Design User Interface

The extension user interface should be purposeful and minimal. Just like extensions themselves, the UI should customize or enhance the browsing experience without distracting from it.

This guide explores required and optional user interface features. Use it to understand how and when to implement different UI elements within an extension.

Activate the extension on all pages

Use a **browser_action** when an extension's features are functional in most situations.

Register browser action

The "browser_action" field is registered in the manifest.

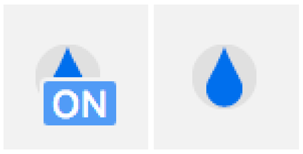
```
{
  "name": "My Awesome browser_action Extension",
  ...
  "browser_action": {
    ...
  }
  ...
}
```

Declaring "browser_action" keeps the icon colorized, indicating the extension is available to users.

Add a badge

Badges display a colored banner with up to four characters on top of the browser icon. They can only be used by extensions that declare "browser_action" in their manifest.

Use badges to indicate the state of the extension. The **Drink Water Event** sample displays a badge with “ON” to show the user they successfully set an alarm and displays nothing when the extension is idle.



Set the text of the badge by calling **chrome.browserAction.setBadgeText** and the banner color by calling **chrome.browserAction.setBadgeBackgroundColor** .

```
chrome.browserAction.setBadgeText({text: 'ON'});
chrome.browserAction.setBadgeBackgroundColor({color: '#4688F1'});
```

Activate the extension on select pages

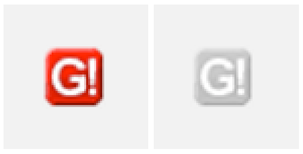
Use **page_action** when an extension's features are only available under defined circumstances.

Declare Page Action

The "page_action" field is registered in the manifest.

```
{
  "name": "My Awesome page_action Extension",
  ...
  "page_action": {
      ...
    }
    ...
  }
```

Declaring "page_action" will colorize the icon only when the extension is available to users, otherwise it will be displayed in greyscale.



Define rules for activating the extension

Define rules for when the extension is usable by calling **chrome.declarativeContent** under the **runtime.onInstalled** listener in a **background script**. The **Page action by URL** sample extension sets a condition that the url must include a 'g'. If the condition is met, the extension calls **declarativeContent.ShowPageAction()**.

如果url包含一个字母g，那么激活这个扩展

```
  chrome.runtime.onInstalled.addListener(function() {
    // Replace all rules ...
    chrome.declarativeContent.onPageChanged.removeRules(undefined, function() {
      // With a new rule ...
      chrome.declarativeContent.onPageChanged.addRules([
        {
          // That fires when a page's URL contains a 'g' ...
          conditions: [
            new chrome.declarativeContent.PageStateMatcher({
              pageUrl: { urlContains: 'g' },
            })
          ],
          // And shows the extension's page action.
          actions: [ new chrome.declarativeContent.ShowPageAction() ]
        }
      ]);
    });
  });
```

Enable or disable the extension

Extensions using "page_action" can activate and disable dynamically by calling **pageAction.show** and **pageAction.hide**.

The **Mappy** sample extension scans a web page for an address and shows its location on a static map in the **popup**. Because the extension is dependent on page content, it cannot declare rules to predict which pages will be relevant. Instead, if an address is found on a page it calls `pageAction.show` to colorize the icon and signal the extension is usable on that tab.

```
chrome.runtime.onMessage.addListener(function(req, sender) {
  chrome.storage.local.set({'address': req.address})
  chrome.pageAction.show(sender.tab.id);
  chrome.pageAction.setTitle({tabId: sender.tab.id, title: req.address});
});
```

Provide the extension icons

Extensions require at least one icon to represent it. Provide icons in PNG format form the best visual results, although any format supported by WebKit including BMP, GIF, ICO, and JPEG is accepted.

Designate toolbar icons

Icons specific to the toolbar are registered in the `"default_icon"` field under `browser_action` or `page_action` in the manifest. Including multiple sizes is encouraged to scale for the 16-dip space. At minimum, 16x16 and 32x32 sizes are recommended.

```
{
  "name": "My Awesome page_action Extension",
  ...
  "page_action": {
    "default_icon": {
      "16": "extension_toolbar_icon16.png",
      "32": "extension_toolbar_icon32.png"
    }
  }
  ...
}
```

All icons should be square or they may be distorted. If no icons are supplied, Chrome will add a generic one to the toolbar.

Create and register additional icons

Include additional icons in the following sizes for uses outside of the toolbar.

Icon Size	Icon Use
16x16	favicon on the extension's pages

32x32	Windows computers often require this size. Providing this option will prevent size distortion from shrinking the 48x48 option.
48x48	displays on the extensions management page
128x128	displays on installation and in the Chrome Webstore

Register icons in the manifest under the `"icons"` field.

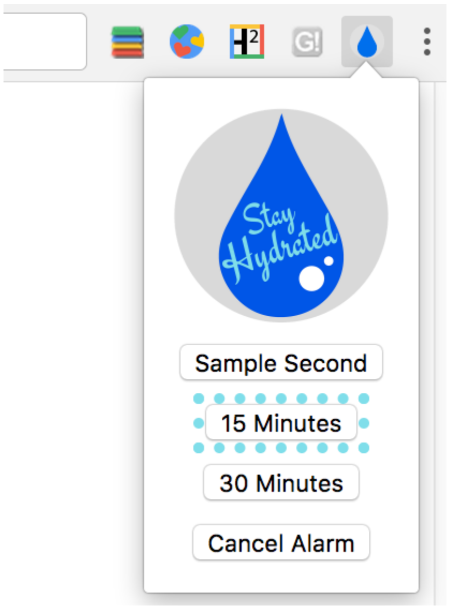
```
{
  "name": "My Awesome Extension",
  ...
  "icons": {
    "16": "extension_icon16.png",
    "32": "extension_icon32.png",
    "48": "extension_icon48.png",
    "128": "extension_icon128.png"
  }
  ...
}
```

Additional UI Features

Popup

A popup is an HTML file that is displayed in a special window when the user clicks the toolbar icon. A popup works very similarly to a web page; it can contain links to stylesheets and script tags, but does not allow inline JavaScript.

The **Drink Water Event** example popup displays available timer options. Users set an alarm by clicking one of the provided buttons.



```
<html>
  <head>
    <title>Water Popup</title>
  </head>
  <body>
      <img src='./stay_hydrated.png' id='hydrateImage'>
      <button id='sampleSecond' value='0.1'>Sample Second</button>
      <button id='15min' value='15'>15 Minutes</button>
      <button id='30min' value='30'>30 Minutes</button>
      <button id='cancelAlarm'>Cancel Alarm</button>
   <script src="popup.js"></script>
  </body>
</html>
```

The popup can be registered in the manifest, under browser action or page action.

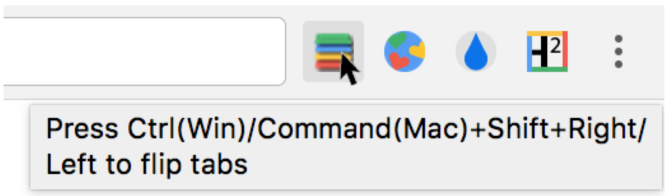
```
{
  "name": "Drink Water Event",
  ...
  "browser_action": {
    "default_popup": "popup.html"
  }
  ...
}
```

Popups can also be set dynamically by calling `browserAction.setPopup` or `pageAction.setPopup`.

```
  chrome.storage.local.get('signed_in', function(data) {
    if (data.signed_in) {
      chrome.browserAction.setPopup({popup: 'popup.html'});
    } else {
      chrome.browserAction.setPopup({popup: 'popup_sign_in.html'});
    }
  });
```

Tooltip

Use a tooltip to give short descriptions or instructions to users when hovering over the browser icon.


Press Ctrl(Win)/Command(Mac)+Shift+Right/
Left to flip tabs

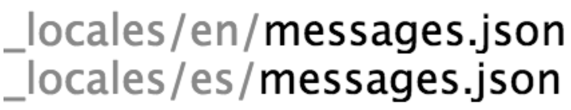
Tooltips are registered in the `"default_title"` field **`browser_action`** or **`page_action`** in the manifest.

```
  {
  "name": "Tab Flipper",
   ...
    "browser_action": {
      "default_title": "Press Ctrl(Win)/Command(Mac)+Shift+Right/Left to flip tabs"
    }
  ...
}
```

Tooltips can also be set or updated by calling **`browserAction.setTitle`** and **`pageAction.setTitle`**.

```
  chrome.browserAction.onClicked.addListener(function(tab) {
    chrome.browserAction.setTitle({tabId: tab.id, title: "You are on tab:" +
tab.id});
  });
```

Specialized locale strings are implemented with **Internationalization**. Create directories to house language specific messages within a folder called `_locales`. The following image shows a file path for an extension that supports English and Spanish locales.


_locales/en/messages.json
_locales/es/messages.json

Format messages inside of each language's `messages.json`.

```
{
  "__MSG_tooltip__": {
      "message": "Hello!",
      "description": "Tooltip Greeting."
  }
}
```

```
{
  "__MSG_tooltip__": {
      "message": "Hola!",
      "description": "Tooltip Greeting."
  }
}
```

Include the name of the message in the tooltip field instead of the message to enable localization.

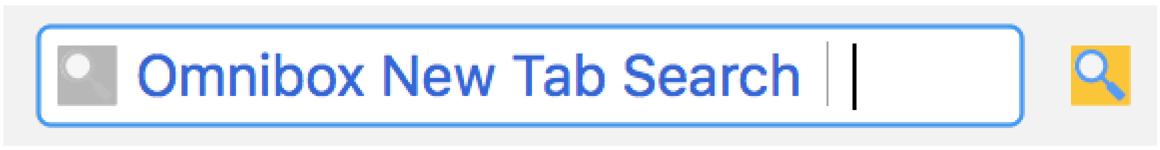
```
 {
 "name": "Tab Flipper",
  ...
  "browser_action": {
    "default_title": "__MSG_tooltip__"
  }
 ...
}
```

Omnibox

Users can invoke extension functionality through the **omnibox**. Include the `"omnibox"` field in the manifest and designate a keyword. The **Omnibox New Tab Search** sample extension uses "nt" as the keyword.

```
{
  "name": "Omnibox New Tab Search",\
  ...
  "omnibox": { "keyword" : "nt" },
  "default_icon": {
    "16": "newtab_search16.png",
    "32": "newtab_search32.png"
  }
  ...
}
```

When the user types "nt" into the omnibox, it activates the extension. To signal this to the user, it greyscales the provided 16x16 icon and includes it in the omnibox next to the extension name.



The extension listens to the `omnibox.onInputEntered` event. After it's triggered, the extension opens a new tab containing a Google search for the user's entry.

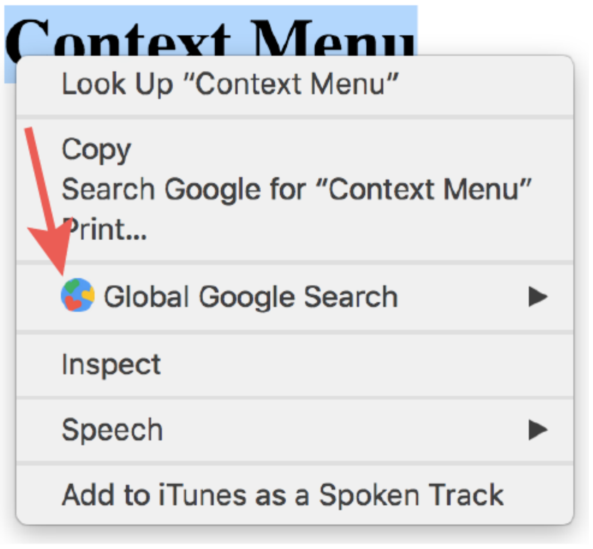
```
chrome.omnibox.onInputEntered.addListener(function(text) {
  // Encode user input for special characters , / ? : @ & = + $ #
  var newURL = 'https://www.google.com/search?q=' + encodeURIComponent(text);
  chrome.tabs.create({ url: newURL });
});
```

Context Menu

Add new **context menu** options by granting the `"contextMenus"` permission in the manifest.

```
  {
    "name": "Global Google Search",
    ...
    "permissions": ["contextMenus", "storage"],
    "icons": {
      "16": "globalGoogle16.png",
      "48": "globalGoogle48.png",
      "128": "globalGoogle128.png"
   }
   ...
  }
```

The 16x16 icon is displayed next to the new menu entry.

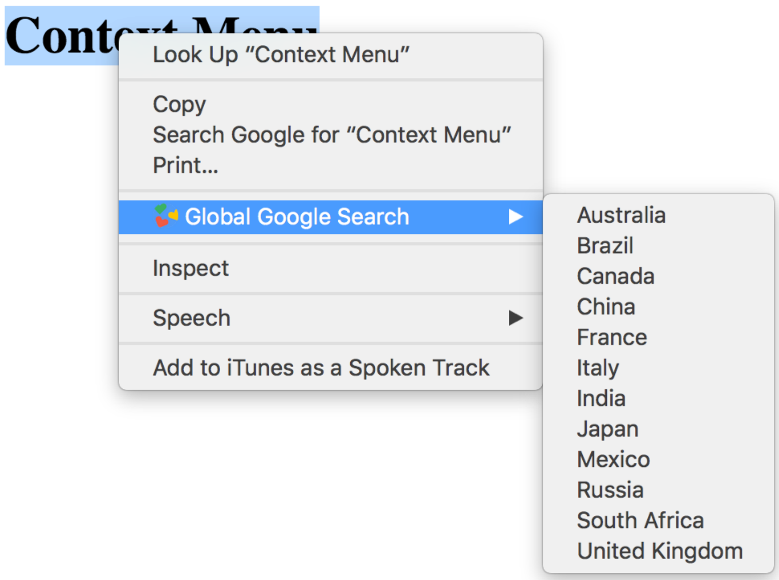


Create a context menu by calling `contextMenus.create` in the **background script**. This should be done under the `runtime.onInstalled` listener event.

```
chrome.runtime.onInstalled.addListener(function() {
  for (let key of Object.keys(kLocales)) {
    chrome.contextMenus.create({
      id: key,
      title: kLocales[key],
      type: 'normal',
      contexts: ['selection'],
    });
  }
});
```

```
const kLocales = {
  'com.au': 'Australia',
  'com.br': 'Brazil',
  'ca': 'Canada',
  'cn': 'China',
  'fr': 'France',
  'it': 'Italy',
  'co.in': 'India',
  'co.jp': 'Japan',
  'com.ms': 'Mexico',
  'ru': 'Russia',
  'co.za': 'South Africa',
  'co.uk': 'United Kingdom'
};
```

The Global Google Search context menu example creates multiple options from the list in **locales.js** . When an extension contains more than one context menu, Google Chrome automatically collapses them into a single parent menu.



Commands

Extensions can define specific **commands** and bind them to a key combination. Register one or more commands in the manifest under the `"commands"` field.

```
{
  "name": "Tab Flipper",
  ...
  "commands": {
    "flip-tabs-forward": {
      "suggested_key": {
        "default": "Ctrl+Shift+Right",
        "mac": "Command+Shift+Right"
      },
      "description": "Flip tabs forward"
    },
    "flip-tabs-backwards": {
      "suggested_key": {
        "default": "Ctrl+Shift+Left",
        "mac": "Command+Shift+Left"
      },
      "description": "Flip tabs backwards"
    }
  }
  ...
}
```

Commands can be used to provide new or alternative browser shortcuts. The **Tab Flipper** sample extension listens to the **`commands.onCommand`** event in the **background script** and defines functionality for each registered combination.

```
chrome.commands.onCommand.addListener(function(command) {
  chrome.tabs.query({currentWindow: true}, function(tabs) {
    // Sort tabs according to their index in the window.
    tabs.sort((a, b) => { return a.index < b.index; });
    let activeIndex = tabs.findIndex((tab) => { return tab.active; });
    let lastTab = tabs.length - 1;
    let newIndex = -1;
    if (command === 'flip-tabs-forward')
      newIndex = activeIndex === 0 ? lastTab : activeIndex - 1;
    else  // 'flip-tabs-backwards'
      newIndex = activeIndex === lastTab ? 0 : activeIndex + 1;
    chrome.tabs.update(tabs[newIndex].id, {active: true, highlighted: true});
  });
});
```

Commands can also create a key binding that works specially with its extension. The **Hello Extensions** example gives a command to open the popup.

```
{
  "name": "Hello Extensions",
  "description" : "Base Level Extension",
  "version": "1.0",
  "browser_action": {
    "default_popup": "hello.html",
    "default_icon": "hello_extensions.png"
  },
  "manifest_version": 2,
  "commands": {
    "_execute_browser_action": {
      "suggested_key": {
        "default": "Ctrl+Shift+F",
        "mac": "MacCtrl+Shift+F"
      },
      "description": "Opens hello.html"
    }
  }
}
```

Because the extension defines a **`broswer_action`** it can specify `"execute_browser_action"` in the commands to open the popup file without including a **background script**. If using **`page_action`**, it can be replaced with "execute_page_action". Both browser and extension commands can be used in the same extension.

Override Pages

An extension can **override** and replace the History, New Tab, or Bookmarks web page with a custom HTML file. Like a **popup**, it can include specialized logic and style, but does not allow inline JavaScript. A single extension is limited to overriding only one of the three possible pages.

Register an override page in the manifest under the "chrome_url_overrides" field.

```
{
  "name": "Awesome Override Extension",
  ...

  "chrome_url_overrides" : {
    "newtab": "override_page.html"
  },
  ...
}
```

The "newtab" field should be replaed with "bookmarks" or "history" when overriding those pages.

```
<html>
 <head>
  <title>New Tab</title>
 </head>
 <body>
    <h1>Hello World</h1>
  <script src="logic.js"></script>
 </body>
</html>
```