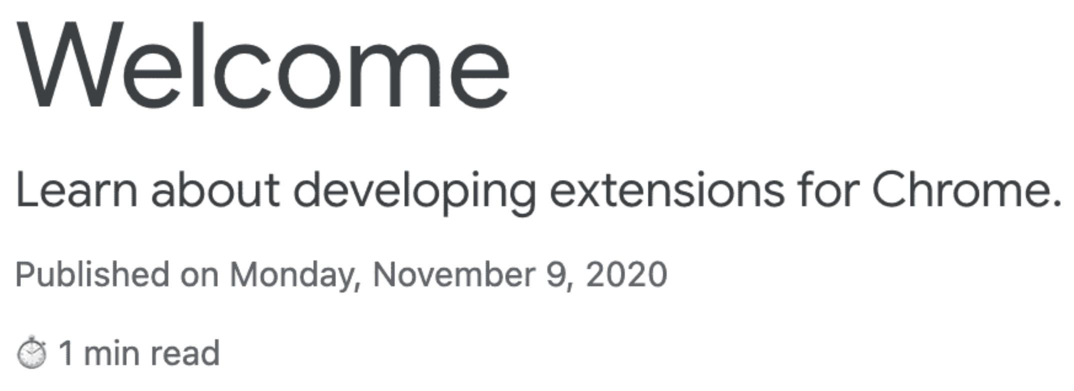
Run scripts on every page

Create your first extension that inserts a new element on the page.

Overview

This tutorial builds an extension that adds the expected reading time to any Chrome extension and Chrome Web Store documentation page.



Welcome

Learn about developing extensions for Chrome.

Published on Monday, November 9, 2020

⏱ 1 min read

Reading time extension on the extension's Welcome page.

In this guide, we're going to explain the following concepts:

- The extension manifest.
- What icon sizes an extension uses.
- How to inject code into pages using content scripts.
- How to use match patterns.
- Extension permissions.

Before you start

This guide assumes that you have basic web development experience. We recommend checking out the Hello world tutorial for an introduction to the extension development workflow.

Build the extension

To start, create a new directory called `reading-time` to hold the extension's files. If you prefer, you can download the complete source code from GitHub.

Step 1: Add information about the extension

The manifest JSON file is the only required file. It holds important information about the extension. Create a `manifest.json` file in the *root* of the project and add the following code:

```
{
  "manifest_version": 3,
  "name": "Reading time",
  "version": "1.0",
  "description": "Add the reading time to Chrome Extension documentation
articles"
}
```

These keys contain basic metadata for the extension. They control how the extension appears on the extensions page and, when published, on the Chrome Web Store. To dive deeper, check out the `"name"`, `"version"` and `"description"` keys on the Manifest overview page.

💡 **Other facts about the extension manifest**

- It must be located at the **root** of the project.
- The only required keys are `"manifest_version"`, `"name"`, and `"version"`.
- It supports comments (`//`) during development, but these must be removed before uploading your code to the Chrome Web Store.

Step 2: Provide the icons

So, why do you need icons? Although icons are optional during development, they are required if you plan to distribute your extension on the Chrome Web Store. They also appear in other places like the Extensions Management page.

Create an `images` folder and place the icons inside. You can download the icons on GitHub. Next, add the highlighted code to your manifest to declare icons:

```
{
  "icons": {
    "16": "images/icon-16.png",
    "32": "images/icon-32.png",
    "48": "images/icon-48.png",
    "128": "images/icon-128.png"
  }
}
```

We recommend using PNG files, but other file formats are allowed, except for SVG files.

💡 **Where are these differently-sized icons displayed?**

Icon size	Icon use
16x16	Favicon on the extension's pages and context menu.
32x32	Windows computers often require this size.
48x48	Displays on the Extensions page.
128x128	Displays on installation and in the Chrome Web Store.

Step 3: Declare the content script

Extensions can run scripts that read and modify the content of a page. These are called *content scripts*. They live in an isolated world, meaning they can make changes to their JavaScript environment without conflicting with their host page or other extensions' content scripts.

Add the following code to the `manifest.json` to register a content script called `content.js`.

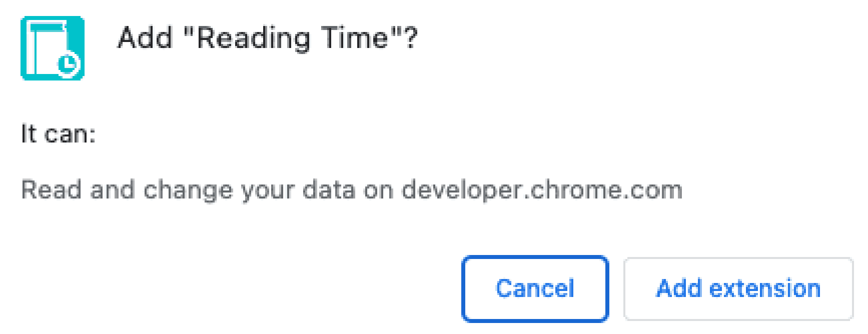
```
{
  "content_scripts": [
    {
      "js": ["scripts/content.js"],
      "matches": [
        "https://developer.chrome.com/docs/extensions/*",
        "https://developer.chrome.com/docs/webstore/*"
      ]
    }
  ]
}
```

The `"matches"` field can have one or more match patterns. These allow the browser to identify which sites to inject the content scripts into. Match patterns consist of three parts: `<scheme>://<host><path>`. They can contain '`*`' characters.

💡 **Does this extension display a permission warning?**

When a user installs an extension, the browser informs them what the extension can do. Content scripts request permission to run on sites that meet the match pattern criteria.

In this example, the user would see the following permission warning:


Add "Reading Time"?

It can:

Read and change your data on developer.chrome.com

Reading time permission warning.

To dive deeper on extension permissions, see Declaring permissions and warn users.

Step 4: Calculate and insert the reading time

Content scripts can use the standard Document Object Model (DOM) to read and change the content of a page. The extension will first check if the page contains the `<article>` element. Then, it will count all the words within this element and create a paragraph that displays the total reading time.

Create a file called `content.js` inside a folder called `scripts` and add the following code:

```
const article = document.querySelector("article");
```

```
// `document.querySelector` may return null if the selector doesn't match
anything.
if (article) {
  const text = article.textContent;
  const wordMatchRegExp = /[^\s]+/g; // Regular expression
  const words = text.matchAll(wordMatchRegExp);
  // matchAll returns an iterator, convert to array to get word count
  const wordCount = [...words].length;
  const readingTime = Math.round(wordCount / 200);
  const badge = document.createElement("p");
  // Use the same styling as the publish information in an article's header
  badge.classList.add("color-secondary-text", "type--caption");
  badge.textContent = `⏱️ ${readingTime} min read`;

  // Support for API reference docs
  const heading = article.querySelector("h1");
  // Support for article docs with date
  const date = article.querySelector("time")?.parentNode;

  (date ?? heading).insertAdjacentElement("afterend", badge);
}
```

💡 **Interesting JavaScript used in this code**

- Regular expressions used to count only the words inside the `<article>` element.
- `insertAdjacentElement()` used to insert the reading time node after the element.
- The classList property used to add CSS class names to the element class attribute.
- Optional chaining used to access an object property that may be undefined or null.
- Nullish coalescing returns the `<heading>` if the `<date>` is null or undefined.

Test that it works

Verify that the file structure of your project looks like the following:

```
└── reading-time/
    ├── manifest.json
    ├── scripts/
    │   └── content.js
    └── images/
        ├── icon-16.png
        ├── icon-32.png
        ├── icon-48.png
        └── icon-128.png
```

Load your extension locally

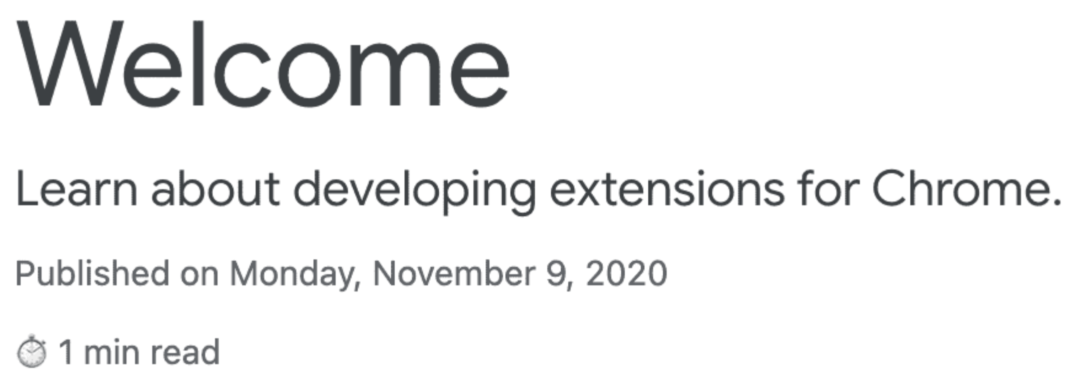
To load an unpacked extension in developer mode, follow the steps in Development Basics.

Open an extension or Chrome Web Store documentation

Here are a few pages you can open to see how long each article will take to read.

- Publish in the Chrome Web Store
- Understanding Content Scripts

It should look like this:



Welcome

Learn about developing extensions for Chrome.

Published on Monday, November 9, 2020

⏱️ 1 min read

Extension Welcome page with the Reading time extension

🎯 Potential enhancements

Based on what you've learned today, try to implement any of the following:

- Add another **match pattern** in the manifest.json to support other chrome developer pages, like for example, the Chrome DevTools or Workbox.
- Add a new content script that calculates the reading time to any of your favorite blogs or documentation sites.

Hint: You can use DevTools to inspect DOM elements.

Keep building

Congratulations on finishing this tutorial 🎉. Continue building your skills by completing other tutorials on this series:

Extension	What you will learn
Focus Mode	To run code on the current page after clicking the extension action.
Tabs Manager	To create a popup that manages browser tabs.

Continue exploring

We hope you enjoyed building this Chrome extension and are excited to continue your Chrome development learning journey. We recommend the following learning path:

- The developer's guide has dozens of additional links to pieces of documentation relevant to advanced extension creation.
- Extensions have access to powerful APIs beyond what's available on the open web. The Chrome APIs documentation walks through each API.