Overview

Extensions are zipped bundles of HTML, CSS, JavaScript, images, and other files used in the web platform, that customize the Google Chrome browsing experience. Extensions are built using web technology and can use the same APIs the browser provides to the open web.

Extensions have a wide range of functional possibilities. They can modify web content users see and interact with or extend and change the behavior of the browser itself.

Consider extensions the gateway to making the Chrome browser the most personalized browser.

Extension Files

Extensions vary in types of files and amount of directories, but they are all required to have a **manifest**. Some basic, but useful, extensions may consist of just the manifest and its toolbar icon.

The manifest file, titled `manifest.json`, gives the browser information about the extension, such as the most important files and the capabilities the extension might use.

```
{
  "name": "My Extension",
  "version": "2.1",
  "description": "Gets information from Google.",
  "icons": {
    "128": "icon_16.png",
    "128": "icon_32.png",
    "128": "icon_48.png",
    "128": "icon_128.png"
  },
  "background": {
    "persistent": false,
    "scripts": ["background_script.js"]
  },
  "permissions": ["https://*.google.com/", "activeTab"],
  "browser_action": {
    "default_icon": "icon_16.png",
    "default_popup": "popup.html"
  }
}
```

Extensions must have an icon that sits in the browser toolbar. Toolbar icons allow easy access and keep users aware of which extensions are installed. Most users will interact with an extension that uses a **popup** by clicking on the icon.

This **Google Mail Checker extension** uses a **browser action**.

This **Mappy extension** uses a **page action** and **content script**.

Referring to files

An extension's files can be referred to by using a relative URL, just as files in an ordinary HTML page.

```
<img src="images/my_image.png">
```

Additionally, each file can also be accessed using an absolute URL.

chrome-extension://*<extensionID>***/***<pathToFile>*

In the absolute URL, the *<extensionID>* is a unique identifier that the extension system generates for each extension. The IDs for all loaded extensions can be viewed by going to the URL **chrome://extensions**. The *<pathToFile>* is the location of the file under the extension's top folder; it matches the relative URL.

While working on an unpacked extension the extension ID can change. Specifically, the ID of an unpacked extension will change if the extension is loaded from a different directory; the ID will change again when the extension is packaged. If an extension's code relies on an absolute URL, it can use the `chrome.runtime.getURL()` method to avoid hardcoding the ID during development.

Architecture

An extension's architecture will depend on its functionality, but many robust extensions will include multiple components:

- **Manifest**
- **Background Script**
- **UI Elements**
- **Content Script**
- **Options Page**

Background Script

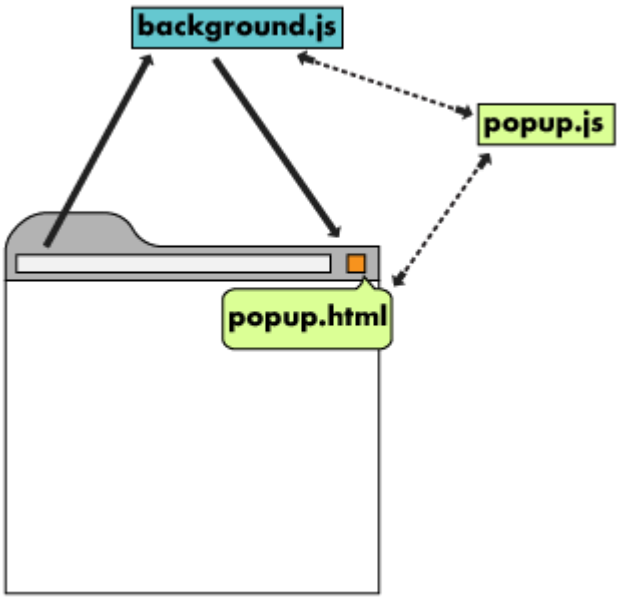
The **background script** is the extension's event handler; it contains listeners for browser events that are important to the extension. It lies dormant until an event is fired then performs the instructed logic. An effective background script is only loaded when it is needed and unloaded when it goes idle.

UI Elements

An **extension's user interface** should be purposeful and minimal. The UI should customize or enhance the browsing experience without distracting from it. Most extensions have a **browser action** or **page action**, but can contain other forms of UI, such as **context menus**, use of the **omnibox**, or creation of a **keyboard shortcut**.

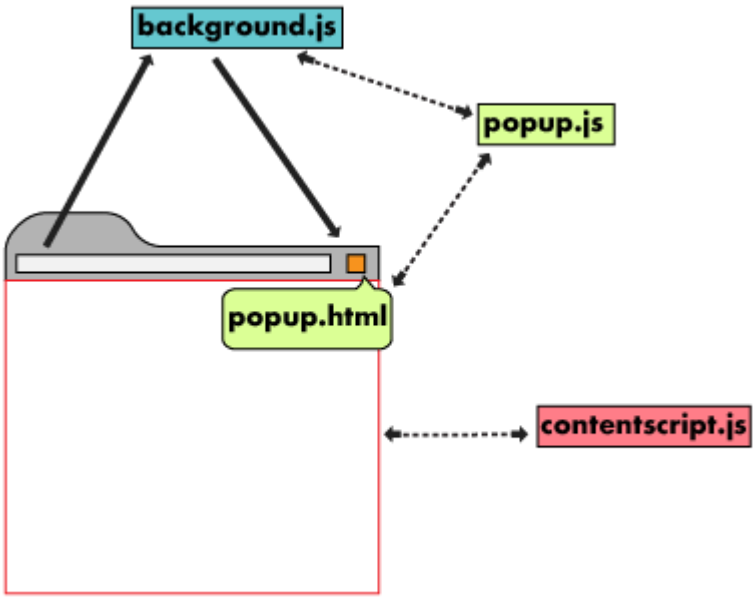
Extension UI pages, such as a **popup**, can contain ordinary HTML pages with JavaScript logic. Extensions can also call **tabs.create** or `window.open()` to display additional HTML files present in the extension.

An extension using a page action and a popup can use the **declarative content** API to set rules in the background script for when the popup is available to users. When the conditions are met, the background script communicates with the popup to make it's icon clickable to users.

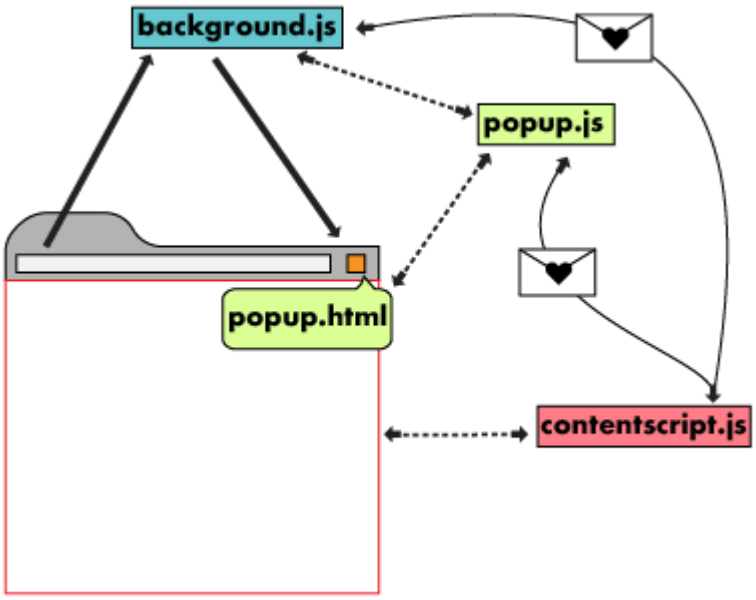


Content scripts

Extensions that read or write to web pages utilize a **content script**. The content script contains JavaScript that executes in the contexts of a page that has been loaded into the browser. Content scripts read and modify the DOM of web pages the browser visits.



Content scripts can communicate with their parent extension by exchanging **messages** and storing values using the **storage** API.



Options Page

Just as extensions allow users to customize the Chrome browser, the **options page** enables customization of the extension. Options can be used to enable features and allow users to choose what functionality is relevant to their needs.

Using Chrome APIs

In addition to having access to the same APIs as web pages, extensions can also use **extension-specific APIs** that create tight integration with the browser. Extensions and webpages can both access the standard `window.open()` method to open a URL, but extensions can specify which window that URL should be displayed in by using the Chrome API **tabs.create** method instead.

Asynchronous vs. synchronous methods

异步方法的解释，非常经典！！

Most Chrome API methods are asynchronous: they return immediately without waiting for the operation to finish. If an extension needs to know the outcome of an asynchronous operation it can pass a callback function into the method. The callback is executed later, potentially much later, after the method returns.

If the extension needed to navigate the user's currently selected tab to a new URL, it would need to get the current tab's ID and then update that tab's address to the new URL.

If the **tabs.query** method were synchronous, it may look something like below.

```
//THIS CODE DOESN'T WORK
var tab = chrome.tabs.query({'active': true}); //WRONG!!!
chrome.tabs.update(tab.id, {url:newUrl});
someOtherFunction();
```

This approach will fail because `query()` is asynchronous. It returns without waiting for the work to complete, and does not return a value. A method is asynchronous when the callback parameter is available in its signature.

```
// Signature for an asynchronous method
chrome.tabs.query(object queryInfo, function callback)
```

To correctly query a tab and update its URL the extension must use the callback parameter.

```
//THIS CODE WORKS
chrome.tabs.query({'active': true}, function(tabs) {
  chrome.tabs.update(tabs[0].id, {url: newUrl});
});
someOtherFunction();
```

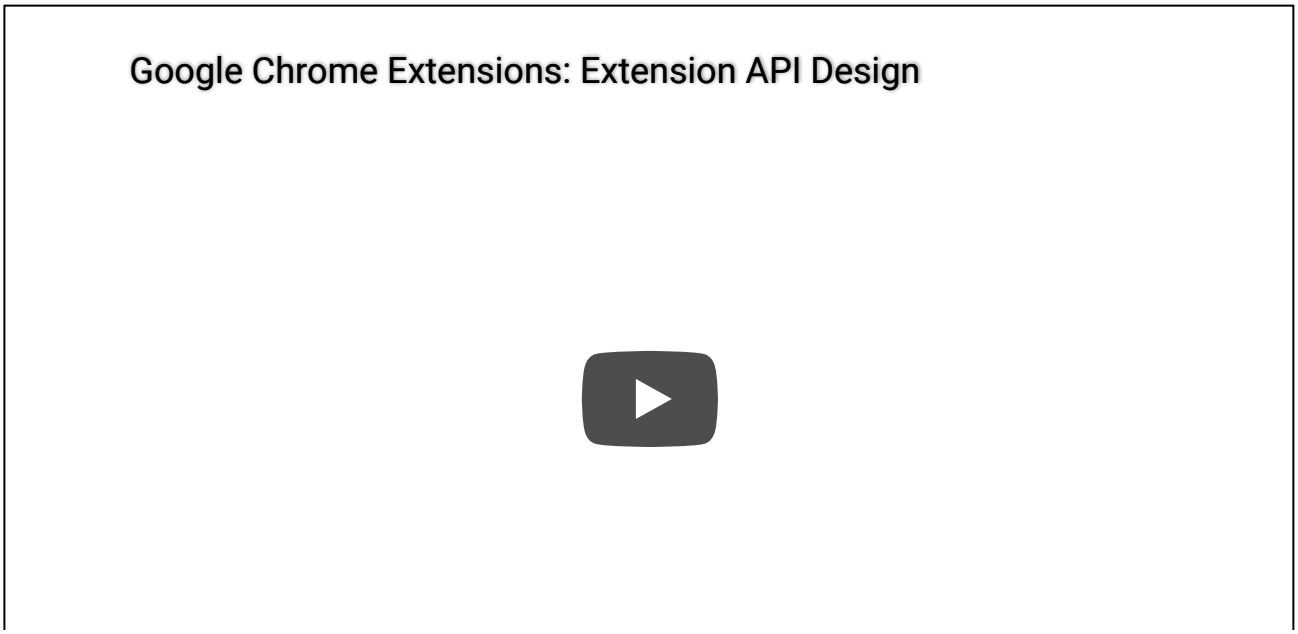
In the above code, the lines are executed in the following order: 1, 4, 2. The callback function specified to `query()` is called and then executes line 2, but only after information about the currently selected tab is available. This happens sometime after `query()` returns. Although `update()` is asynchronous the code doesn't use a callback parameter, since the extension doesn't do anything with the results of the update.

```
// Synchronous methods have no callback option and returns a type of string
string chrome.runtime.getURL()
```

This method synchronously returns the URL as a `string` and performs no other asynchronous work.

More details

For more information, explore the **Chrome API reference docs** and watch the following video.



Communication between pages

Different components in an extension often need to communicate with each other. Different HTML pages can find each other by using the `chrome.extension` methods, such as `getViews()` and `getBackgroundPage()`. Once a page has a reference to other extension pages the first one can invoke functions on the other pages and manipulate their DOMs. Additionally, all components of the extension can access values stored using the **storage** API and communicate through **message passing**.

Saving data and incognito mode

Extensions can save data using the **storage** API, the HTML5 **web storage API** , or by making server requests that result in saving data. When the extension needs to save something, first consider if it's from an incognito window. By default, extensions don't run in incognito windows.

Incognito mode promises that the window will leave no tracks. When dealing with data from incognito windows, extensions should honor this promise. If an extension normally saves browsing history, don't save history from incognito windows. However, extensions can store setting preferences from any window, incognito or not.

To detect whether a window is in incognito mode, check the `incognito` property of the relevant **tabs.Tab** or **windows.Window** object.

```
function saveTabData(tab) {
  if (tab.incognito) {
    return;
  } else {
    chrome.storage.local.set({data: tab.url});
  }
}
```

Take the Next Step

After reading the overview and completing the **Getting Started** tutorial, developers should be ready to start writing their own extensions! Dive deeper into the world of custom Chrome with the following resources.

- Learn about the options available for debugging Extensions in the **debugging tutorial**.

- Chrome Extensions have access to powerful APIs above and beyond what's available on the open web. The **chrome.* APIs documentation** will walk through each API.

- The **developer's guide** has dozens of additional links to pieces of documentation relevant to advanced extension creation.