_update);

browser.tabs.onActivated.addListener(listen_when_tab_activated);

browser.tabs.onCreated.addListener(listen_when_tab_create);
```

After listen we will have same set of actions, but different listeners have different structure of call back functions. So we need 3 methods.

Adding Implementation for listeners

So we will be adding the Listener for each and every type we have mentioned. In some cases we will be getting the current tab id directly, in some cases we will be fetching them.

Adding onUpdate Listener

When there is any update in current tab we will be listening the change in this method for this extension.

```
function listen_tab_update(tabId, changeInfo, tab) {
  getCurrentTabDetails();
  }
function getCurrentTabDetails(){
    var querying = browser.tabs.query({currentWindow: true, active: true});
    querying.then(getInfoForTab, onError);
  }

  function getInfoForTab(tabs) {
    if (tabs.length > 0 && tabs[0] != null && tabs[0].isArticle) {
      browser.pageAction.show(tabs[0].id);
    }
  }

  function onError(error) {
    console.log(`Error: ${error}`);
  }
```

The callback in this function will be having 3 parameters tabId in which the update action has taken place, change info option and the current state of the tab object. Among the all 3 we have got we will be needing the **tabId** alone, so for the current tab we will be showing the **pageAction icon**.

But we also need the details like whether the current tab is in readable format. So first in **getCurrentTabDetails** we will be querying the active tab from current active window, we will be getting the list of tab array. From that tab array we will iterate each obejct and will be getting the first one alone.

Adding onActivated Listener

So we have to add a listener method for tab activated. In this we will be recieving only one parameter (activeInfo Object), where we wont get the tab details. So for this we will be querying the current tab and from that we will be receiving the tabId. Similar to what we have done during close tabs we will do here.

```
function listen_when_tab_activated (activeInfo) {
  getCurrentTabDetails();
}
```

Here also similarly we will be call the **getCurrentTabDetails**

Adding onCreated Listener

So our final listener method is required when we are creating the new tab.

```
function listen_when_tab_create(tab){
  getCurrentTabDetails();
}
```

Here also similarly we will be call the **getCurrentTabDetails**

The only thing which we are doing in each of the listener method is to show the **pageAction** icon. As discussed previously, when the user clicks we will be saving it as PDF

Adding pageAction listener

So when the pageAction icon is clicked, we have to do something. So we will be listening from backgroun.js and will wait for pageAction click.

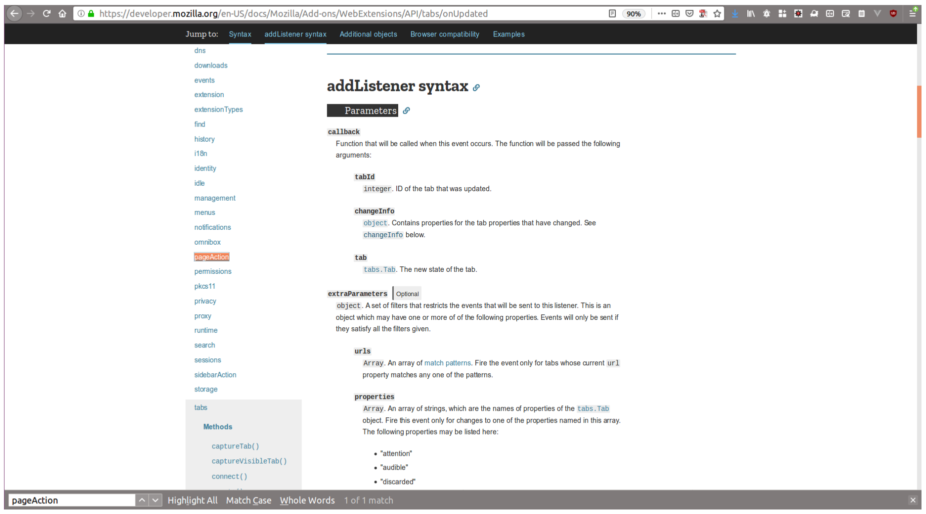
```
browser.pageAction.onClicked.addListener(doPageActClickProc);
```

Save as PDF

So our final step once the pageAction icon click is just to save the current readable page as PDF.

```
function doPageActClickProc(){
  // browser.tabs.toggleReaderMode();
  browser.tabs.saveAsPDF({});
}
```

So when you are running your extension you will be seeing the icon near the bookmark icon inside the address bar. Like below



addListener syntax

Parameters

`callback`
Function that will be called when this event occurs. The function will be passed the following arguments:

`tabId`
`integer`. ID of the tab that was updated.

`changeInfo`
`object`. Contains properties for the tab properties that have changed. See `changeInfo` below.

`tab`
`tabs.Tab`. The new state of the tab.

`extraParameters` Optional
`object`. A set of filters that restricts the events that will be sent to this listener. This is an object which may have one or more of of the following properties. Events will only be sent if they satisfy all the filters given.

`urls`
`Array`. An array of match patterns. Fire the event only for tabs whose current `url` property matches any one of the patterns.

`properties`
`Array`. An array of strings, which are the names of properties of the `tabs.Tab` object. Fire this event only for changes to one of the properties named in this array. The following properties may be listed here:

- "attention"
- "audible"
- "discarded"

Figure 15: PDF IT PageAction Icon

When we click on the icon we will be getting the pop-up for saving the file as pdf.

Exercise

To learn More about below list of API, visit MDN link given below.

- onUpdate [1]
- onActivate [2]
- onCreate [3]
- pageAction [4]

Your simple task is to create a small social share button. So when the user clicks on the pageAction icon he should have options to share it to various social media.

Optional: It will be great if you can share your code or blog about this learning in twitter. Our Mozilla Webextension twitter handle is (@MozWebExt) and make sure you use hashtag #WebExtLearn when you are tweeting about this learning.

References

- [1] https://mzl.la/2OngPCk

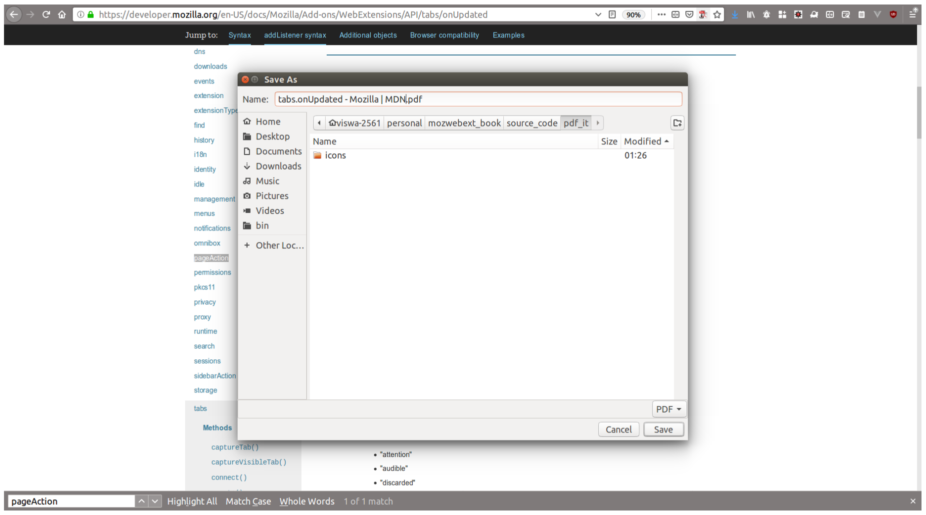

https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/API/tabs/onUpdated
Save As
Name: tabs.onUpdated - Mozilla | MDN.pdf
viswa-2561 personal mozwebext_book source_code pdf_it
Name Size Modified
icons 01:26
Home
Desktop
Documents
Downloads
Music
Pictures
Videos
bin
Other Loc...
PDF
Cancel
Save
pageAction
Highlight All Match Case Whole Words 1 of 1 match

Figure 16: PDF IT Save file

- [2] https://mzl.la/2JDMsqq
- [3] https://mzl.la/2EYAnx9
- [4] https://mzl.la/2SHe02h

Experiment 5: Building our Water Notifier Extension

Humans depends on a lot on notification system. The simplest thing and more like a hello world program for WebExtension is building Notification system. I got started with sending simple notification.

Problem statement and Solution

Problem Statement

We are in a very busy life where we even forget to drink the recommended about of water for our body. This leads to dehydration and we start performing badly. One of the simplest solution is being well disciplined and drinking 1 glass of water for every 45 mins, this wont happen unless we have self control. Another solution is having alarm in mobiles, sometimes if it rings people around us will get irritated. Another solution which can be done with WebExtension is sending us notification in our browser for every 45 mins so we know some actions to be done.

Solution

So our solution is simple, have a alarm in our system so we will be sending the notification every 45 mins.

How we can capture this

Our proposed system has clearly mentioned we need to have alarm in our browser, and to show something in UI for that lets have a small Notification UI. We are having the following list of WebExtension API.

- alarms API [1]
- notifications API [2]
- Runtime API [3]

In our system we need to set alarm for every 45 mins from the time we started our browser, so we need to find out when the browser is started so for that we need the help of **Runtime API**. This Extension is one of my favorite as I started learning WebExtension development with this.

Building Blocks of Our Water Notifier Extension

We can develop this extension very easily. Our building blocks for this is very simple and we will use simple 3 API for the whole process.

Finding the Browser Startup & Add-ons Installation

Our first step is to find out when the Add-ons is installed or when the Add-on is opened. From that time we have to calculate 45 mins and send the Notification.

```
// Add a method what to happen when Addon is installed
browser.runtime.onInstalled.addListener(handleInstalled);

// Add a method what to happen when browser starts
browser.runtime.onStartup.addListener(handleStartup);
```

We have two different Listener. First one is when the Extension is installed and the second one is when we are starting our Browser.

Setting the Alarm

Our next step after finding the Add-ons installation or browser startup is to set the alarm.

```
browser.alarms.create("my-periodic-alarm-install", {
  when,
  periodInMinutes
});
```

Our good practice is to give first the name of the alarm. In our above example **my-periodic-alarm-install** is the name of the alarm.

After that we will be mentioning the value of **when**, the time at what the very first alarm should be triggered. Secondly we will also be giving the value of **periodInMinutes** this is set to 45 because we wanted to trigger the alarm each 45 mins.

If we already having alarm with same name for this extension, then the previous alarm will be deleted and new alarm will be set.

Capture the Alarm

Once we set the alarm our next step is to listen in the background when our alarm is triggered and we have to capture it and do our processing. Even in our mobiles, when the alarm signal is trigger it is captured then some sounds (alarm tone) is made to play, we should understand when alarm is triggered it means signals are triggered not the sound.

```
  browser.alarms.onAlarm.addListener(handleAlarm);
```

We have an listener method in alarms API for this. So we will have method and do the actions we need to do.

Fire the Notification

After we have captured the alarm triggered our final step is to show the notification.

```
browser.notifications.create(waterNotification, {
    "type": "basic",
    "iconUrl": browser.extension.getURL("icons/icon-96.png"),
    "title": "Time for Water",
    "message": "Drink enough water to stay healthy"
  });
```

We will have the notification id at the place of **waterNotification**. As of now Firefox is supporting only basic type of notification. We can have a small **title** text and a descriptive **message** alone in our notification.

Note in case we call notification creation more than once in rapid succession, Firefox may end up not displaying any notification at all.

Assembling our Water Notifier Extension Parts

So almost everything what we have discussed is available as a single code file (background.js). If we copy and paste with one addition of method thats enough to make this Extension work.

Writing manifest.json

Our manifest.json is very simple. Only change which is available in at the permission's part. We need alarms permission and notification permission.

```
{
"manifest_version": 2,
"name" : "Notification Alarm",
"description" : "Sending Notification and Alarm",
"homepage_url": "http://iamvp7.github.io/",

"version": "2.0",
"permissions" : [
  "notifications",
  "alarms"
  ],
"background": {
    "scripts": ["background.js"]
  }
}
```

Adding listeners

So as discussed before we are having two listeners in our program.

```
// Add a method what to happen when Addon is installed
browser.runtime.onInstalled.addListener(handleInstalled);

// Add a method what to happen when browser starts
browser.runtime.onStartup.addListener(handleStartup);
```

Adding Implementation for listeners

Both the listeners have different parameters in the callback function. So we have to define two methods.

Listener for startup

```
// Method handling when the browser starts
function handleStartup() {
  createAlarm();
}
```

So this method wont have any params. The only action we are doing from here is to moving the call where we can create the alarm.

Listener for installation

```
// Method to handle when the addon is installed
function handleInstalled(details) {
  createAlarm();
}
```

This call back method will have installation details like id, previous version. Here also we are just moving to create the alarm method, no other action is done.

Create the alarm

So after listening to installation and startup our step is to create the alarm.

```
function createAlarm(){
  const when = nearestMin30();
  browser.alarms.create("my-periodic-alarm-install", {
    when,
    periodInMinutes
  });
}

// Finding the nearest 30 Mins
function nearestMin30(){
  var now = new Date();
  now.setMinutes(d.getMinutes() + 45);
  return  now.getTime();
}
```

We will be finding the nearest 45 mins from startup / installation and we will be setting the alarm. The method **nearestMin30** job is to find the nearest 45 mins and return its milliseconds value.

Adding listeners for alarms

Our major one line of this program is adding listener for the Alarms. Once the alarm is set we should be listening for its trigger in background always. If the trigger is set we need to do our required action, in our case its pushing the notification.

```
browser.alarms.onAlarm.addListener(handleAlarm);
```

Push the notification

So our final step is to push the simple notification. A simple notification will come, so we can understand water has to be drank at that time.

Before installing the extension for testing bring the time to 1 min or 2 mins. After installing the Extension, wait for the mentioned time and a notification will be coming in soon.

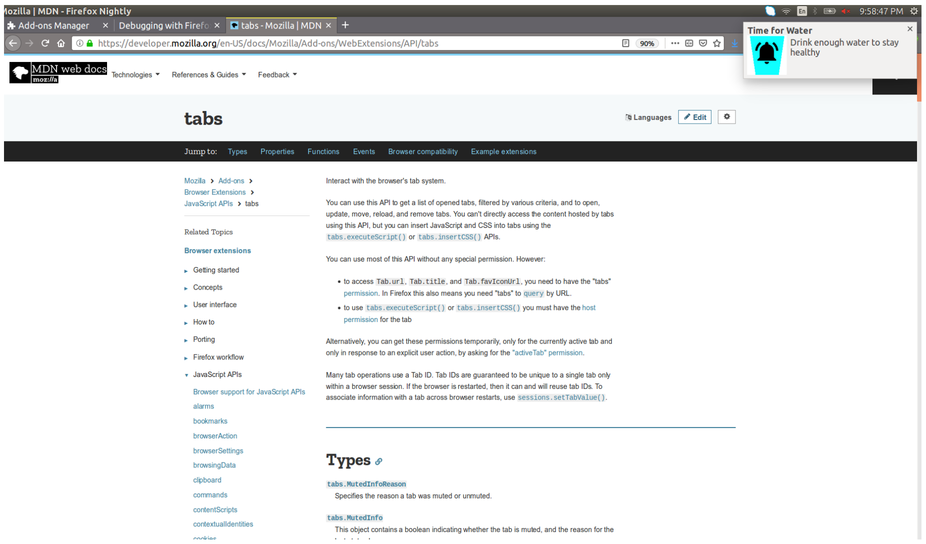
Exercise

To learn More about below list of API, visit MDN link given below.

- alarms API [1]
- notifications API [2]
- Runtime API [3]

Your simple task is to create a WebExtension which will have a pageAction icon, on click it we should make the article to be opened after 30 mins. This Extension will look like a simple version of Firefox Test pilot Snooze tab. In this we will using Open tab API, Alarms API, pageAction API.

Optional: It will be great if you can share your code or blog about this learning in twitter. Our Mozilla Webextension twitter handle is (@MozWebExt) and make sure you use hashtag #WebExtLearn when you are tweeting about this learning.



Time for Water

Drink enough water to stay healthy

tabs

Interact with the browser's tab system.

You can use this API to get a list of opened tabs, filtered by various criteria, and to open, update, move, reload, and remove tabs. You can't directly access the content hosted by tabs using this API, but you can insert JavaScript and CSS into tabs using the `tabs.executeScript()` or `tabs.insertCSS()` APIs.

You can use most of this API without any special permission. However:

- to access `Tab.url`, `Tab.title`, and `Tab.favIconUrl`, you need to have the "tabs" permission. In Firefox this also means you need "tabs" to `query` by URL.
- to use `tabs.executeScript()` or `tabs.insertCSS()` you must have the host permission for the tab

Alternatively, you can get these permissions temporarily, only for the currently active tab and only in response to an explicit user action, by asking for the "activeTab" permission.

Many tab operations use a Tab ID. Tab IDs are guaranteed to be unique to a single tab only within a browser session. If the browser is restarted, then it can and will reuse tab IDs. To associate information with a tab across browser restarts, use `sessions.setTabValue()`.

Types

`tabs.MutedInfoReason`
Specifies the reason a tab was muted or unmuted.

`tabs.MutedInfo`
This object contains a boolean indicating whether the tab is muted, and the reason for the

Figure 17: Water Notification

References

- [1] https://mzl.la/2zBlyuT
- [2] https://mzl.la/2zrMn4r
- [3] https://mzl.la/2zqzxDE

Experiment 6: Building our Shortcuts for Youtube Controller

Problem statement and Solution

Problem Statement

Think of the busy working day. To relax you should be playing some of your favorite songs in youtube. And as you continue listening you will be start browsing the internet. It is very easy to open so many tabs and it will become difficult to find the youtube tab which you opened long back. You may love to skip the song when some irritating song comes or you would like to switch between pause and play when someone comes and talks to you.

Solution

In such a case we should be quickly able to find the youtube tab and do the action. We can have two types to do this.

First solution is we can have a small pop-up with different buttons like play/pause and next song. Second solution is to make use of the shortcuts. Yes we can build our own shortcuts in Firefox.

In this tutorial we will be learning about creating our own Shortcuts in WebExtensions.

How we can capture this

As our proposed solution is related to shortcuts we have a WebExtension API named commands [1] which we can use to build our own shortcuts for Firefox. And after creating shortcuts we have programmatically click the next song button or play/pause button in youtube. For achieving this we need to write some minimal script to click the buttons. After we write the script we have to execute them, to this in the Tabs API we have a special method called **executeScript()** [2], similarly we also have for CSS named **insertCSS()** which we wont use as of now.

Building Blocks of Our Youtube Controller Extension

Define the Shortcut

Our very first task will be to define the shortcut which we are going to use for our actions. We need to define unique shortcuts for both the set of actions. Our shortcuts will be defined in the manifest.json

```
"commands": {
  "next-song": {
        "suggested_key": { "default": "Ctrl+Alt+N" },
        "description": "Go to next song"
    }
}
```

Each and every command will have a name, in our example **next-song** is the name of the shortcut. Each command will have two properties *suggested_key* and *description*. **Description** is the just one line about this particular shortcut. And **suggested_key** will be having the set of shortcuts based on the operating system. List of keys which can be present in the suggested_key json is shared below. **default** is not specific to any platform it will be used in all platforms.

```
"default", "mac", "linux", "windows", "chromeos", "android", "ios"
```

Listening to commands

In our background script we should be listening when our command is triggered. We have **onCommand.addListener** method for this. In this for the call back function we will be receiving the **command_name** as the argument. Based on the **command_name** we have to execute different scripts.

```
  browser.commands.onCommand.addListener(on_command_received);

  function on_command_received(command_name){
    // Do something based on command_name
  }
```

Execute the Click Scripts

So we have manually find out the button id in the youtube. It should be done before starting the development of extension. It is one time process only.

```
// Script for clicking the next button
document.querySelector(".ytp-next-button.ytp-button").click();

// Script for clicking the play/pause button
document.querySelector(".ytp-play-button.ytp-button").click()
```

First part of executing the script is to find in which tab the *youtube* is running. To this we have already using tabs.query API. We just need to change the value of the variable **url**

```
  var querying = browser.tabs.query({
    url: "*://*.youtube.com/*"
  });
```

So from this we will be getting the tabs object, in that for the first tab we will be getting the ID and executing the script on that tab alone.

```
  var executing = browser.tabs.executeScript(
    tabID, {
      code: commandToExecute
      });
```

Here the value of **commandToExecute** is the script which we have got earlier for programatically click the play/pause or next song button.

Assembling our Power Search Extension Parts

manifest.json file

So as usual we will be starting with our manifest.json file and will make the single script file.

```
{
    "manifest_version": 2,
    "name": "Youtube controller Using Shortcuts",
    "description": "Control Youtube play without visiting the tab.",
    "homepage_url": "http://iamvp7.github.io/",
    "version": "1.0",
    "icons": {
        "96": "icons/icon-96.png"
    },
    "background": {
        "scripts": ["background.js"]
    },
    "permissions": [
        "*://*.youtube.com/*",
        "tabs"
    ],
    "commands": {
        "next-song": {
            "suggested_key": {
                "default": "Ctrl+Alt+N"
            },
            "description": "Go to next song"
        },
        "pp-song": {
            "suggested_key": {
                "default": "Ctrl+Alt+P"
            },
            "description": "play or pause the song"
        }
    }
}
```

In this manifest.json in permission we will be requiring the website permission of youtube and tabs because we are going to execute the script where youtube in running.

And the newest part of manifest.json you will notice is the **commands** object. We will be defining the what are the commands we will be using and what are their shortcuts and their description in manifest.json itself.

Our Background script file

```
var commandToExecute = '';

browser.commands.onCommand.addListener(on_command_received);
function on_command_received(command_name) {

    if (command_name == 'next-song') {
        commandToExecute = 'document.querySelector(".ytp-next-button.ytp-button").click()';
    } else if (command_name == 'pp-song') {
        commandToExecute = 'document.querySelector(".ytp-play-button.ytp-button").click()';
    }

    // Query tabs based on url
    var querying = browser.tabs.query({
        url: "*://*.youtube.com/*"
    });
    querying.then(executeOnTab, onError);
}


function executeOnTab(tabs) {
    var tabID = tabs[0].id; // getting the first tab ID alone

    var executing = browser.tabs.executeScript(
        tabID, {
            code: commandToExecute
        }
    );
    executing.then(onExecuted, onError);
}

function onError(error) {
    console.log(`Error: ${error}`);
}
```

```
function onExecuted(result) {
    console.log(`We executed script in the first youtube tab`);
}
```

So our background script file is well explanatory and is organized well. For learning quickly without checking the code sharing this.

First we will be having a listener (**onCommand.addListener**) to listen when the shortcut is triggered. We have a callback function (**on_command_received**) for this listener. This callback function will be receiving the command as the argument.

Then based on the command (either *next-song* or *pp-song*) we will be assigning the script which we need to execute. After that our process is to find the first youtube tab. Using the **tabs.query** method we will be getting the **tabId** of first youtube tab.

As we are having the required tabID and the script to execute we will be using **browser.tabs.executeScript** method to execute the script and make our necessary changes.

Note: Since this extension is purely based on commands it is not possible to add the screenshot of UI.

Exercise

To learn More about Commands API [1] and tabs.Execute [2], visit MDN link given below.

We are having small limitation in this. Our system will work when we are having only one youtube tab. But in the ideal world people used to have more than one youtube tabs. So instead of using the commands API, try to create the browserAction popup with the screenshot of the images. You can get the screenshot of the tab using **tabs.captureTab()** [3].

Another exercise is, try to find out other amazing site likes this for listening to music. Replicate the same action for them. It will be one of the best Extension you can use and many will love to use them.

Optional: It will be great if you can share your code or blog about this learning in twitter. Our Mozilla WebExtension twitter handle is (@MozWebExt) and make sure you use hashtag #WebExtLearn when you are tweeting about this learning.

References

- [1] https://mzl.la/2zuxQoJ
- [2] https://mzl.la/2Lnf1aT
- [3] https://mzl.la/2D4QoPJ

Experiment 7: Building Quick Stackoverflow searcher

Problem Statement

You know very well the number of times you have been to stackoverflow for searching throughout this day. Wont it be great to have a quick searcher for the stackoverflow. In this experiment we will be building a simple Extension where we can type the shortcut in the newtab and the text to be searched so we will be directly taken to the search page.

Solution

One of the solution we can see which is similar to both chrome and Firefox is below.

Step 1: Visit https://stackoverflow.com/search?q= Step 2: Bookmark this page. Step 3: Edit the property of the bookmark. Step 4: Give the shortcut (Keyword) for this bookmark in the bookmark manager.

Another solution using WebExtension, which is more similar to the mentioned above. We can do lot of things using this. We will be defining the keyword first, and in the newtab when we are giving the keyword space the search string and hit enter we will be going to stackoverflow search page. Using WebExtension we are just replicating the above mentioned steps.

How we can capture this

The address bar in the browser is called as omnibox. In WebExtension's we are having specific API to handle the omnibox. With this we can capture the letters being typed after our specified keyword, we can provide suggestion as we start typing.

Note: The very important thing we should note here is the keyword, when the same keyword is used by another WebExtension then the extension which is installed at last will be taking over the control.

Building Blocks of Our Quick Stackoverflow searcher

Define the Keyword

Our first step is to define the right keyword. Our **keyword** should be very unique and so it wont be used by other or taken by other WebExtensions. Our keyword is defined in the manifest.json

```
"omnibox": {
  "keyword": "sof"
}
```

Adding Listener to our keyword

Once we type our keyword of the extension and space the characters what we are typing will be search string. We can capture this (but its not mandatory) and can show the default search suggestion.

```
  browser.omnibox.onInputChanged.addListener(text_typed);

  function text_typed(typed_text, suggest){
   // Do something
  }
```

So the listener callback will be giving two arguments, first is the text which is being typed and second is the suggest callback method.

Adding Listener when text is entered

This listener is mandatory for this extension. When the user types the string and then hits enter we need to take the control and do the action we have defined. So **omnibox.onInputEntered** is mandatory for this type of extension. All the post action is done here. The listener call back method will have two arguments first is the entered string and second is disposition object.

```
browser.omnibox.onInputEntered.addListener(text_entered);

function text_entered(entered_text, disposition) {
    // Do something
}
```

Assembling our Power Search Extension Parts

manifest.json file

In our manifest.json file we are making almost everything as usual, the additional thing we have added is for giving the keyword for this extension.

```
{
  "name": "Search in Stackoverflow",
  "description": "Shortcut to search quickly in StackOverFlow",
  "version": "1.0",
  "applications": {
      "gecko": {
          "strict_min_version": "52.0a1"
      }
  },
  "omnibox": {
      "keyword": "sof"
  },
  "background": {
      "scripts": ["background.js"]
  },
  "permissions": [
      "tabs"
  ],
  "manifest_version": 2
}
```

We have already discussed about the **omnibox** JSON, here is where we will be defining the keyword for this extension, and this is the most important part. The only part which will look different to you should be the below code, which sets the value of **strict__min__version**

```
"applications": {
        "gecko": {
            "strict_min_version": "52.0a1"
        }
    }
```

The meaning of above code is this extension will work only if the browser is having the version greater than *52.0a1* , this is very important. Some of the API's are introduced very late (after 57) or will be upcoming API's. We should make sure the user who is going to install our extension is having minimum the browser version from which our API is supported; so we can avoid the breakage in extension usage. Similiarly we are also having **strict__max__version** which will say the maximum version this extension will work. Generally there was no much need for setting *strict__max__version* but we have to be careful when major version of browser is released.