FREE eBook

LEARNING google-chrome-extension

Free unaffiliated eBook created from **Stack Overflow contributors.**

#google-chrome-extension

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: google-chrome-extension

It is an unofficial and free google-chrome-extension ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official google-chrome-extension.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with google-chrome-extension

Remarks

TODO: Short description of Chrome Extensions

Official documentation

- What are extensions? (documentation hub)
- Getting Started tutorial (basic tutorial)
- Overview
- JavaScript APIs (comprehensive list of `chrome.*` APIs)

Further reading

TODO: Populate with links to important overview topics

Examples

Absolute minimum example

Any Chrome extension starts as an *unpacked extension*: a folder containing the extension's files.

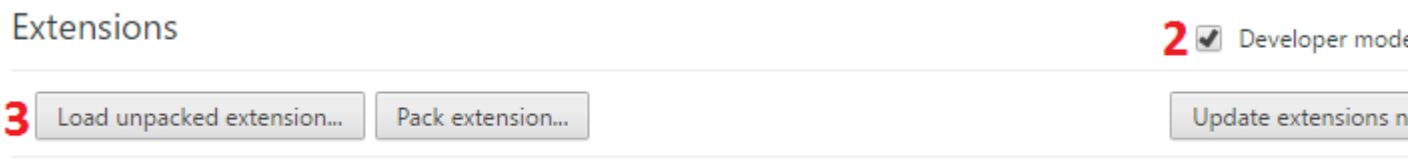
One file it must contain is `manifest.json`, which describes the basic properties of the extension. Many of the properties in that file are optional, but here is an absolute minimum `manifest.json` file:

```
{
  "manifest_version": 2,
  "name": "My Extension",
  "version": "1.0"
}
```

Create a folder (for example, `myExtension`) somewhere, add `manifest.json` as listed above to it.

Then, you need to load the extension in Chrome.

1. Open the `chrome://extensions/` page, accessible though **Menu > More tools > Extensions**.
2. Enable **Developer Mode** with a checkbox in the top right, if it's not enabled already.
3. Click on **Load unpacked extension...** button and select the created `myExtension` folder.



That's it! Your first extension is loaded by Chrome:

Of course, it doesn't do anything yet, so it's a good moment to read an overview of extension architecture to start adding parts you need.

Important: When you do any changes to your extension, do not forget to return to `chrome://extensions/` and press the **Reload** link for your extension after you make changes. In case of content scripts, reload the target page as well.

Background Page

Background pages are implicit pages which contain background scripts. A background script is a single long-running script to manage some task or state. It exists for the lifetime of your extension, and only one instance of it at a time is active.

You can declare it like this in your `manifest.json`:

```
"background": {
    "scripts": ["background.js"]
}
```

A background page will be generated by the extension system that includes each of the files listed in the scripts property.

You have access to all permitted `chrome.*` APIs.

There are two types of background pages: **persistent background pages** which is always open, and **event pages** that is opened and closed as needed.

If you want your background page to be non-persistent, you just have to set the `persistent`-flag to false:

```
"background": {
    "scripts": ["eventPage.js"],
    "persistent": false
}
```

This background script is only active if an event is fired on which you have a listener registered. In general you use a `addListener` for registration.

Example: The app or extension is first installed.

```
chrome.runtime.onInstalled.addListener(function() {
   console.log("The Extension is installed!");
});
```

Content Scripts

A **content script** is extension code that runs alongside a normal page.

They have full access to the web page's DOM (and are, in fact, **the only part of the extension that can access a page's DOM**), but the JavaScript code is isolated, a concept called Isolated World. Each extension has its own content script JavaScript context invisible to others and the page, preventing code conflicts.

Example definition in `manifest.json`:

```
  "content_scripts": [
    {
      "matches": ["http://www.stackoverflow.com/*"],
      "css": ["style.css"],
      "js": ["jquery.js", "myscript.js"]
    }
  ]
```

The attributes have the following meaning:

Attribute	Description
matches	Specifies which pages this content script will be injected into. Follows the Match Pattern format.
css	List of CSS files to be injected into matching pages.
js	List of JS files to be injected into matching pages. **Executed in order listed.**

Content scripts can also be injected on demand using `chrome.tabs.executeScript`, which is called Programmatic Injection.

See also

- Official documentation: Content Scripts
- Stack Overflow documentation: Content Scripts

Options Page

Options pages are used to give the user the possibility to maintain settings for your extension.

Version 2

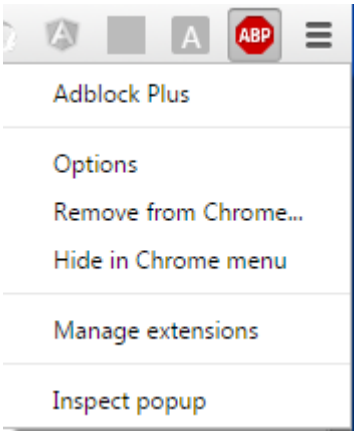
Since Chrome 40 there is the possibility to have the option page as a predefined dialogue at *chrome://extensions*.

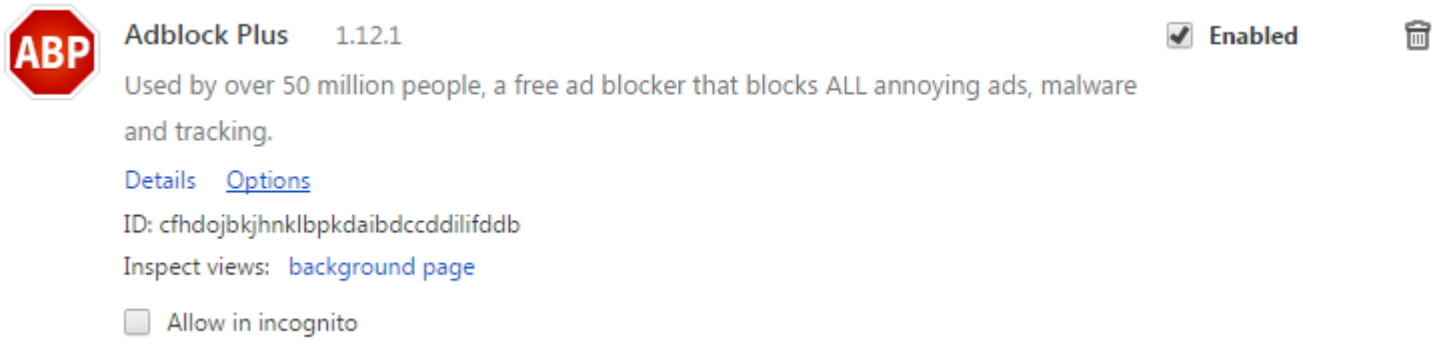
The way to define an option page in the `manifest.json` is like the following:

```
"options_ui": {
    "page": "options.html",
    "chrome_style": true
}
```

This option page will behave as a dialogue, it will open as a popup, where the **options.html** will be displayed. `chrome_style` will apply a Chrome stylesheet for style consistency reasons to your options page.

The options will be automatically exposed via the context menu of the **extension button** or the **chrome://extensions** page.





You can also open the options page programmatically, for example from a popup UI:

```
chrome.runtime.openOptionsPage();
```

Version 1 (deprecated)

Example definition in `manifest.json`:

```
"options_page": "options.html"
```

It is recommended to use Version 2 since the `options_ui` behavior will be soon applied to Version 1 options pages.

Storage

Normally the settings need to persist, so using `chrome.storage` API is recommended. The permissions can be declared like this in the `manifest.json`:

```
"permissions": [
   "storage"
]
```

Official documentation

- Options Page – Version 1
- Options Page – Version 2
- Storage API

Create a new tab

In the extension code you can use any `chrome.*` API if you decalared the required permissions. In addition, some API's works only from background pages, and some API's works only from content scripts.

You can use most of `chrome.tabs` methods declaring any permissions. Now we focus on `chrome.tabs.create`

Note: The new tab will be opened without any `popup` warning.

```
chrome.tabs.create({
       url:"http://stackoverflow.com",
       selected:false  // We open the tab in the background
})
```

You can learn more about tab object, in the official chrome developer

Read Getting started with google-chrome-extension online: https://riptutorial.com/google-chrome-extension/topic/787/getting-started-with-google-chrome-extension

Chapter 2: Background pages

Examples

Declaring background page in the manifest

There are two ways to register a background page in the extension manifest.

1. The `scripts` property

   In the common case, a background page doesn't require any HTML markup. We can register these kinds of background pages using the `scripts` property.

   In this case, a background page will be generated by the extension system that includes each of files listed in the `scripts` property.

   ```
   {
     ...
     "background": {
       "scripts": ["background1.js", "background2.js"],
       "persistent": true
     },
     ...
   }
   ```

2. The `page` property

   In some cases, we may want to specify HTML in background page, we can achieve that using the `page` property.

   ```
   {
     ...
     "background": {
       "page": "background.html",
       "persistent": true
     },
     ...
   }
   ```

`scripts` VS `page`

It's hard to say which one is better. we could use `page` property and have some elements declared in HTML page for future use. We could also dynamically create such elements in the scripts without explicitly declaring HTML page. It all depends on the actual needs.

Read Background pages online: https://riptutorial.com/google-chrome-extension/topic/4066/background-pages

Chapter 3: Content scripts

Remarks

Official documentation

- Content Scripts
- Content Security Policy > Content Scripts

Examples

Declaring content scripts in the manifest

Content scripts can be declared in `manifest.json` to be always injected into pages that match a set of URL patterns.

Minimal example

```
"content_scripts" : [
  {
    "js": ["content.js"],
    "css": ["content.css"]
    "matches": ["http://example.com/*"]
  }
]
```

This manifest entry instructs Chrome to inject a content script `content.js`, along with the CSS file `content.css`, after any navigation to a page matching the match pattern `http://example.com/*`

Both `js` and `css` keys are optional: you can have only one of them or both depending on what you need.

`content_scripts` key is an array, and you can declare several content script definitions:

```
"content_scripts" : [
  {
    "js": ["content.js"],
    "matches": ["http://*.example.com/*"]
  },
  {
    "js": ["something_else.js"],
    "matches": ["http://*.example.org/*"]
  }
]
```

Note that both `js` and `matches` are arrays, even if you only have one entry.

More options are available in the official documentation and other Examples.

Important note

Content scripts declared in the manifest **will only be injected on new navigations after the extension load**. They will not be injected in existing tabs. This also applies to extension reloads while developing, and extension updates after release.

If you need to ensure that currently opened tabs are covered, consider also doing programmatic injection on startup.

Injecting content scripts from an extension page

If, instead of always having a content script injected based on the URL, you want to directly control when a content script is injected, you can use Programmatic Injection.

Minimal example

- JavaScript

```
chrome.tabs.executeScript({file: "content.js"});
```

- CSS

```
chrome.tabs.insertCSS({file: "content.css"});
```

Called from an extension page (e.g. background or popup), and assuming you have permission to inject, this will execute `content.js` or insert `content.css` as a content script in the top frame of the current tab.

Inline code

You can execute inline code instead of a file as a content script:

```
var code = "console.log('This code will execute as a content script');";
chrome.tabs.executeScript({code: code});
```

Choosing the tab

You can provide a tab ID (usually from other `chrome.tabs` methods or messaging) to execute in a tab other than the currently active.

```
chrome.tabs.executeScript({
  tabId: tabId,
  file: "content.js"
```

```
});
```

More options are available in the `chrome.tabs.executeScript()` documentation and in other Examples.

Permissions

Using `chrome.tabs.executeScript()` does not require `"tabs"` permission, but requires host permissions for the page's URL.

Checking for errors

If script injection fails, one can catch it in the optional callback:

```
chrome.tabs.executeScript({file: "content.js"}, function() {
  if(chrome.runtime.lastError) {
    console.error("Script injection failed: " + chrome.runtime.lastError.message);
  }
});
```

Multiple content scripts in the manifest

Same conditions, multiple scripts

If you need to inject multiple files with all other conditions being the same, for example to include a library, you can list all of them in the `"js"` array:

```
"content_scripts" : [
  {
    "js": ["library.js", "content.js"],
    "matches": ["http://*.example.com/*"]
  }
]
```

Order matters: `library.js` will be executed before `content.js`.

Same scripts, multiple sites

If you need to inject the same files into multiple sites, you can provide multiple match patterns:

```
"matches": ["http://example.com/*", "http://example.org/*"]
```

If you need to inject in basically every page, you can use broad match patterns such as `"*://*/*"` (matches every HTTP(S) page) or `"<all_urls>"` (matches every supported page).

Different scripts or different sites

`"content_scripts"` section is an array as well, so one can define more than one content script block:

```
"content_scripts" : [
  {
    "js": ["content.js"],
    "matches": ["http://*.example.com/*"]
  },
  {
    "js": ["something_else.js"],
    "matches": ["http://*.example.org/*"]
  }
]
```

Read Content scripts online: https://riptutorial.com/google-chrome-extension/topic/2850/content-scripts

Chapter 4: Debugging Chrome Extensions

Examples

Using the Developer tools to debug your extension

A chrome extension is seperated into a maximum of 4 parts:

- the background page
- the popup page
- one or more content scripts
- the options page

Each part, since they are innately separate, require individual debugging.

Keep in mind that these pages are separate, meaning that variables are not directly shared between them and that a `console.log()` in one of these pages will not be visible in any other part's logs.

Using the chrome devtools:

Chrome extensions are debugged similar as to other webapps and webpages. Debugging is most often done with the use of chrome's devtools inspector opened by using the keyboard shortcut for windows and macs respectively: `ctrl`+`shift`+`i` and `cmd`+`shift`+`i` or by right clicking on the page and selecting inspect.

From the inspector a developer can check html elements and how css affects them, or use the console to inspect the values of javascript variables and read the outputs from any `console.log()`s the developer(s) set up.

More information about the usage of the inspector can be found at Chrome Devtools.

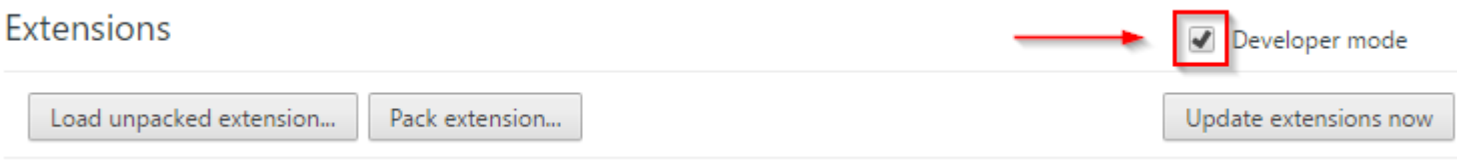
Inspecting the popup, options page, and other pages accessible using chrome://.....yourExtensionId.../:

The *popup page* and *options page* can each be accessed simply by inspecting them when they are open.

Additional html pages that are part of the extension, but are neither the popup nor the options page are also debugged the same way.

Inspecting the background page:

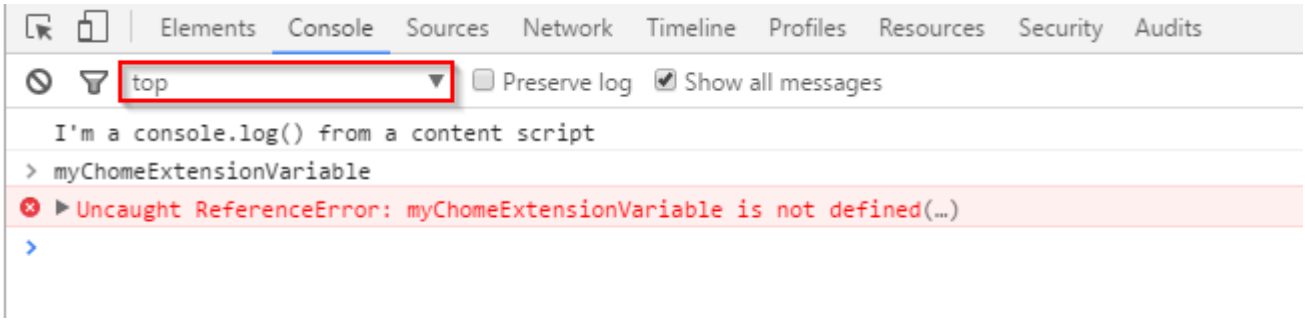
To access your *background page* you must first navigate to the chrome extension page at chrome://extensions/. Make sure the *'Developer mode'* checkmark is enabled.



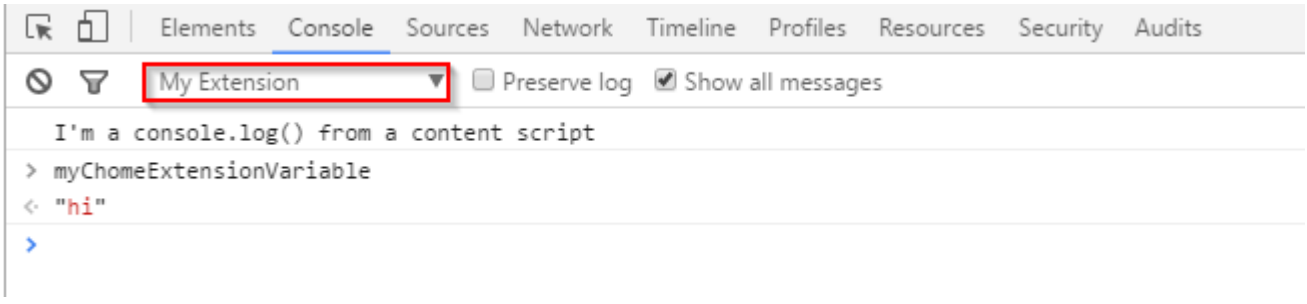
Then click on your background script beside *"Inspect views"* to inspect your background page.

Inspecting content scripts:

Content scripts run along-side the websites they were inserted into. You can inspect the content script by first inspecting the website where the content script is inserted. In the console you will be able to view any `console.log()`s outputted by your extension, but you will not be able to change or inspect the content script's variables.



To fix this you must click on the drop down that is usually set to `'top'` and select your extension from the list of extensions.



From there you will have access to the variables within your extension.

Read Debugging Chrome Extensions online: https://riptutorial.com/google-chrome-extension/topic/5730/debugging-chrome-extensions

Chapter 5: Developer Tool Integration

Examples

Programmatic Breakpoint Hinting

Add the debugger statement in your content script

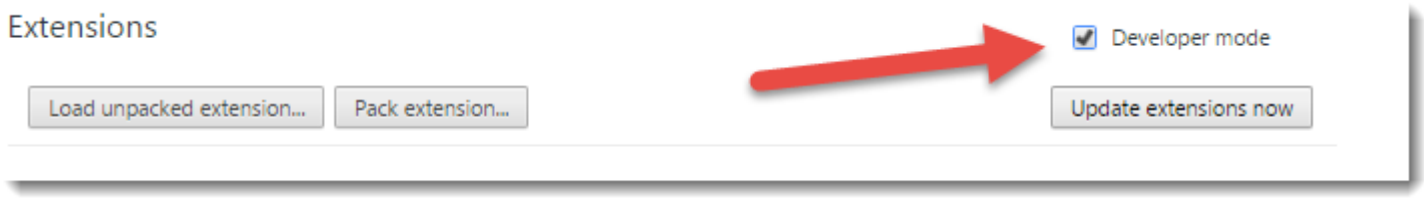
```
var foo = 1;
debugger;

foo = 2;
```

Open the Developer Tool on the web page where your content script is injected to see the code execution pause at those lines.

Debugging the background page/script

The background script is like any other JavaScript code. You can debug it using same tools you debug other JavaScript code in Chrome.

To open the Chrome Developer Tools, go to `chrome://extensions`, and turn on **Developer mode**:



Now you can debug any extension that have a background page or script. Just scroll to the extension you want to debug and click on the **background page** link to inspect it.

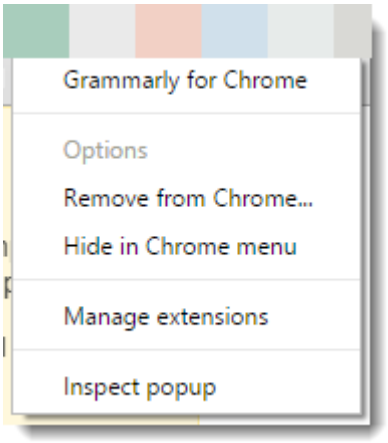
Tip: To reload the extension, you can press `F5` inside the developer tools window. You can put breakpoints in the initialization code before reloading.

Tip: Right-clicking the extension action button and selecting "Manage extensions" will open `chrome://extensions` page scrolled to that extension.

Debugging the popup window

You have 2 ways to debug the popup window. Both ways are by using the Chrome DevTools.

Option 1: Right click the extension's action button, and choose **Inspect popup**



Option 2: Open the popup window, directly in your browser as a tab.

For example, if you extension id is `abcdefghijkmnop`, and your popup html file is `popup.html`. Go to the address and navigate to:

```
chrome-extension://abcdefghijklmnop/popup.html
```

Now you see the poup in regular tab. And you can press `F12` to open the developer tools.

Read Developer Tool Integration online: https://riptutorial.com/google-chrome-extension/topic/5938/developer-tool-integration

Chapter 6: manifest.json

Remarks

Official documentation

Manifest File Format

Format

Manifest file is written in JSON (JavaScript Object Notation) format.

This format differs from more loose rules of writing object literals in JavaScript code. Among important differences:

- Every key name and string literal **must be in double quotes**.
  - Correct: `"key": "value"`
  - Wrong: `key: "value", 'key': 'value'`
- **No comments** are allowed by the format.
  - Wrong: `"key": "value" // This controls feature foo`
- Strict comma rules: **items separated by commas, no dangling commas**.
  - Correct:

```
{
  "foo": "bar",
  "baz": "qux"
}
```

  - Wrong (comma missing):

```
{
  "foo": "bar"
  "baz": "qux"
}
```

  - Wrong (dangling comma):

```
{
  "foo": "bar",
  "baz": "qux",
```

```
}
```

Examples

Absolute minimum manifest.json

`manifest.json` gives information about the extension, such as the most important files and the capabilities that the extension might use. Among the supported manifest fields for extensions, the following **three** are required.

```
{
    "manifest_version": 2,
    "name": "My Extension",
    "version": "1.0"
}
```

Obtaining manifest from extension code

`chrome.runtime.getManifest()` returns the extension's manifest in a form of a parsed object.

This method works both on content scripts and all extension pages, it requires no permissions,

Example, obtaining the extension's version string:

```
var version = chrome.runtime.getManifest().version;
```

Read manifest.json online: https://riptutorial.com/google-chrome-extension/topic/948/manifest-json

Chapter 7: Message Passing

Remarks

Official documentation

- Message Passing
- Native Messaging
- `chrome.runtime` API (most messaging functions and all messaging events)

Examples

Send a response asynchronously

In attempt to send a response asynchronously from `chrome.runtime.onMessage` callback we might try this **wrong code**:

```
chrome.runtime.onMessage.addListener(function(request, sender, sendResponse) {
    $.ajax({
        url: 'https://www.google.com',
        method: 'GET',
        success: function(data) {
            // data won't be sent
            sendResponse(data);
        },
    });
});
```

However, we would find that `data` is never sent. This happens because we have put `sendResponse` inside an asynchronous ajax call, when the `success` method is executed, the message channel has been closed.

The solution would be simple, as long as we explicitly `return true;` at the end of the callback, which indicates we wish to send a response asynchronously, so the message channel will be kept open to the other end (caller) until `sendResponse` is executed.

```
chrome.runtime.onMessage.addListener(function(request, sender, sendResponse) {
    $.ajax({
        url: 'https://www.google.com',
        method: 'GET',
        success: function(data) {
            // data would be sent successfully
            sendResponse(data);
        },
    });

    return true; // keeps the message channel open until `sendResponse` is executed
});
```

Of course, it applies to an explicit `return` from the onMessage callback as well:

```
chrome.runtime.onMessage.addListener(function(request, sender, sendResponse) {
    if (request.action == 'get') {
        $.ajax({
            url: 'https://www.google.com',
            method: 'GET',
            success: function(data) {
                // data would be sent successfully
                sendResponse(data);
            },
        });

        return true; // keeps the message channel open until `sendResponse` is executed
    }

    // do something synchronous, use sendResponse

    // normal exit closes the message channel
});
```

Read Message Passing online: https://riptutorial.com/google-chrome-extension/topic/2185/message-passing

Chapter 8: Porting to/from Firefox

Remarks

> If you're using a *Firefox* version before 48, you'll also need an additional key in `manifest.json` called applications:

```
"applications": {
  "gecko": {
    "id": "borderify@example.com",
    "strict_min_version": "42.0",
    "strict_max_version": "50.*",
    "update_url": "https://example.com/updates.json"
  }
}
```

applications

Note:

Extension Signing:

> With the release of Firefox 48, extension signing can no longer be disabled in the release and beta channel builds by using a preference. As outlined when extension signing was announced, we are publishing specialized builds that support this preference so developers can continue to test against the code that beta and release builds are generated from.

Status of `WebExtensions`:

> WebExtensions are currently in an experimental alpha state. From Firefox 46, you can publish WebExtensions to Firefox users, just like any other add-on. We're aiming for a first stable release in Firefox 48.

UPD: *Firefox* 48 released 02.08.2016.

Links:

API support status - The list of APIs and their status.

Chrome incompatibilities

WebExtensions - JavaScript APIs, keys of manifest.json, tutorials, etc.

Examples

Porting through WebExtensions

Before talking about porting *Firefox* extensions from/to, one should know what `WebExtensions` is.

`WebExtensions` - is a platform that represents an API for creating *Firefox* extensions.

It uses the same architecture of extension as *Chromium*, as a result, this API is compatible in many ways with API in *Google Chrome* and *Opera* (Opera which based on Chromium). In many cases, extensions developed for these browsers will work in *Firefox* with a few changes or even without them at all.

MDN recommends to use `WebExtension` for new extensions:

> In the future, WebExtensions will be the recommended way to develop Firefox add-ons, and other systems will be deprecated.

In view of the foregoing, if you want to port extensions to *Firefox*, you have to know, how the extension was written.

Extensions for *Firefox* can be based on `WebExtension`, `Add-on SDK` or `XUL`.

Compatible extensions based on WebExtension

When using `WebExtension`, one has to look through the list of incompatibilities, because some functions are supported fully or partially, that is in other words, one should check one's `manifest.json`.

It also enables to use the same namespace:

> At this time, all APIs are accessible through the chrome.* namespace. When we begin to add our own APIs, we expect to add them to the browser.* namespace. Developers will be able to use feature detection to determine if an API is available in browser.*.

A simple extension that can work in Firefox and Google Chrome

`manifest.json`:

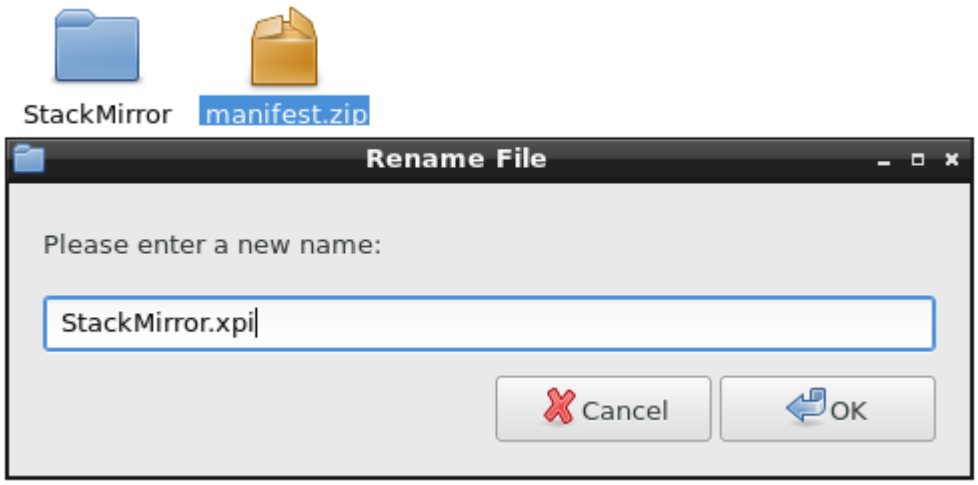
```
{
  "manifest_version": 2,

  "name": "StackMirror",

  "version": "1.0",
```

```
  "description": "Mirror reflection of StackOverflow sites",

  "icons": {
    "48": "icon/myIcon-48.png"
  },

  "page_action": {
    "default_icon": "icon/myIcon-48.png"
  },

  "background": {
    "scripts"   : ["js/background/script.js"],
    "persistent": false
  },

  "permissions": ["tabs", "*://*.stackoverflow.com/*"]
}
```

`background` script:

```
function startScript(tabId, changeInfo, tab) {

    if (tab.url.indexOf("stackoverflow.com") > -1) {

        chrome.tabs.executeScript(tabId,

            {code: 'document.body.style.transform = "scaleX(-1)";'}, function () {

            if (!chrome.runtime.lastError) {

                chrome.pageAction.show(tabId);
            }
        });
    }
}

chrome.tabs.onUpdated.addListener(startScript);
```

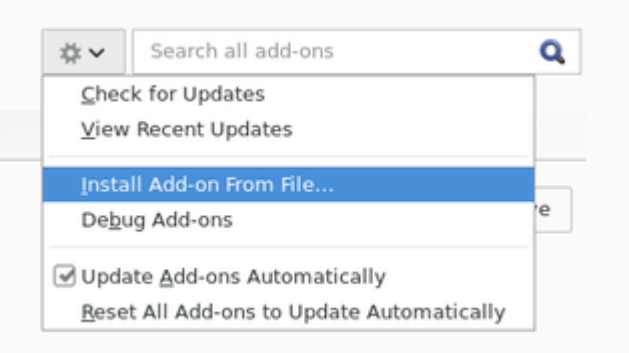
Pack project as standard `zip` file, but with `.xpi` extensions.



Then,you have to load the extension in *Firefox*.

Open the `about:addons` page, accessible through **Menu > Add-ons**.

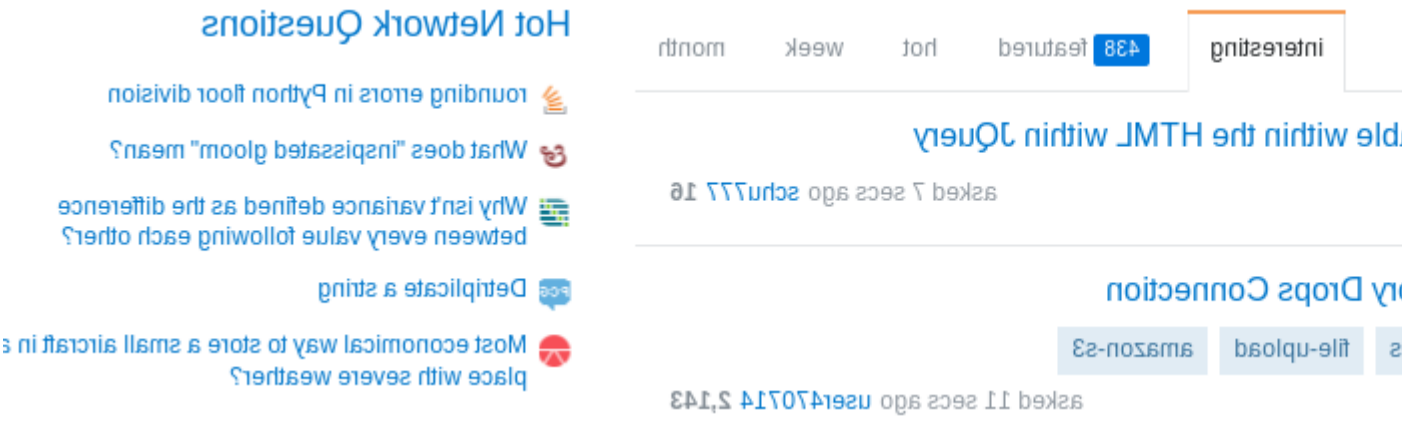
Click on **Tools for all add-ons** button.



When the extension is loaded the page `about:addons` will look like this:

Directions on loading the extension in Google Chrome is in other topic - Getting started with Chrome Extensions.

The result of extension operation will be same in both browsers (*Firefox*/*Google Chrome*):



If the current add-on is based on Add-on SDK or XUL

When extension being ported is based on `Add-on SDK` one has to look through the comparison table for `Add-on SDK` **=>** `WebExtensions`, because these technologies have similar features, but differ in implementation. Each section of table describes the equivalent of `Add-on SDK` for `WebExtension`.

Comparison with the Add-on SDK

A similar approach and for XUL extensions.

Comparison with XUL/XPCOM extensions

Read Porting to/from Firefox online: https://riptutorial.com/google-chrome-extension/topic/5731/porting-to-from-firefox

Credits

S. No	Chapters	Contributors
1	Getting started with google-chrome-extension	Aminadav, Community, Deliaz, Haibara Ai, ScientiaEtVeritas, Xan
2	Background pages	Haibara Ai, Noam Hacker
3	Content scripts	Haibara Ai, Xan
4	Debugging Chrome Extensions	Marc Guiselin
5	Developer Tool Integration	Aminadav, Paul Sweatte, Xan
6	manifest.json	Haibara Ai, Xan
7	Message Passing	Haibara Ai, wOxxOm, Xan
8	Porting to/from Firefox	Deliaz, UserName