

Create Your First App

This tutorial walks you through creating your first Chrome App. Chrome Apps are structured similarly to extensions so current developers will recognize the manifest and packaging methods. When you're done, you'll just need to produce a zip file of your code and assets in order to [publish](#) your app.

A Chrome App contains these components:

- The **manifest** tells Chrome about your app, what it is, how to launch it and the extra permissions that it requires.
- The **background script** is used to create the event page responsible for managing the app life cycle.
- All code must be included in the Chrome App package. This includes HTML, JS, CSS and Native Client modules.
- All **icons** and other assets must be included in the package as well.

API Samples: Want to play with the code? Check out the [hello-world](#) sample.

Step 1: Create the manifest

First create your `manifest.json` file ([Formats: Manifest Files](#) describes this manifest in detail):

```
{
  "name": "Hello World!",
  "description": "My first Chrome App.",
  "version": "0.1",
  "manifest_version": 2,
  "app": {
    "background": {
      "scripts": ["background.js"]
    }
  },
  "icons": { "16": "calculator-16.png", "128": "calculator-128.png" }
}
```

Important: Chrome Apps **must** use [manifest version 2](#).

Step 2: Create the background script

Next create a new file called `background.js` with the following content:

```
chrome.app.runtime.onLaunched.addListener(function() {
  chrome.app.window.create('window.html', {
    'outerBounds': {
      'width': 400,
      'height': 500
    }
  });
});
```

```
    }  
  });  
});
```

In the above sample code, the [onLaunched event](#) will be fired when the user starts the app. It then immediately opens a window for the app of the specified width and height. Your background script may contain additional listeners, windows, post messages, and launch data, all of which are used by the event page to manage the app.

Step 3: Create a window page

Create your `window.html` file:

```
<!DOCTYPE html>  
<html>  
  <head>  
  </head>  
  <body>  
    <div>Hello, world!</div>  
  </body>  
</html>
```

Step 4: Create the icons

Copy these icons to your app folder:

- [calculator-16.png](#)
- [calculator-128.png](#)


Step 5: Launch your app

Enable flags

Many of the Chrome Apps APIs are still experimental, so you should enable experimental APIs so that you can try them out:

- Go to **chrome://flags**.
- Find "Experimental Extension APIs", and click its "Enable" link.
- Restart Chrome.

Load your app

To load your app, bring up the apps and extensions management page by clicking the settings icon  and choosing **Tools > Extensions**.

Make sure the **Developer mode** checkbox has been selected.

Click the **Load unpacked extension** button, navigate to your app's folder and click **OK**.

Open new tab and launch

Once you've loaded your app, open a New Tab page and click on your new app icon.

Or, load and launch from command line

These command line options to Chrome may help you iterate:

- `--load-and-launch-app=/path/to/app/` installs the unpacked application from the given path, and launches it. If the application is already running it is reloaded with the updated content.
- `--app-id=ajjhbohkpincjgiieeomimlgnll` launches an app already loaded into Chrome. It does not restart any previously running app, but it does launch the new app with any updated content.

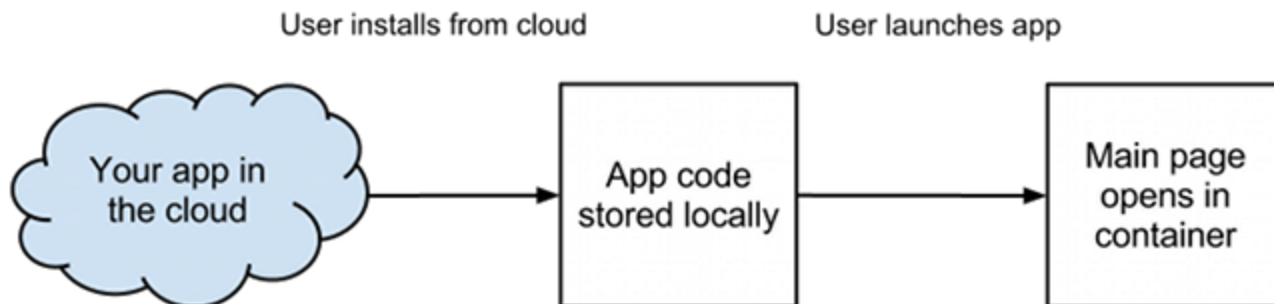
Chrome Apps Architecture

Chrome Apps integrate closely with a user's operating system. They are designed to be run outside of a browser tab, to run robustly in offline and poor connectivity scenarios and to have far more powerful capabilities than are available in a typical web browsing environment. The app container, programming, and security models support these Chrome App requirements.

App container model

The app container describes the visual appearance and loading behavior of Chrome Apps. Chrome Apps look different than traditional web apps because the app container does not show any traditional web page UI controls; it simply contains a blank rectangular area. This allows an app to blend with “native” apps on the system, and it prevents the user from “messing” with the app logic by manually changing the URL.

Chrome Apps are loaded differently than web apps. Both load the same type of content: HTML documents with CSS and JavaScript; however, a Chrome App is loaded in the app container, not in the browser tab. Also, the app container must load the main document of the Chrome App from a local source. This forces all Chrome Apps to be at least minimally functional when offline and it provides a place to enforce stricter security measures.



Programming model

The programming model describes the lifecycle and window behavior of Chrome Apps. Similar to native apps, the goal of this programming model is to give users and their systems full control over the app lifecycle. The Chrome App lifecycle should be independent of browser window behavior or a network connection.

The “event page” manages the Chrome App lifecycle by responding to user gestures and system events. This page is invisible, only exists in the background, and can be closed automatically by the system runtime. It controls how windows open and close and when the app is started or terminated. There can only be one “event page” for a Chrome App.

App lifecycle at a glance

For detailed instructions on how to use the programming model, see [Manage App Lifecycle](#). Here's a brief summary of the Chrome App lifecycle to get you started:

Stage	Summary
Installation	User chooses to install the app and explicitly accepts the permissions .
Startup	The event page is loaded, the 'launch' event fires, and app pages open in windows. You create the windows that your app requires, how they look, and how they communicate with the event page and with other windows.
Termination	User can terminate apps at any time and app can be quickly restored to previous state. Stashing data protects against data loss.
Update	Apps can be updated at any time; however, the code that a Chrome App is running cannot change during a startup/termination cycle.
Uninstallation	User can actively uninstall apps. When uninstalled, no executing code or private data is left behind.

Security model

The Chrome Apps security model protects users by ensuring their information is managed in a safe and secure manner. [Comply with CSP](#) includes detailed information on how to comply with content security policy. This policy blocks dangerous scripting reducing cross-site scripting bugs and protecting users against man-in-the-middle attacks.

Loading the Chrome App main page locally provides a place to enforce stricter security than the web. Like Chrome extensions, users must explicitly agree to trust the Chrome App on install; they grant the app permission to access and use their data. Each API that your app uses will have its own permission. The Chrome Apps security model also provides the ability to set up privilege separation on a per window basis. This allows you to minimize the code in your app that has access to dangerous APIs, while still getting to use them.

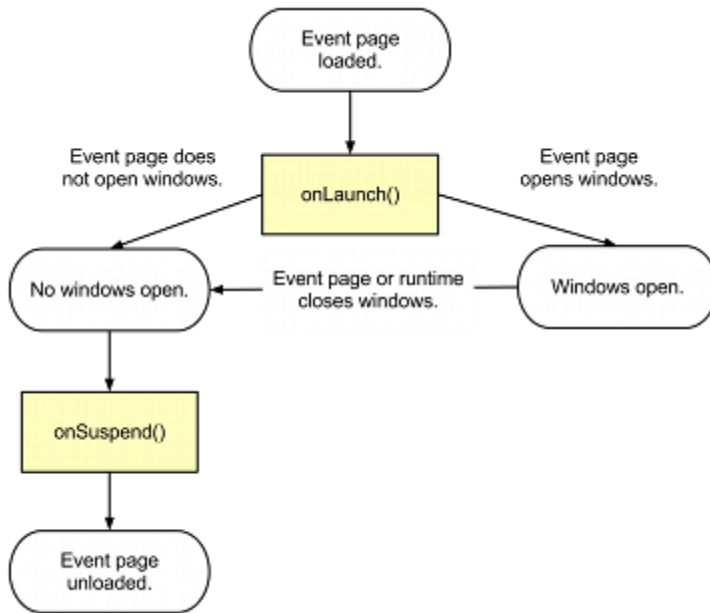
Chrome Apps reuse Chrome extension process isolation, and take this a step further by isolating storage and external content. Each app has its own private storage area and can't access the storage of another app or personal data (such as cookies) for websites that you use in your browser. All external processes are isolated from the app. Since iframes run in the same process as the surrounding page, they can only be used to load other app pages. You can use the `object` tag to [embed external content](#); this content runs in a separate process from the app.

Chrome App Lifecycle

The app runtime and event page are responsible for managing the app lifecycle. The app runtime manages app installation, controls the event page, and can shutdown the app at anytime. The event page listens out for events from the app runtime and manages what gets launched and how.

How the lifecycle works

The app runtime loads the event page from a user's desktop and the `onLaunch()` event is fired. This event tells the event page what windows to launch and their dimensions. The lifecycle diagram here isn't the nicest to look at, but it's practical (and we will make it nicer soon).



When the event page has no executing JavaScript, no pending callbacks, and no open windows, the runtime unloads the event page and closes the app. Before unloading the event page, the `onSuspend()` event is fired. This gives the event page opportunity to do simple clean-up tasks before the app is closed.

Create event page and windows

All apps must have an event page. This page contains the top-level logic of the application with none of its own UI and is responsible for creating the windows for all other app pages.

Create event page

To create the event page, include the "background" field in the app manifest and include the `background.js` in the scripts array. Any library scripts used by the event page need to be added to the "background" field first:

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

Your event page must include the `onLaunched()` function. This function is called when your application is launched in any way:

```
chrome.app.runtime.onLaunched.addListener(function() {  
    // Tell your app what to launch and how.  
});
```

Create windows

An event page may create one or more windows at its discretion. By default, these windows are created with a script connection to the event page and are directly scriptable by the event page.

Windows in Chrome Apps are not associated with any Chrome browser windows. They have an optional frame with title bar and size controls, and a recommended window ID. Windows without IDs will not restore to their size and location after restart.

Here's a sample window created from `background.js`:

```
chrome.app.runtime.onLaunched.addListener(function() {  
    chrome.app.window.create('main.html', {  
        id: 'MyWindowID',  
        bounds: {  
            width: 800,  
            height: 600,  
            left: 100,  
            top: 100  
        },  
        minWidth: 800,  
        minHeight: 600  
    });  
});
```

Including Launch Data

Depending on how your app is launched, you may need to handle launch data in your event page. By default, there is no launch data when the app is started by the app launcher. For apps that have file handlers, you need to handle the `launchData.items` parameter to allow them to be launched with files.

Listening for app runtime events

The app runtime controls the app installs, updates, and uninstalls. You don't need to do anything to set up the app runtime, but your event page can listen out for the `onInstalled()` event to store local settings and the `onSuspend()` event to do simple clean-up tasks before the event page is unloaded.

Storing local settings

`chrome.runtime.onInstalled()` is called when your app has first been installed, or when it has been updated. Any time this function is called, the `onInstalled` event is fired. The event page can listen for this event and use the [Storage API](#) to store and update local settings (see also [Storage options](#)).

```
chrome.runtime.onInstalled.addListener(function() {  
  chrome.storage.local.set(object items, function callback);  
});
```

Preventing data loss

Users can uninstall your app at any time. When uninstalled, no executing code or private data is left behind. This can lead to data loss since the users may be uninstalling an app that has locally edited, unsynchronized data. You should stash data to prevent data loss.

At a minimum, you should store user settings so that if users reinstall your app, their information is still available for reuse. Using the Storage API ([storage.sync](#)), user data can be automatically synced with Chrome sync.

Clean-up before app closes

The app runtime sends the `onSuspend()` event to the event page before unloading it. Your event page can listen out for this event and do clean-up tasks and save state before the app closes.

Once this event is fired, the app runtime starts the process of closing the app. If the app has open windows it may be restarted in the future via the `onRestarted` event. In this case the app should save its current state to persistent storage so it can restart in the same state if it receives an `onRestarted` event.

The app only has a few seconds to save its state, after which it will be terminated, so it is a good idea to be incrementally saving app state while the app is running normally.

After receiving `onSuspend` no further events will be delivered to the app, unless the suspension is aborted for some reason. In this case the `onSuspendCanceled` will be delivered to the app, and the app won't be unloaded.

```
chrome.runtime.onSuspend.addListener(function() {  
  // Do some simple clean-up tasks.  
});
```

Content Security Policy

If you're not familiar with Content Security Policy (CSP), [An Introduction to Content Security Policy](#) is a good starting point. That document covers the broader web platform view of CSP; Chrome App CSP isn't as flexible. You should also read the [Chrome extension Content Security Policy](#), as it's the foundation for the Chrome App CSP. For brevity's sake, we don't repeat the same information here.

CSP is a policy to mitigate against cross-site scripting issues, and we all know that cross-site scripting is bad. We aren't going to try and convince you that CSP is a warm-and-fuzzy new policy. There's work involved; you'll need to learn how to do fundamental tasks differently.

The purpose of this document is to tell you exactly what the CSP policy is for Chrome Apps, what you need to do to comply with it, and how you can still do those fundamental tasks in a way that is CSP-compliant.

What is the CSP for Chrome Apps?

The content security policy for Chrome Apps restricts you from doing the following:

- You can't use inline scripting in your Chrome App pages. The restriction bans both `<script>` blocks and event handlers (`<button onclick="...">`).
- You can't reference any external resources in any of your app files (except for video and audio resources). You can't embed external resources in an `iframe`.
- You can't use string-to-JavaScript methods like `eval()` and `new Function()`.

This is implemented via the following policy value:

```
default-src 'self';
connect-src * data: blob: filesystem;;
style-src 'self' data: chrome-extension-resource: 'unsafe-inline';
img-src 'self' data: chrome-extension-resource;;
frame-src 'self' data: chrome-extension-resource;;
font-src 'self' data: chrome-extension-resource;;
media-src * data: blob: filesystem;;
```

Your Chrome App can only refer to scripts and objects within your app, with the exception of media files (apps can refer to video and audio outside the package). Chrome extensions will let you relax the default Content Security Policy; Chrome Apps won't.

How to comply with CSP

All JavaScript and all resources should be local (everything gets packaged in your Chrome App).

"But then how do I..."

It's very possible that you are using templating libraries and many of these won't work with CSP. You may also want to access external resources in your app (external images, content from websites).

Use templating libraries

Use a library that offers precompiled templates and you're all set. You can still use a library that doesn't offer precompilation, but it will require some work on your part and there are restrictions.

You will need to use sandboxing to isolate any content that you want to do 'eval' things to. Sandboxing lifts CSP on the content that you specify. If you want to use the very powerful Chrome APIs in your Chrome App, your sandboxed content can't directly interact with these APIs (see [Sandbox local content](#)).

Access remote resources

You can fetch remote resources via `XMLHttpRequest` and serve them via `blob:`, `data:`, or `filesystem:`URLs (see [Referencing external resources](#)).

Video and audio can be loaded from remote services because they have good fallback behavior when offline or under spotty connectivity.

Embed web content

Instead of using an `iframe`, you can call out to an external URL using a `webview` tag (see [Embed external web pages](#)).