

S I G N A L C U L T U R E C O O K B O O K

VOL. 2

Edited by Jason Bernagozzi
E-Book Designed by Brian Murphy



**S I G N A L
C U L T U R E
C O O K
B O O K**

VOL. 2

Edited by Jason Bernagozzi
Introduction by Jason Bernagozzi
E-Book Designed by Brian Murphy

All Rights Reserved
Copyright © 2019 Signal Culture
Owego, New York
<http://signalculture.org/>
ISBN:

Browser Blowup: Explode Web Pages Containing Third-Party Trackers

Joelle Dietrick, Gretta Louw, Owen Mundy



Introduction

This chapter will discuss browser extensions and online tracking. We'll speak to both common and more experimental implementations of extensions, and show readers how to build their own cross browser extension that explodes web pages with hidden third-party data trackers on them.

What are browser extensions?

Browser extensions are software that add features to a web browser. The functionality they add can be useful, like the **Wayback Machine** extension that shows you what a web page looked like in the past, even if it was deleted, or **Google Translate** which changes text on a web page into any language. Others are more whimsical, such as the **Meow Met** (2015) extension created by Emily McAllister at the Metropolitan Museum of Art, which displays a new random image from their collection that contains a cat whenever you open a new browser window or tab¹.

One of the most popular browser extensions on the Chrome Web Store is Adblock which blocks annoying banner, popup, and video advertisements in real time². Ad blockers are a subset of a larger group of “tracker blockers,” which prevent hidden online data trackers from quantifying and monetizing everything from internet users’ most private searches and communications to their shopping habits, interests, and intimate demographic details.

While the vast majority of users are not aware of the extent to which they are being tracked; think that they ‘have nothing to hide’ and therefore that tracking is not a concern; or do not like being tracked but feel powerless to do anything about it,^{3 4} a significant share are actively resisting this mindset through tracker blockers. According to the website eMarketer, the number of internet users in the U.S. who have an ad blocker enabled has doubled in the last four years, from 15% in 2014 to a substantial 30%, or about 80 million people, projected for 2018⁵.

1 Claire Voon, “Cat Art from the Met Museum Makes the Purrfect Browser Plug-in,” Hyperallergic, July 16, 2015, accessed July 23, 2018. <https://hyperallergic.com/222538/cat-art-from-the-met-museum-makes-the-purrfect-browser-plug-in/>

2 “Choosing the right filterlist,” Adblock Plus, Accessed July 23, 2018. https://adblockplus.org/en/getting_started#-subscription

3 Mary Madden and Lee Rainie, “Americans’ Attitudes About Privacy, Security and Surveillance,” Pew Research Center, May 20, 2015, Accessed July 23, 2018. <http://www.pewinternet.org/2015/05/20/americans-attitudes-about-privacy-security-and-surveillance/>

4 Joseph Turow, Michael Hennessey and Nora Draper, “The Tradeoff Fallacy: How Marketers Are Misrepresenting American Consumers And Opening Them Up to Exploitation,” The Annenberg School for Communication at the University of Pennsylvania, June 2015, Accessed July 23, 2018. https://www.asc.upenn.edu/sites/default/files/TradeoffFallacy_1.pdf

5 “US Ad Blocking User Penetration, 2014-2018 (% of internet users),” eMarketer, February 22, 2017, Accessed July 23, 2018. <https://www.emarketer.com/Chart/US-Ad-Blocking-User-Penetration-2014-2018-of-internet-users/204561>

Most ad and tracker blockers work by accessing the source code of a web page, identifying content areas that are known to import third-party trackers or show advertisements, and deleting that source code. This is possible because we can view the content of web pages using the browser's "view source" option. The view source option has been available almost as long as web browsers have been in use and is an essential part of what makes the internet "open." With it, anyone can see how a web page is constructed, to inspect or learn from others' code, and improve the web. As Stack Overflow co-founder Jeff Atwood explained on his popular blog Coding Horror, it is the "ultimate form of open source" and an important reason for why the internet is so accessible today⁶. Mark Surman, the Executive Director of the Mozilla Foundation, argues for "view source" not only as pragmatic, but instead, as fundamental to the transparency, openness, and collaborative goals that underpin all of civil society⁷.

Thanks to the fact we have access to the source of web pages, there are more experimental uses of browser extensions that challenge our notion of what is possible online. Because browser extensions can access a web page's source code, they can transform the content of pages, modifying our experience while we search, work, or play online.

6 Jeff Atwood. "The Power of 'View Source,'" Coding Horror (blog), August 17, 2006, Accessed July 23, 2018. <https://blog.codinghorror.com/the-power-of-view-source/>

7 Mark Surman and Jason Diceman, "Choosing Open Source: A decision-making guide for civil society organizations," January 2004, Accessed July 23, 2018. <https://marksurman.commons.ca/publications/choosing-open-source-a-decision-making-guide-for-civil-society-organizations/full-text/>

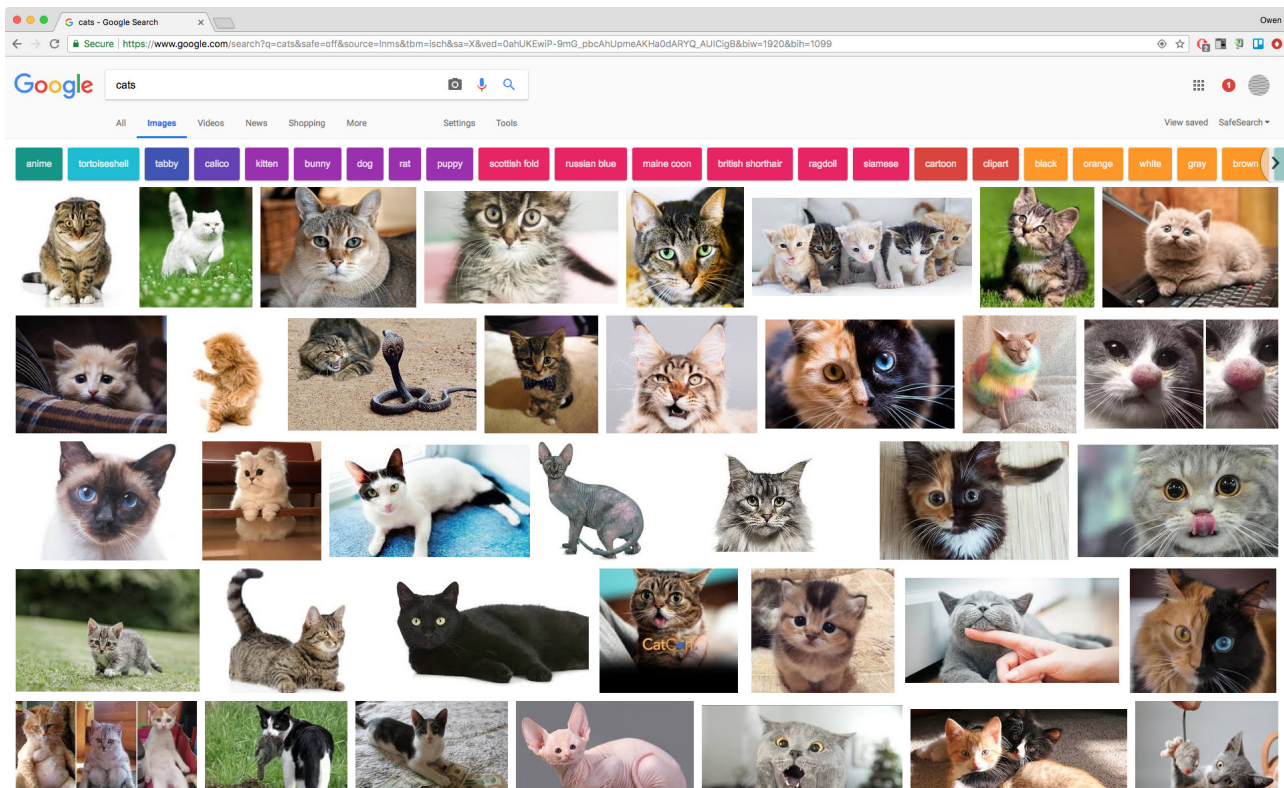


Figure 0.1



Figure 0.2

Some extensions alter the source in order to obscure web pages, like Rafaël Rozendaal's **Abstract Browsing** (2016) which transfigures all block-level HTML elements into brightly-colored rectangles reminiscent of a modernist artwork. The extension does more than make it pretty. It reminds us that the content of the web is mutable, following Richard Stallman's decree that software is only useful when it "respects the users' essential freedoms." He says users should have "the freedom to run it, to study and change it, and to redistribute copies with or without changes."⁸ This freedom, which Stallman states is a philosophical matter, like freedom of speech (not free as in beer), is essential in a free and transparent society - in which we should have access to not only the source of the content of web pages, but the origins of our food, dealings by our governments, and knowledge of what happens to our information.

Ben Grosser's **Facebook Demetricator** (2012) hides the barely noticable numbers that motivate our interactions with others on social media. Grosser's software removes the power of the scoring systems that permeate these spaces. He detaches these psychological

⁸ Richard Stallman, "Why Open Source misses the point of Free Software," November 18, 2016, Accessed July 23, 2018. <http://www.gnu.org/philosophy/open-source-misses-the-point.html>

motivators which urge us to interact, for fear of missing out (because others have already “liked” it) or because accumulation is unfortunately rewarded more than thoughtful interaction.

Another example of an extension that combines utility and experimentation within the browser is **Google Alarm** (2010) produced “copyfree” by Jamie Wilkinson and Greg Leuch with the **ffff.at lab**. Similar to tracker blockers today, this extension watches the source code of web pages, but when it finds a reference to a Google-owned tracker, that code immediately triggers a loud, air horn sound and a siren gif in the browser⁹. *Google Alarm* is not subtle, yet perhaps that is what we need to alter the course of the increasingly commercialized and surveilled spaces of the internet.

Accessing the source code and blocking trackers is also a core mechanic in our in-production browser game to be released next year. **Tally**, the name of our game and its central character, teaches players about hidden data trackers and algorithmic profiling by awarding users points when they battle and capture hidden “product monsters” (visual representations of trackers and algorithmic profiling) and blocking those trackers in the process. Through using industry product marketing categories and gamifying the process of surveilling users, Tally transforms the entire internet into a game, redefining the power balance and allowing users to “play” the advertisers, instead of the other way around.

In the following tutorial, we walk you through the process of creating part of our game. When users beat product monsters in a turn-based “Pokémon-style” battle, their reward, in addition to blocking a tracker from that site, is to see the entire web page “explode”, reminding them that with Tally, they are exercising certain freedoms and doing their part to deconstruct the surveillance economy. This is easily fixed by reloading the page, but is a fun, and engaging tactic to - as we have done in a modification here - alert users they’re being tracked.

Building cross-browser extensions

In this section we’ll learn to create a cross-browser extension, starting with a simple “hello world” example, and progressively building up to a working extension that temporarily explodes a user’s web page whenever it detects a tracker. This tutorial assumes you have some familiarity with writing code. For example, you know what a variable does. We’ll start easy and then get more advanced so check out the W3Schools **HTML**, **CSS**, and **Javascript** tutorials as needed.

You’ll need three things for this tutorial: a web browser like **Chrome** or **Firefox**, a code editor like **Sublime Text** or **Atom**, and the project assets, which can be downloaded from our **Github repository**. We’ll be using Chrome and Atom primarily, but will occasionally explain significant differences in Firefox. Here are some suggestions for coding this tutorial, and making works for the internet in general:

⁹ Gillian Tee, “‘Google Alarm’ plug-in tries to wake the world up to privacy issues,” CNN, August 6, 2010, Accessed July 23, 2018. <http://www.cnn.com/2010/TECH/web/08/06/google.alarm/index.html>

1. File organization is key. Keep your files in folders (a.k.a. “directories”) with descriptive names. Use version control software (like [Git](#)) or a naming convention that helps you organize your own files.
2. Pay careful attention to code syntax. A misplaced comma or quotation mark can break your entire project. Use a validator like [jsonlint.com](#) to check your code occasionally.
3. Use standard [file-naming conventions](#). The web is case-sensitive, so use capitalization sparingly and deliberately. Avoid using spaces altogether when creating files for the web. Substitute spaces with either [camelCase](#), underscores, or hyphens. For example, “Explode the Web” becomes either **explodeTheWeb** or **explode-the-web**.

Part 1: Hello World!

Most [browser extensions](#) are made using HTML, CSS, and Javascript code. They are installed to the browser either locally (for development and testing) or packaged and then published on the [Chrome Web Store](#) or the [Firefox Add-ons page](#).

The simplest possible browser extension contains a single [manifest.json](#) file, which specifies only the metadata required to load an extension into the browser. This is how we’ll start our project. First, download the [Github repository](#) and unzip it. Open the entire directory in Atom by either dragging it to the Atom icon in your dock (on Mac) or opening Atom first, then choosing **File > Add Project Folder...** and select the **explode-the-web** folder.

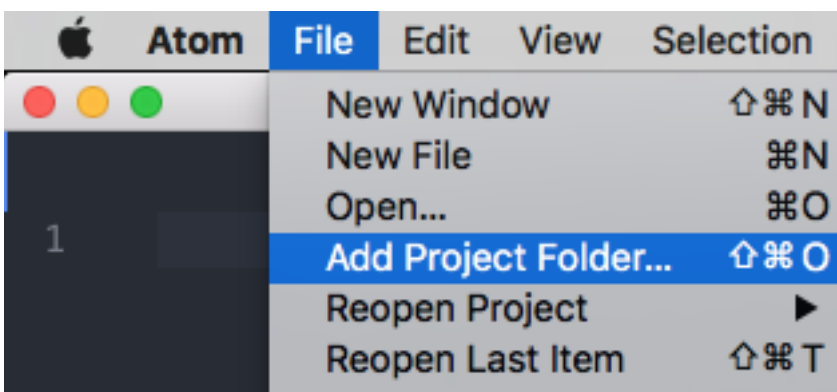


Figure 1.0

Inside this folder you will see the following directories

- /extension** - Our completed browser extension
- /figures** - A list of figures for this tutorial
- /tutorial** - The directory where we'll do our work

As you see, the tutorial directory contains a **manifest.json** file, a folder of assets that we'll add to our project, and a sections-complete folder, with completed copies of only the files we modify in each section. You can refer to these completed copies as needed.

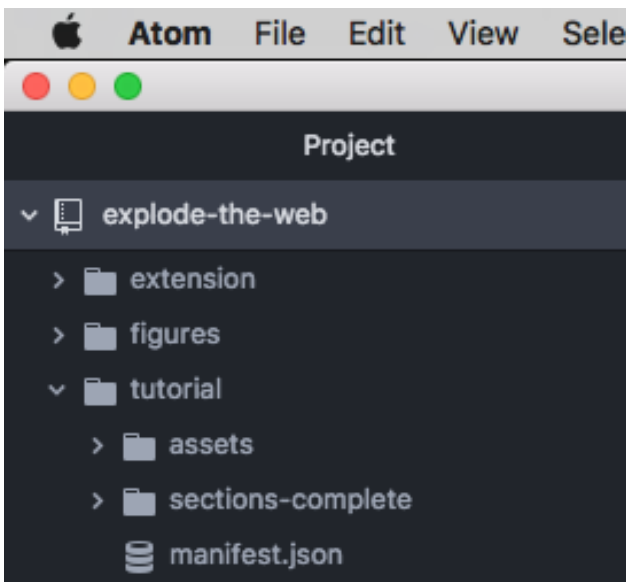


Figure 1.1

Click on the **manifest.json** file in Atom (Figure 1.2). A manifest is made using JSON (JavaScript Object Notation), a hierarchical data format with properties and values that are readable by both humans and computers. Each property has a *key*, like “name,” and a corresponding *value*, like “Explode the Web!”. Note they are both individually contained by quotations and the pairs are separated by a comma. Let's install this extension in our browser and then add features to it.

A screenshot of the Atom IDE showing the content of the 'manifest.json' file. The file name 'manifest.json' is in the tab. The code is as follows:

```
1  {  
2    "name": "Explode the Web!",  
3    "description": "Explodes web pages that contain data trackers!",  
4    "version": "1.0",  
5    "manifest_version": 2  
6  }
```

Figure 1.2

To install an in-development extension in Chrome

1. Navigate to **chrome://extensions**
2. Make sure **Developer Mode** is “on”
3. Click **Load Unpacked**, and select the folder that contains the **manifest.json** file
4. You should see the extension appear in the list like in this image (Figure 1.3)

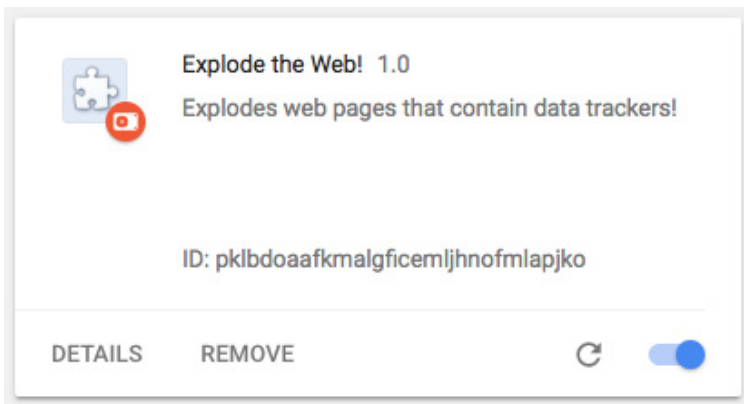


Figure 1.3

To install an in-development extension in Firefox

1. Navigate to **about:debugging** in Firefox
2. Click **Load Temporary Add-on** and select the **manifest.json** in the extension directory
3. The extension will now be installed, and will stay so until you restart Firefox.
4. You should see the extension appear in the list like in this image (Figure 1.4)

Temporary Extensions



Explode the Web!

This WebExtension has a temporary ID. [Learn more](#)

Location	/Users/owmundy/Sites/owenmundy.../owenmundy.com/work/explode-the-web/
Extension ID	6954b736021805adb130db0b024ed393f46e9294@temporary-addon
Internal UUID	49500dc9-b38e-3a42-a371-c92fac9a723e Manifest URL

[Debug](#) [Reload](#) [Remove](#)

Figure 1.4

Now, let's change the version property from 1.0 to 1.1 and save the file. In Chrome, return to the **chrome://extensions** page and note the version has *not* changed. Normally, if you are building web pages you would reload the web page in the browser and see your changes, but that won't work here. To view changes to an extension you must reload it by clicking the small reload button at the bottom right. Do that and you will see your version update to 1.1.



Explode the Web! 1.1

Explodes web pages that contain data trackers!

ID: pkldbdaafkmalgfcemljhnofmlapjko

DETAILS

REMOVE



Figure 1.5

If you see an error then you should check your syntax. A misplaced or missing comma or quotation mark is usually the issue. JSON files in particular can be very finicky about trailing commas and other seemingly inconsequential characters. Feel free to copy and paste from the **example code** at any point in this tutorial in case you get stuck.

To recap, this is the workflow, we will repeat as we add to our extension:

1. Edit the file's contents in Atom
2. Return to the browser's extension page and refresh the extension
3. Then, refresh a test web page to see your changes.

Congratulations! You've made your first extension and know how to create and view changes! Now let's add some features. Create a new file in Atom, in the same directory as your manifest, called **content.js** and add the code in Figure 1.6.

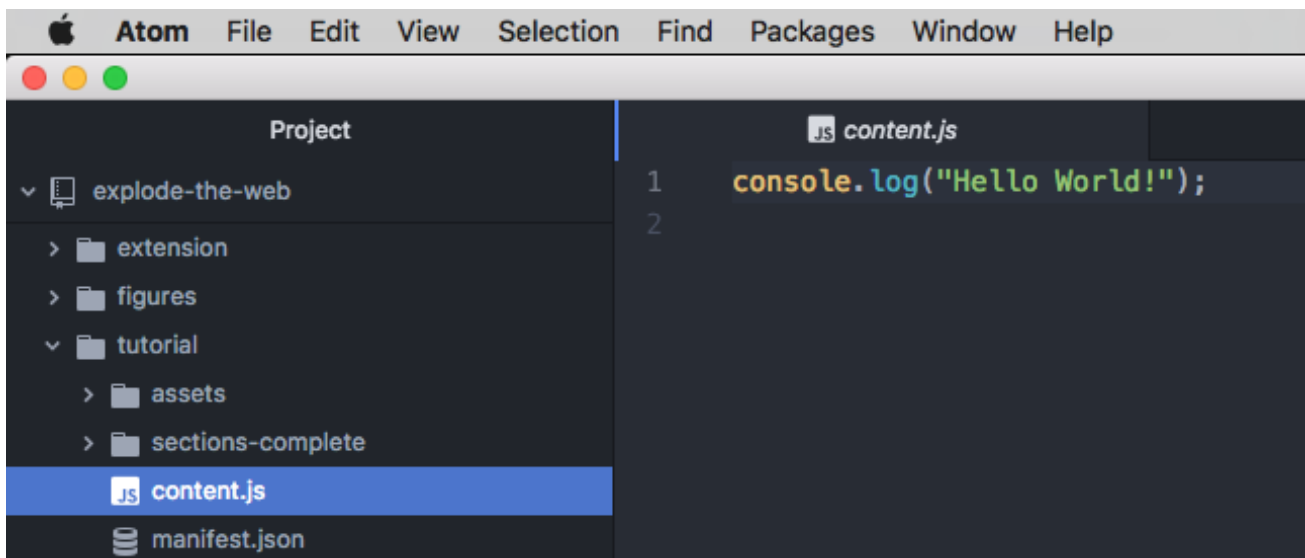
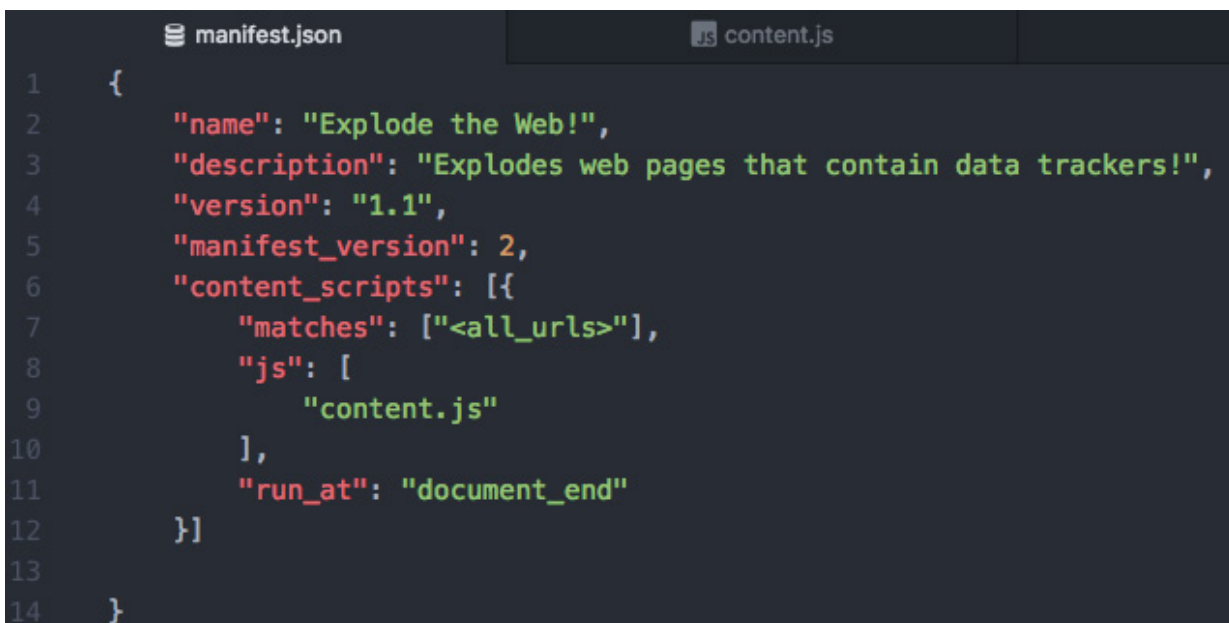


Figure 1.6

According to the Mozilla Developer Network (MDN) documentation, a **content script** is “a part of your extension that runs in the context of a particular web page.” That is, they run when your users open web pages and can access and change the content of those pages. The single line of code you added uses a “function” (a collection of code that can be referenced by a single command) to “log” messages (the content inside the parentheses) to the browser’s console. In this case we are displaying a string of text (everything inside the quotes).

Next, reference the **content.js** file in the manifest by adding the new properties in the code in Figure 1.7. This tells our extension to add **content.js** to our extension and run it on every web page, after that page loads.



```
1  {
2    "name": "Explode the Web!",
3    "description": "Explodes web pages that contain data trackers!",
4    "version": "1.1",
5    "manifest_version": 2,
6    "content_scripts": [{
7      "matches": ["<all_urls>"],
8      "js": [
9        "content.js"
10     ],
11     "run_at": "document_end"
12   }]
13 }
14 }
```

Figure 1.7

Save all your files and refresh the extension at **chrome://extensions**. Then open a new browser tab, view the console with **View > Developer > Javascript Console**, and navigate to a web page. You should see the message we typed above appear in the console. Nearly every programming language has a feature like the console. Consoles are essential for debugging and helping to confirm your code is working. **Browser consoles** also allow us to examine details about a web page’s code, structure, and performance.



Figure 1.8

Part 2: The Document Object Model (DOM)

Browser extensions load every time a user loads a new web page, giving them unique access to that page's **Document Object Model** (or DOM for short). The DOM is a representation of a web page's structure, style, and content and provides interfaces for reading or changing the look or functionality of that page. For example, by accessing the DOM, an ad blocker extension can compare links in a page to a "filter list" of known advertisers and prevent parts of a page from being loaded.

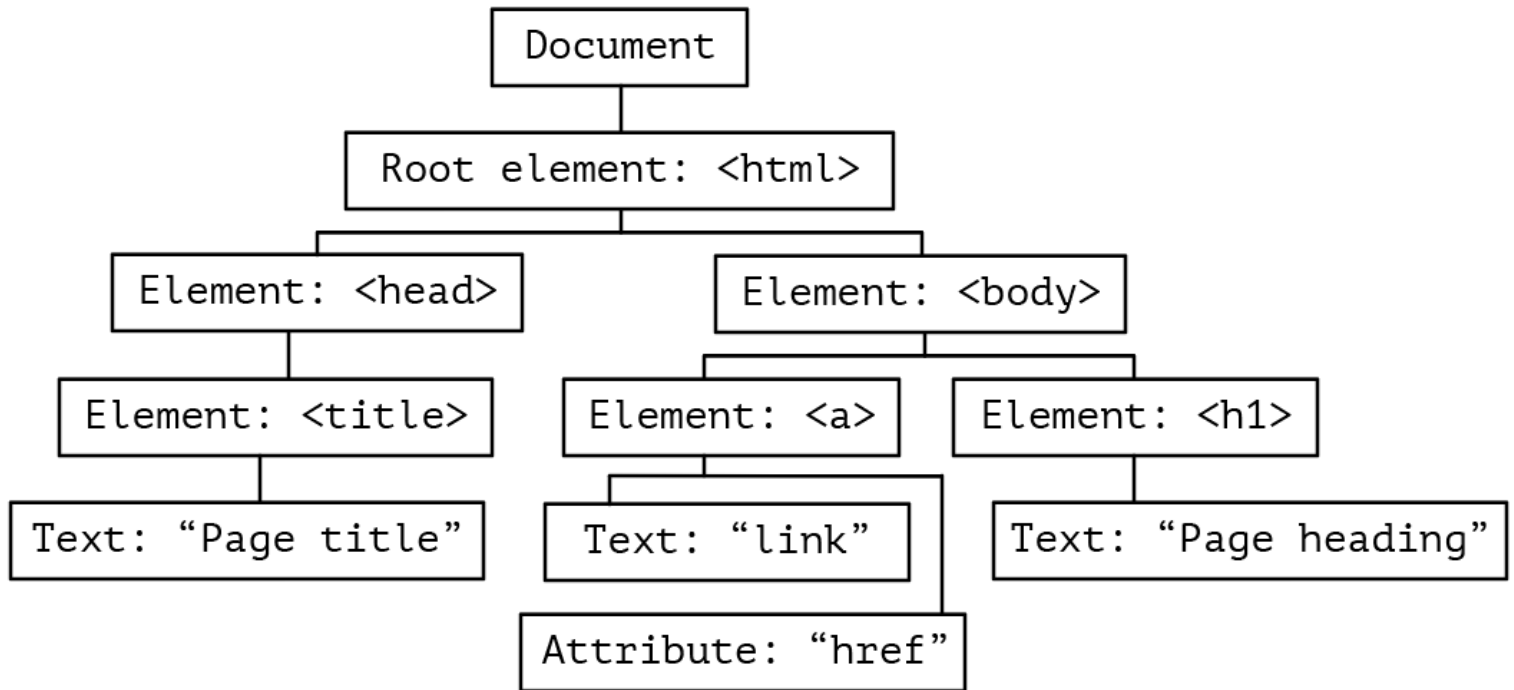


Figure 2.0: How a web page is structured in the DOM

Now let's access a part of the DOM and retrieve information from a page. We'll do this by interacting with the Javascript Console. At the greater than sign > type window object and press return. The window property of the DOM includes all the properties and functions that are available to our code every time a web page loads. Explore the window using the drop-down triangles to see all the properties and functions it contains.. The DOM, like JSON objects, is a hierarchically-organized data structure, where child properties are accessed using a dot. If you want to see the property of a page like the title (the title property of the document of the window) you can type window.document.title and press return. For information, see [dot notation](#) in the MDN reference.

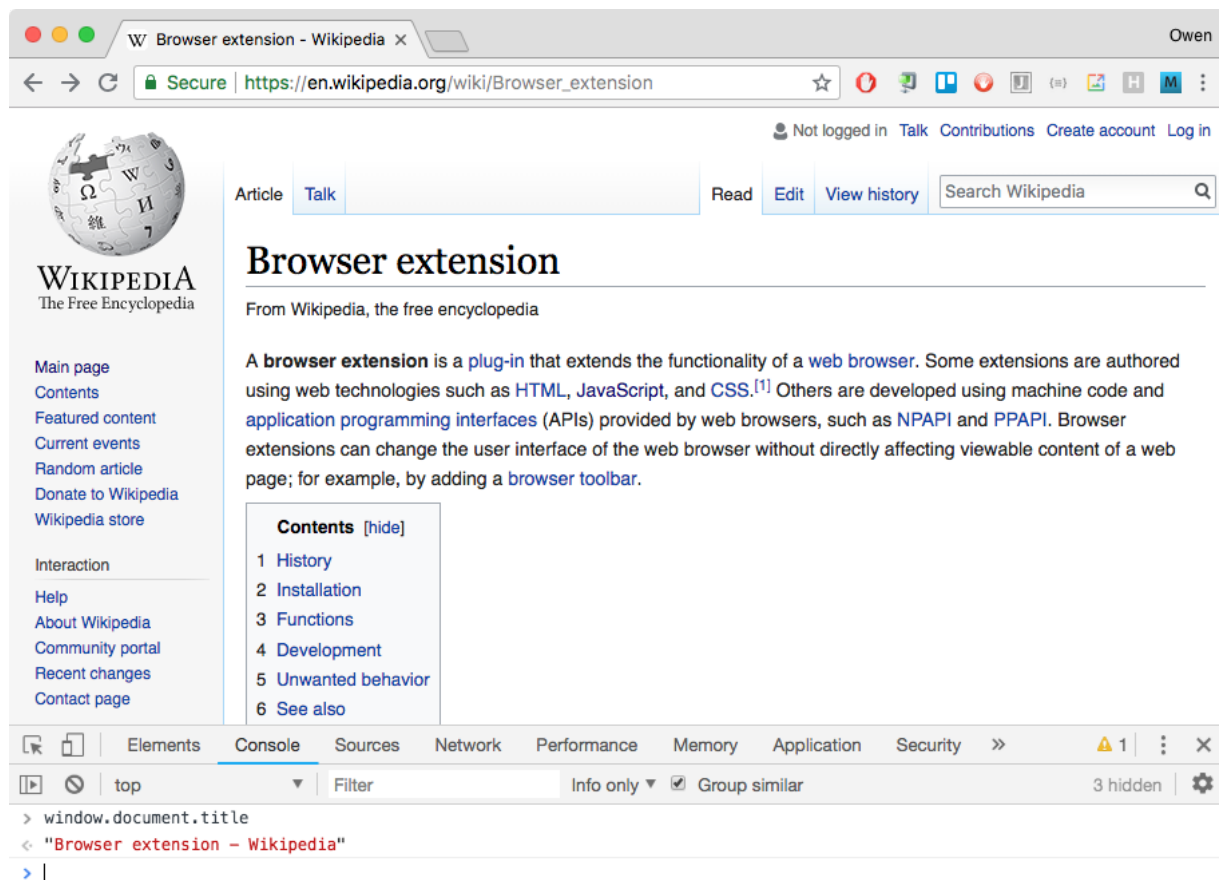


Figure 2.1

Let's make use of this in our browser extension. Starting with your code from Part 1, or the finished code inside the `/tutorial/1/` folder, we'll add a Javascript library called **jQuery** to make coding easier. A code library is a set of resources written by others that you can incorporate into your own code to add functionality. We have supplied all the scripts you need, including this one, in the assets folder you downloaded at `assets/libs/jquery.min.js`. Add it to your `manifest.json` file, just above the line that says `content.js`. Make sure to separate the two lines with a comma. Do you know why we add it *before* we include the `content.js` file?

```
"content_scripts": [{
  "matches": ["<all_urls>"],
  "js": [
    "assets/libs/jquery.min.js",
    "content.js"
  ],
  "run_at": "document_end"
}]
```

Figure 2.2

If you thought that we add the jQuery library *before* the content.js file so that the functions inside the jQuery library are loaded and available to the code inside content.js then you are correct!

The first new function we'll use in our Javascript code is **jQuery's .text() method**, which can retrieve or set the content of text elements in the DOM. Edit your content.js file to look like the following. On the first line we store the text content of the page's title in a variable, and on the second line we log that information to the console.

```
var pageTitle = $("title").text();
console.log( "The title of the page is " + pageTitle );
```

Figure 2.3

Save your work, reload the extension, and then browse the web with your Javascript Console open to see it report the page title on each new page you visit.

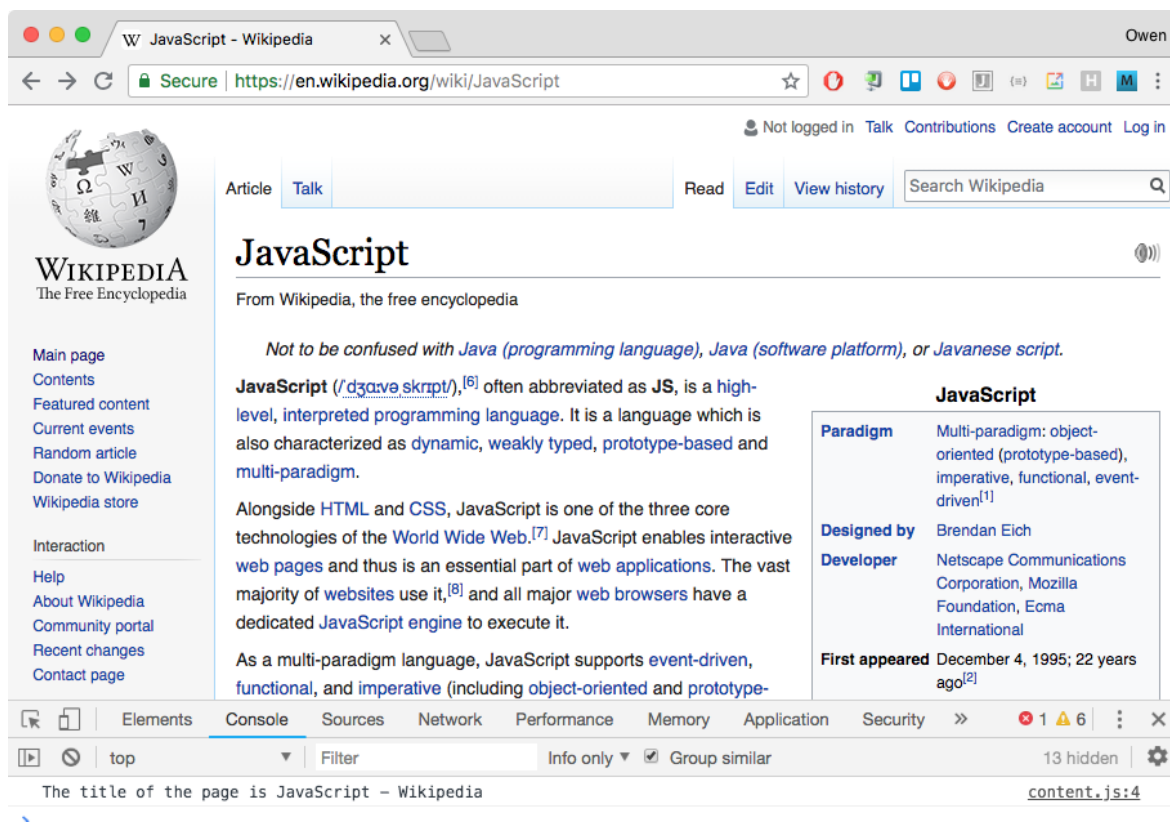


Figure 2.4

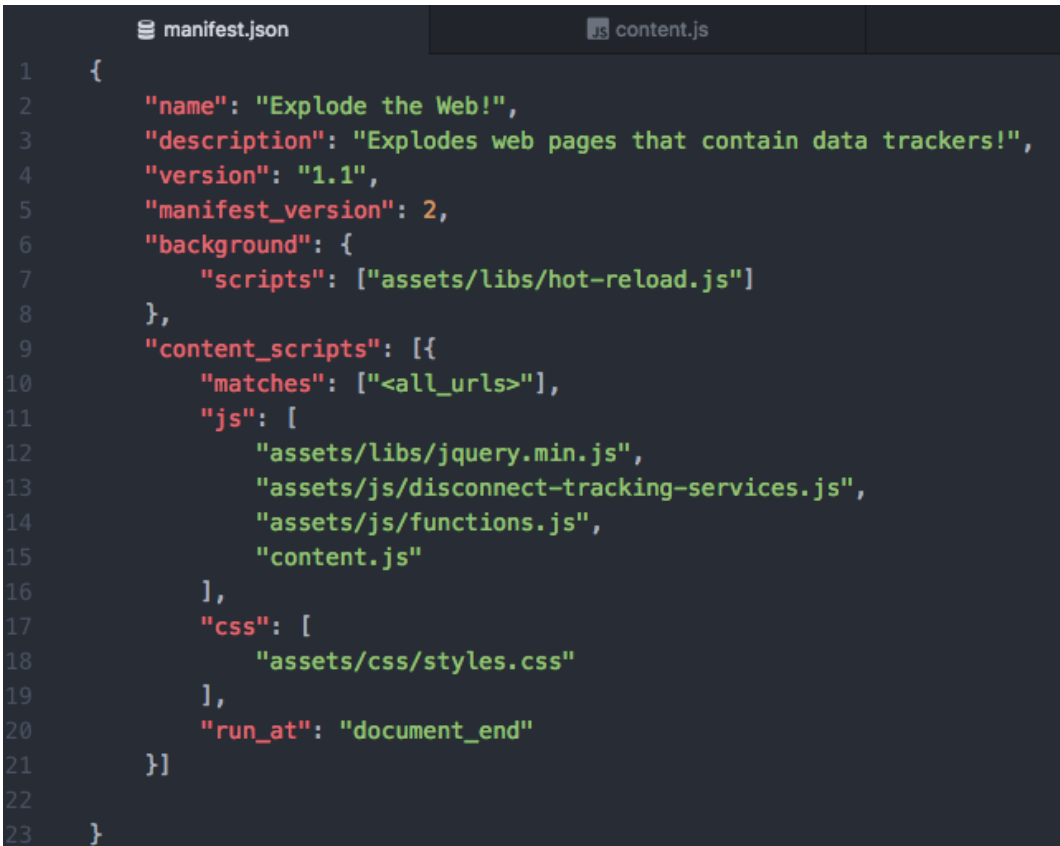
So far we have learned the basics of making and testing a browser extension, and how to access data from web pages using the DOM. As we continue we'll move a little faster so revisit the W3Schools HTML, CSS, and Javascript tutorials as needed.

Part 3: Tracking the trackers

In part 3, we're going to build on what we've learned and update our extension to a basic tracker reporter. If any trackers exist on the page, we'll display a small message in the browser to show the user what we found.

First, let's update `manifest.json` to add additional functions to our extension. Add the following code importing scripts from the `assets` folder so your manifest looks like Figure 3.0:

1. **hot-reload.js** - A **background script** that reloads our extension every time we save it. This will allow us to skip the step where we click refresh at `chrome://extensions` when we want to test our work.
2. After the line where we load jQuery in **content_scripts**, add a reference to **disconnect-tracking-services.js**. This is a list of over 1,800 different web domains that trackers connect to in order to secretly collect your data. We've included a modified version of the **list** that the excellent **Disconnect browser extension** uses to block trackers.
3. Next, in **content_scripts**, add a reference to **functions.js**, which contains helper functions we'll need in our project.
4. Finally, add a **css** property in the manifest to include **styles.css** to help to make our reporting look a little nicer.



```

1  {
2    "name": "Explode the Web!",
3    "description": "Explodes web pages that contain data trackers!",
4    "version": "1.1",
5    "manifest_version": 2,
6    "background": {
7      "scripts": ["assets/libs/hot-reload.js"]
8    },
9    "content_scripts": [{
10     "matches": ["<all_urls>"],
11     "js": [
12       "assets/libs/jquery.min.js",
13       "assets/js/disconnect-tracking-services.js",
14       "assets/js/functions.js",
15       "content.js"
16     ],
17     "css": [
18       "assets/css/styles.css"
19     ],
20     "run_at": "document_end"
21   }]
22 }
23

```

Figure 3.0

Great, so now we'll start editing **content.js** to find and report the trackers. Our **pseudo code**, or the program we'll build in "plain English," looks like this:

1. Declare the variables we need to temporarily store data from the page
2. Loop through all the script tags on the page, comparing their domains (in the "src" attribute) to the Disconnect list to see if they are known trackers
3. Store the list of trackers in a variable
4. Report the results to the user

Now that we know what we want to accomplish with our code let's actually write the program. First, (Figure 3.1) add a variable that will copy a list of all the script elements on the page, then another to hold the trackers we find.


```

1
2  // get scripts on the page
3  var scriptsFound = document.getElementsByTagName("script");
4  // store trackers we find
5  var trackersFound = [];
6

```

Figure 3.1

Now (Figure 3.2) we'll loop through all the scripts we find (line 8) and see if their **src** attribute exists in the Disconnect list (line 14). We are using one of our helper functions **extractRootDomain()** to do this. If the src domain matches a known tracker, we'll log it to the console (line 15) and store it in the **trackersFound** array (line 17). An array, denoted by the square brackets ([...]) is essentially a collection of variables in a list. Rather than holding one value (like 123 or "hello"), it can hold a list of similar, comma-separated values (like 123,456,789).

```

6
7  // loop through scripts
8  for (var i = 0, l = scriptsFound.length; i < l; i++) {
9      // if script tag has a src attribute
10     if (scriptsFound[i].src !== "") {
11         // get root domain of scripts src
12         var scriptDomain = extractRootDomain(scriptsFound[i].src);
13         // look to see if tracker root domain is in disconnectTrackingServices
14         if (disconnectTrackingServices.indexOf(scriptDomain) >= 0) {
15             console.log("!!! getTrackers()", scriptDomain);
16             // add to found list
17             trackersFound.push(scriptDomain);
18         }
19     }
20 }
21

```

Figure 3.2

You'll have to click refresh at **chrome://extensions** one last time before the **hot-reload**. **js** script starts refreshing your extension automatically. It is also possible you may have to remove and then reinstall your extension to get hot-reload to work. Go ahead and save your work and surf around the web with the Javascript console open.

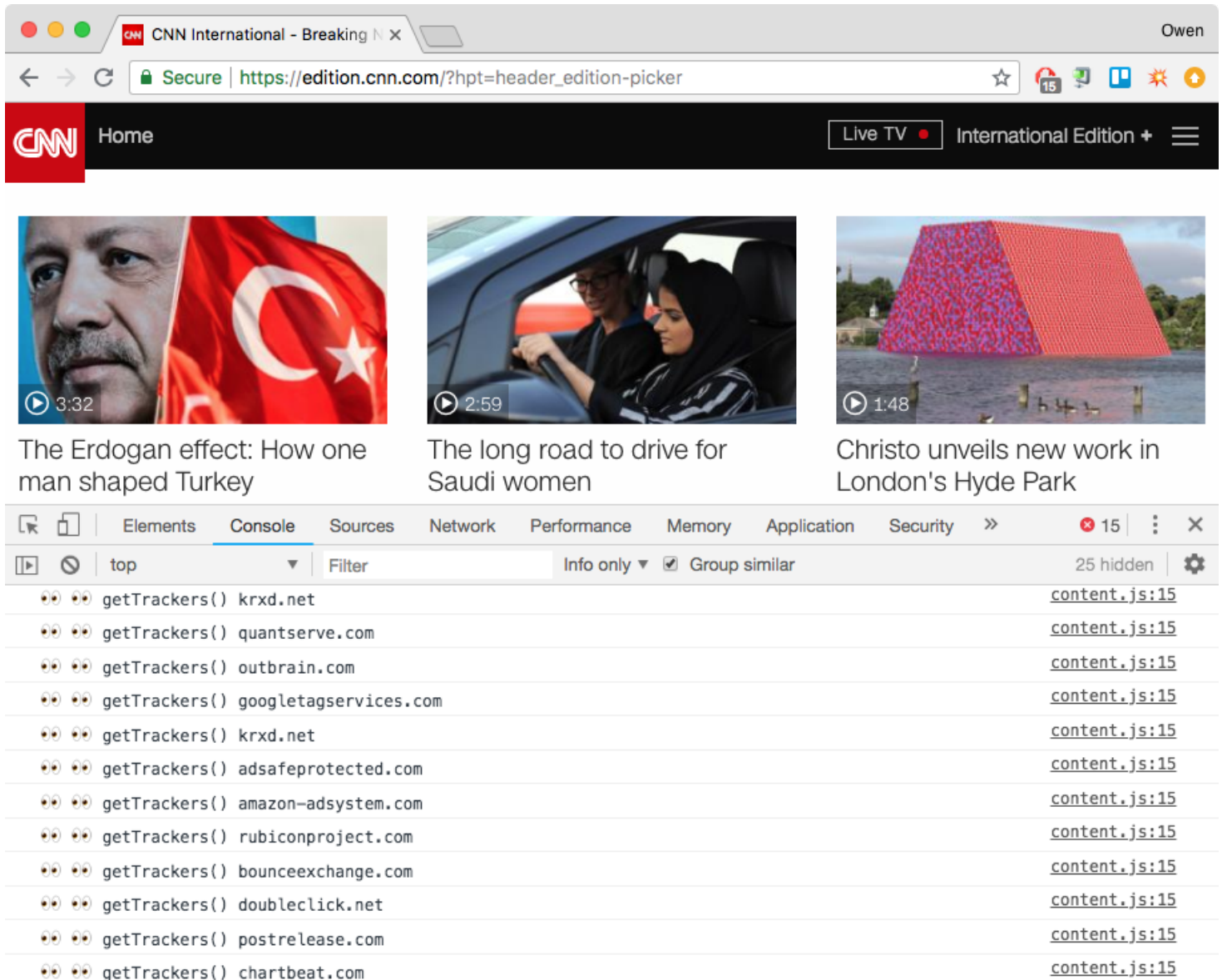


Figure 3.3

As you can see, there are massive amount of trackers out there, each of which is getting access to your data every time you load a new web page!

The screenshot shows the New York Times homepage in a browser. The browser's address bar shows the URL <https://www.nytimes.com>. The page features the newspaper's masthead, navigation links, and a large advertisement for WordPress.com. Below the ad, there are news headlines, including "U.S. Indicts 12 Russians in 2016 Hacking". A browser extension is open, displaying a list of trackers found on the page. The extension's interface includes tabs for Elements, Console, Sources, Network, Performance, Memory, Application, Security, Audits, and AdBlock. The Console tab is active, showing a list of trackers with their domains and the file location where they were found.

Tracker	Domain	File Location
getTrackers()	media.net	content.js:14
getTrackers()	amazon-adsystem.com	content.js:14
getTrackers()	bluekai.com	content.js:14
getTrackers()	media.net	content.js:14
getTrackers()	googletagmanager.com	content.js:14
getTrackers()	doubleclick.net	content.js:14

Figure 3.4

And, keep in mind that the tracker scripts we have found reveal only a tiny window into the world of corporate tracking and **behavioral targeting**.

The data in the network graph in Figure 3.5, gathered by another browser extension and tracker blocker called **Ghostery**, shows how trackers we encounter are part of a much larger network of advertisers, ad markets, and other players intent on manipulating our actions. Every time your browser loads a web page that contains trackers, even before you can see the content of a page, information about you is shared throughout these networks. This naturally increases the time it takes to load a page because many of these services are not just receiving your data, they're using it to determine the content of advertisements that will be shown to you. Further, many of the advertisers in the network are ad markets, which are holding tiny auctions with other advertisers, who are all, in real time, estimating the value of more than your attention, but the potential that they can modify your behavior. Whether the ad service is trying to sell you a car, get you to try a new service, or change how you do, or do not, vote, this is the economic backbone of the surveillance economy, made visible by our ability to view the source code of web pages.

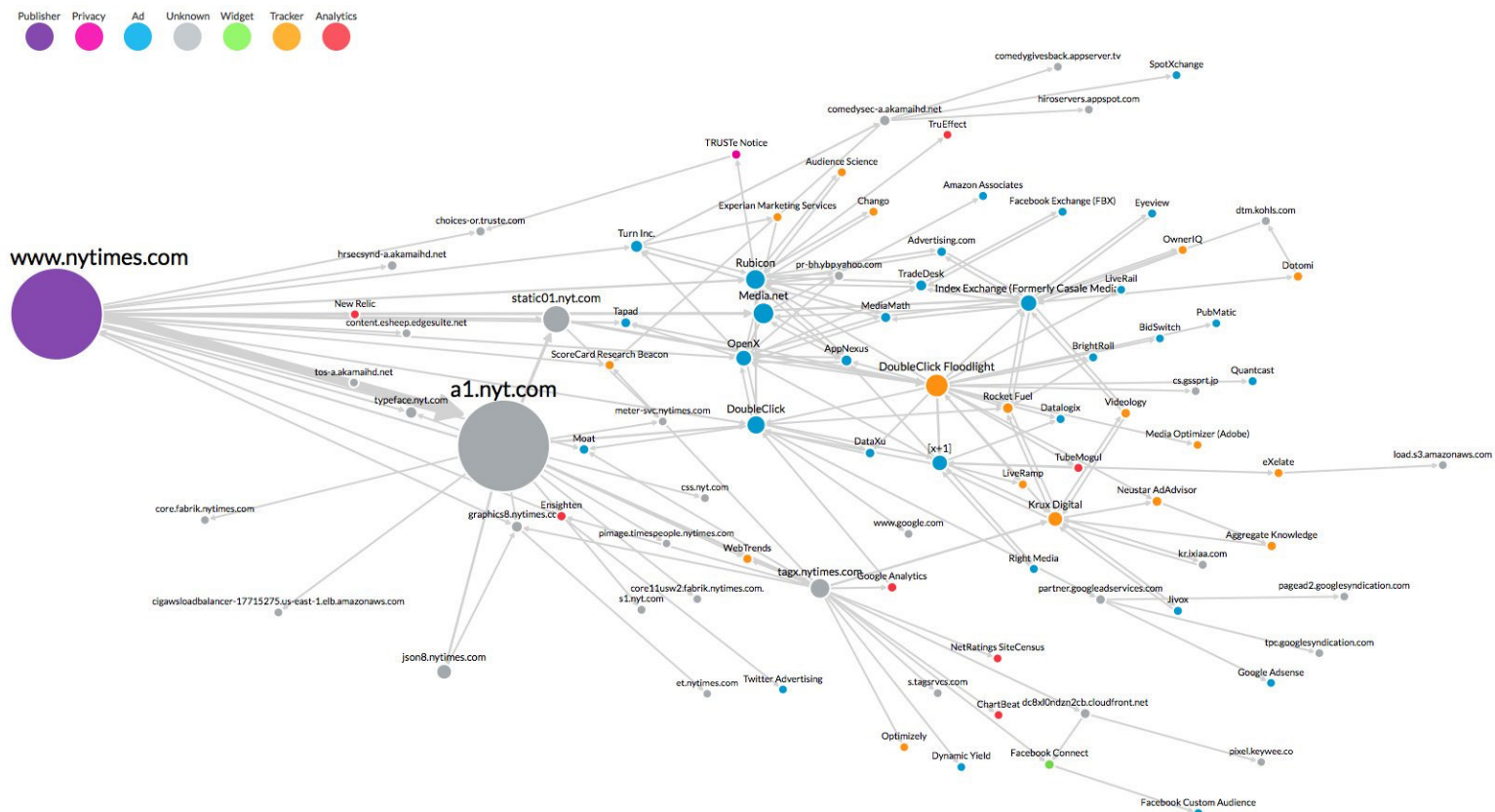


Figure 3.5

Now let's add a feature to show users how many trackers are on the page without them having to have the Javascript console open all the time. In Figure 3.6, wrapped inside an "if" statement on line 22 (a block of text that is only executed based on the stated condition), we make a new HTML element (lines 24–34) that contains the number and a list of all the trackers. Then we add a [click listener](#) with jQuery (line 36) so the list appears if they choose to click and see it.


```

21 // if there are trackers on the page
22 if (trackersFound.length > 0) {
23     // create html element to show trackers
24     var str = "<div class='explode-notification' title='click to see trackers on this page'>";
25     // display the number of trackers
26     str += "<span class='explode-number'>" + trackersFound.length + "</span>";
27     str += "<span class='explode-text'>" + " tracker(s)" + "</span>";
28     str += "</div>";
29     // create the list of trackers (hide by default)
30     str += "<div class='explode-tracker-list'>" + trackersFound.join("<br>") + "</div>";
31     // timer (hide by default)
32     str += "<div class='explode-counter'><div class='explode-counter-text'></div></div>";
33     // append html to loaded web page
34     $('body').append(str);
35     // if the user clicks on the element
36     $(document).on('click', '.explode-notification', function() {
37         // and display the list
38         $('.explode-tracker-list').toggle();
39     });
40 }

```

Figure 3.6

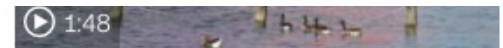
Go back and reload the last tracked page you were viewing. You should now see a report at the bottom of the page that tells you how many trackers there are. Clicking the report will reveal the names of the trackers' domains.



an effect: How one
and Turkey



The long road to drive for
Saudi women



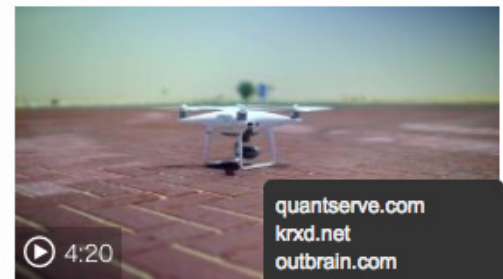
Christo unveils new work in
London's Hyde Park



Coal is driving
on



Noura in her own words



Why Dubai is all

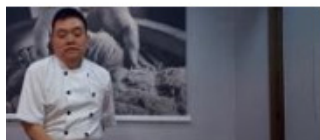


Figure 3.7



quantserve.com
kxrd.net
outbrain.com
googletagservices.com
kxrd.net
adsafeprotected.com
amazon-adsystem.com
rubiconproject.com
bounceexchange.com
doubleclick.net
postrelease.com
chartbeat.com

12 tracker(s)

Part 4: Explode the web!

This brings us to the final part of our extension-building tutorial, exploding web pages that have third-party trackers on them. You may have noticed that a lot of web pages have trackers, so we'll only explode those pages that have an exorbitant amount of tracking. Otherwise, thanks to the pervasiveness of behavioral tracking, the internet would be unusable!

Let's start by adding some new features to our **manifest.json** file. Feel free to begin with the completed section 3 that we provided.

```

1  {
2    "name": "Explode Tracked Web Pages!",
3    "description": "Explode web pages that contain data trackers!",
4    "version": "1.1",
5    "manifest_version": 2,
6    "icons": {
7      "48": "assets/img/explode-icon16.png",
8      "48": "assets/img/explode-icon48.png",
9      "128": "assets/img/explode-icon128.png"
10   },
11   "browser_action": {
12     "default_icon": {
13       "16": "assets/img/explode-icon16.png",
14       "48": "assets/img/explode-icon48.png"
15     },
16     "default_title": "Explode Tracked Web Pages!",
17     "default_popup": "assets/pages/popup.html"
18   },
19   "background": {
20     "scripts": ["assets/libs/hot-reload.js"]
21   },
22   "content_scripts": [{
23     "matches": ["<all_urls>"],
24     "js": [
25       "assets/libs/jquery.min.js",
26       "assets/libs/anime.min.js",
27       "assets/js/disconnect-tracking-services.js",
28       "assets/js/functions.js",
29       "assets/js/keys.js",
30       "assets/js/explode.js",
31       "content.js"
32     ],
33     "css": [
34       "assets/css/styles.css"
35     ],
36     "run_at": "document_end"
37   }],
38   "permissions": [
39     "*/**/*",
40     "activeTab",
41     "tabs"
42   ],
43   "web_accessible_resources": ["*.svg", "*.mp3"]
44 }
```

Figure 4.0

Here are the details of the changes you see in figure 4.0:

1. The **icons** properties (lines 6–10) add an image next to our title on the **chrome://extensions** page (Figure 4.1).

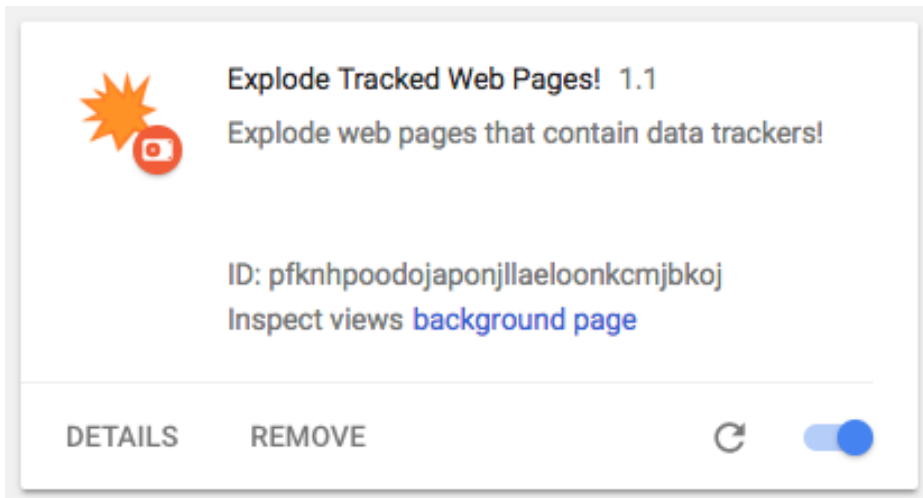


Figure 4.1

2. The **browser_action** (lines 11–18) adds an icon and “popup” menu the user can see at the top right of their browser window (Figure 4.2). When they click it it will show the contents of this page: **assets/pages/popup.html**

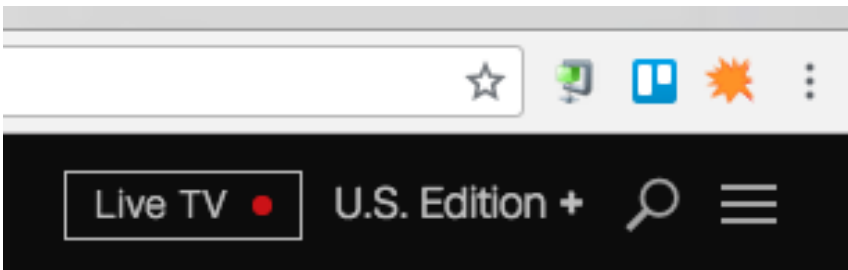


Figure 4.2

3. We import **assets/libs/Anime.js**, (line 26) an animation library, to animate the “explosion.”
4. We import **assets/js/keys.js** (line 29) with code to test our explosion.
5. We import **assets/js/explode.js** (line 30) which contains the **explodeTheWeb()** function.
6. The **permissions** property (lines 38–42) allows our extension to run on any web page.
7. The **web_accessible_resources** property (line 43) allows us to reference external files at runtime, in this case an image (SVG) and a sound (MP3) of the explosion.

Check out the file `.keydown()` function in `keys.js` (Figure 4.3.) and you will see the `explodeTheWeb()` function we are going to call to explode the page.

```

11  /**
12   * If keydown detected
13   */
14  $(document.body).keydown(function(event) {
15      if (event.keyCode == 69) // e
16          keys.e = true;
17      else if (event.keyCode == 192) // ~
18          keys.tilda = true;
19      if (keys.e && keys.tilda) {
20          console.log("e + tilda ~");
21          explodeThePage();
22      }
23      //console.log(event.keyCode, keys);
24  });

```

Figure 4.3

Next let's test the "explosion" function using a key press. Save the `manifest.json` file, reload your test page, and press the ~ (tilda) key (shift + `) and the e keys simultaneously with the console open. You should see the page explode and the extension report that it received the key press (Figure 4.3).

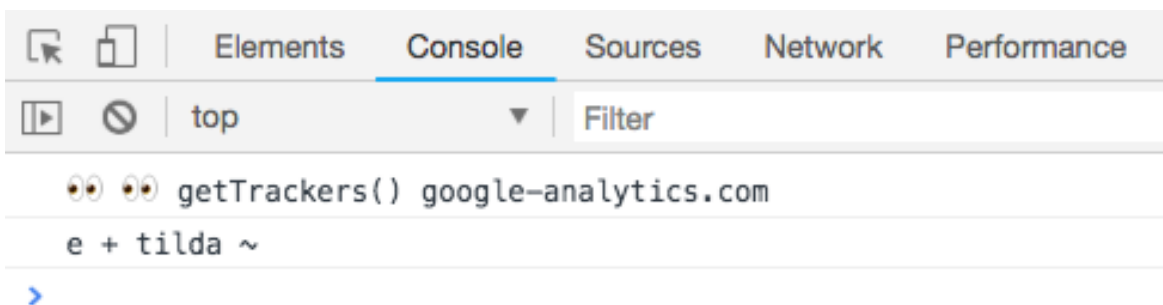


Figure 4.4

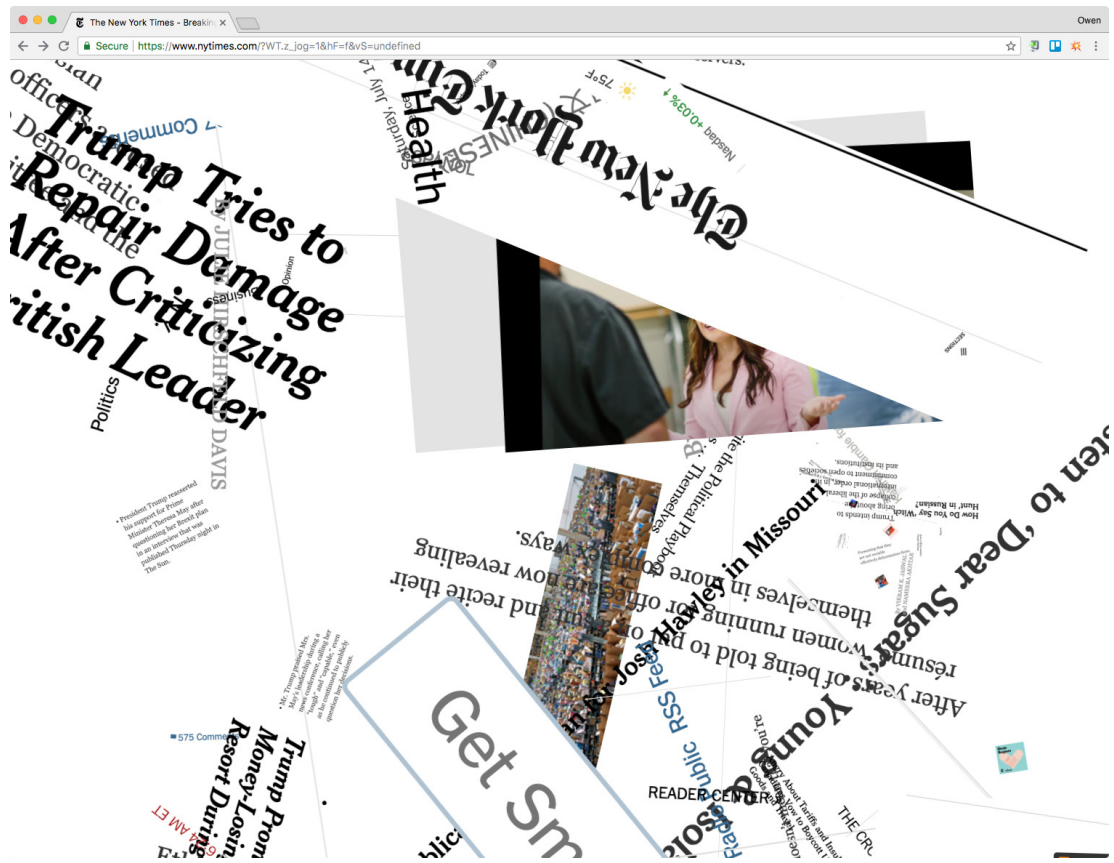


Figure 4.5

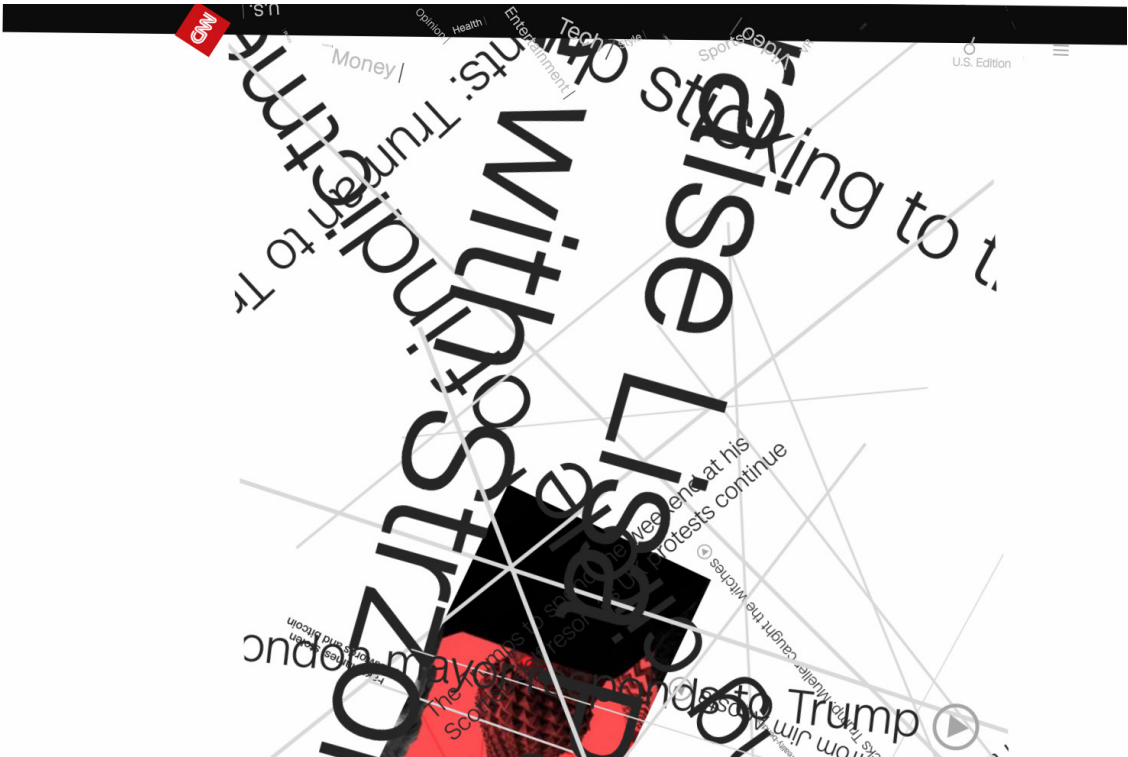


Figure 4.6

Fireworks aside, unless you work in a particular industry, or have seen combat, you've only witnessed fiery explosions as a series of still images—a moving picture. In films, the visual effect resulting from the chaos of shattered materials and ignited fuel is rendered by computer code. To simulate this disorder and reorganization of physical materials on screen, the computer assigns new properties like location, size, rotation, etc. to the individual particles. Seen at speed, together, it is the widely popular spectacle that fills in the gaps where story or dialogue doesn't.

Your computer's GUI, or graphical user interface, is also an image. When you browse the internet, the browser renders an image of the page you visit in the GUI with millions of pixels. When you move your mouse an image of an arrow moves on the screen corresponding to the X and Y values detected by the hardware. Click a hyperlink on the internet, the computer software is actually detecting the current X and Y position of that arrow image and sending a message that a click happened in that space to the browser, which then loads the new page, and another image is rendered with areas designated as "clickable."

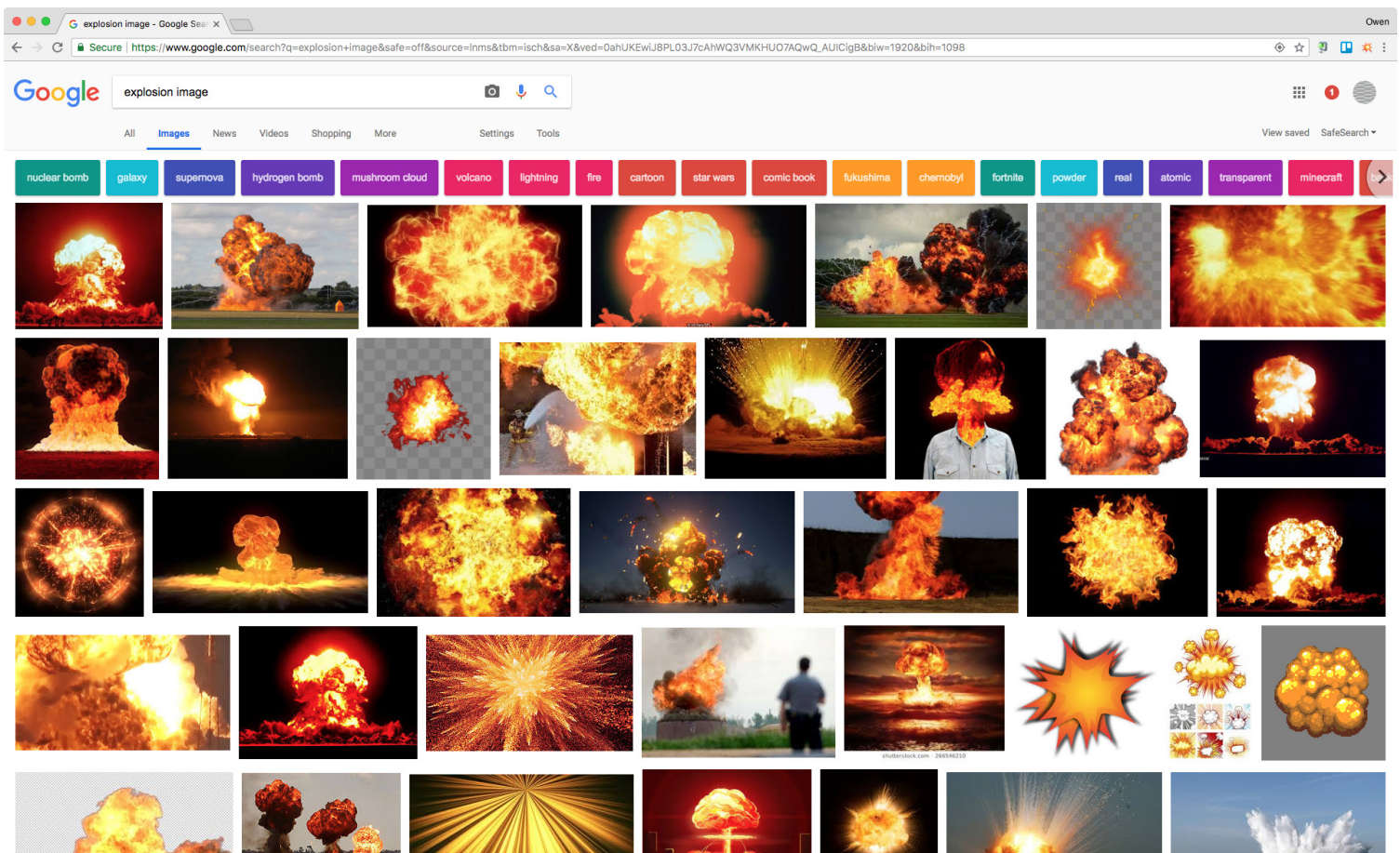


Figure 4.7

Corresponding to how explosions are simulated in films, if you examine the contents of `assets/js/explode.js` (Figure 4.10) you will see that our `explodeThePage()` function:

1. Identifies a collection of HTML nodes (or particles) (line 7)
2. Loops through the HTML nodes and excludes certain classes (like the class for our notification). (line 13)
3. Then, using the Anime.js animation library, it targets the nodes and changes the rotation, scale (increase and decrease in size), and translation (change in position) properties of each element to new, randomly determined, values.
4. Finally, we play the explosion.mp3 audio file. (line 46–47)

```

1  /**
2   * Explode the page
3   */
4  function explodeThePage() {
5
6      // all possible html5 nodes
7      let nodes = ['a', 'a[href]', 'b', 'blockquote', 'br', 'button', 'canvas', 'code', 'dd', 'dl', 'dt',
8                  'em', 'embed', 'footer', 'frame', 'form', 'header', 'h1', 'h2', 'h3', 'h4', 'h5', 'h6', 'hr',
9                  'iframe', 'img', 'input', 'label', 'nav', 'ol', 'ul', 'li', 'option', 'p', 'pre', 'section', 'span',
10                 'strong', 'sup', 'svg', 'table', 'tr', 'td', 'th', 'tbody', 'thead', 'textarea', 'text', 'u', 'video'
11             ];
12      // add any exclusions
13      for (let i = 0, l = nodes.length; i < l; i++) {
14          nodes[i] = nodes[i] + ':not(.tally):not([class!="explode-"])';
15      }
16      // run animation
17      anime({
18          targets: document.querySelectorAll(nodes.toString()),
19          rotate: function() {
20              return Math.random() * randomRange(-360, 360);
21          },
22          translateX: function() {
23              return Math.random() * randomRange(-50, 50);
24          },
25          translateY: function() {
26              return Math.random() * randomRange(-50, 50);
27          },
28          scale: function() {
29              return Math.random() * 2;
30          }
31      });
32      // explode main nodes just a little
33      anime({
34          targets: document.querySelectorAll('div:not(.tally):not([class!="explode-"])'),
35          rotate: function() {
36              return Math.random() * 2;
37          },
38          translateX: function() {
39              return Math.random() * 40;
40          },
41          translateY: function() {
42              return Math.random() * 40;
43          }
44      });
45      // add audio to play explosion sound
46      var audio = new Audio(chrome.extension.getURL('assets/sounds/explode.mp3'));
47      audio.play();
48  }

```

Figure 4.8

Voila! An explosion. However, unlike the “pyro-techniques” of a movie explosion, as the creator of this code, you can keep pressing the test keys to see the page explode and re-explode as many times as you like. Further, you can connect the explosion to other events, like the presence of trackers on a page. Let’s do that now.

In `content.js` add four new variables (Figure 4.9) to the top of the page.



```

1  // get scripts on the page
2  var scriptsFound = document.getElementsByTagName("script");
3  // store trackers we find
4  var trackersFound = [];
5  // number of trackers allowed
6  var trackersFoundLimit = 6;
7  // track time to count down until explosion
8  var timeUntilExplode = 0;
9  // explosion timer duration
10 var timeUntilExplodeDuration = 10;
11 // whether to explode or not
12 var explodePaused = false;
13

```

Figure 4.9

Then, add the code in Figures 4.10 and 4.11 to the end of `content.js`. There’s a lot of code here so you can copy from the finished file in **/tutorial/4** if you want to speed things up. Everything is commented well so you can read the code to see what is happening in detail. Here is an overview:

1. Figure 4.9: Add variables to set a limit for the number of trackers allowed, to keep track of the countdown timer, its limit, and whether or not we’ve already exploded the page (lines 5–12)
2. Figure 4.10: Then, if the number of trackers is over the limit (line 51) we start a timer (line 55) and show an icon on the screen (line 57–60) to let the user know we’re going to explode. We also add a click listener (lines 61–85) so the user can pause, restart, or reset the page after exploded.
3. Figure 4.11: On line 89 is the function we call every second to update the timer. If it reaches zero we explode the page. The function on line 110 is to update the tile on the mouseover so the user knows the status.

```
49
50 // if there are more trackers on the page than the limit
51 if (trackersFound.length >= trackersFoundLimit) {
52     // set time
53     timeUntilExplode = timeUntilExplodeDuration;
54     // start timer
55     var interval = setInterval(updateTimer, 1000);
56     // add explosion image
57     $('.explode-counter').css({
58         "background": "url(" + chrome.extension.getURL("assets/img/explode-ui.svg") + ")",
59         "cursor": "pointer"
60     });
61     // add click listener if user wants to cancel
62     $(document).on('click', '.explode-counter', function() {
63         // if time left
64         if (timeUntilExplode > 0) {
65             // and explode isn't already paused
66             if (!explodePaused) {
67                 // set it paused
68                 explodePaused = true;
69                 // clear timer
70                 clearInterval(interval);
71                 // update title
72                 showExplodeTitle("Click to start timer");
73             } else {
74                 // set it not paused
75                 explodePaused = false;
76                 // start timer
77                 interval = setInterval(updateTimer, 1000);
78                 // update title
79                 showExplodeTitle("Click to cancel");
80             }
81         } else {
82             // reload page
83             location.reload();
84         }
85     });
86 }
87
```

Figure 4.10

```
87
88 // function called every second from setInterval()
89 function updateTimer() {
90     // if time is left
91     if (timeUntilExplode > 0) {
92         // subtract by 1
93         timeUntilExplode--;
94         // show result in browser
95         $('.explode-counter-text').html(timeUntilExplode);
96         // update title text
97         showExplodeTitle("Click to cancel");
98     }
99     // if no time is left
100    else {
101        // clear interval and explode page
102        clearInterval(interval);
103        // explode the page
104        explodeThePage();
105        // update title text
106        showExplodeTitle("Click reset to reset page");
107    }
108 }
109
110 function showExplodeTitle(status) {
111     // change title text
112     var attr = "this page will explode in [" + timeUntilExplode + "] seconds. " + status;
113     $('.explode-counter').attr('title', attr);
114 }
115
```

Figure 4.11

Save your work and surf the web to see this in action!

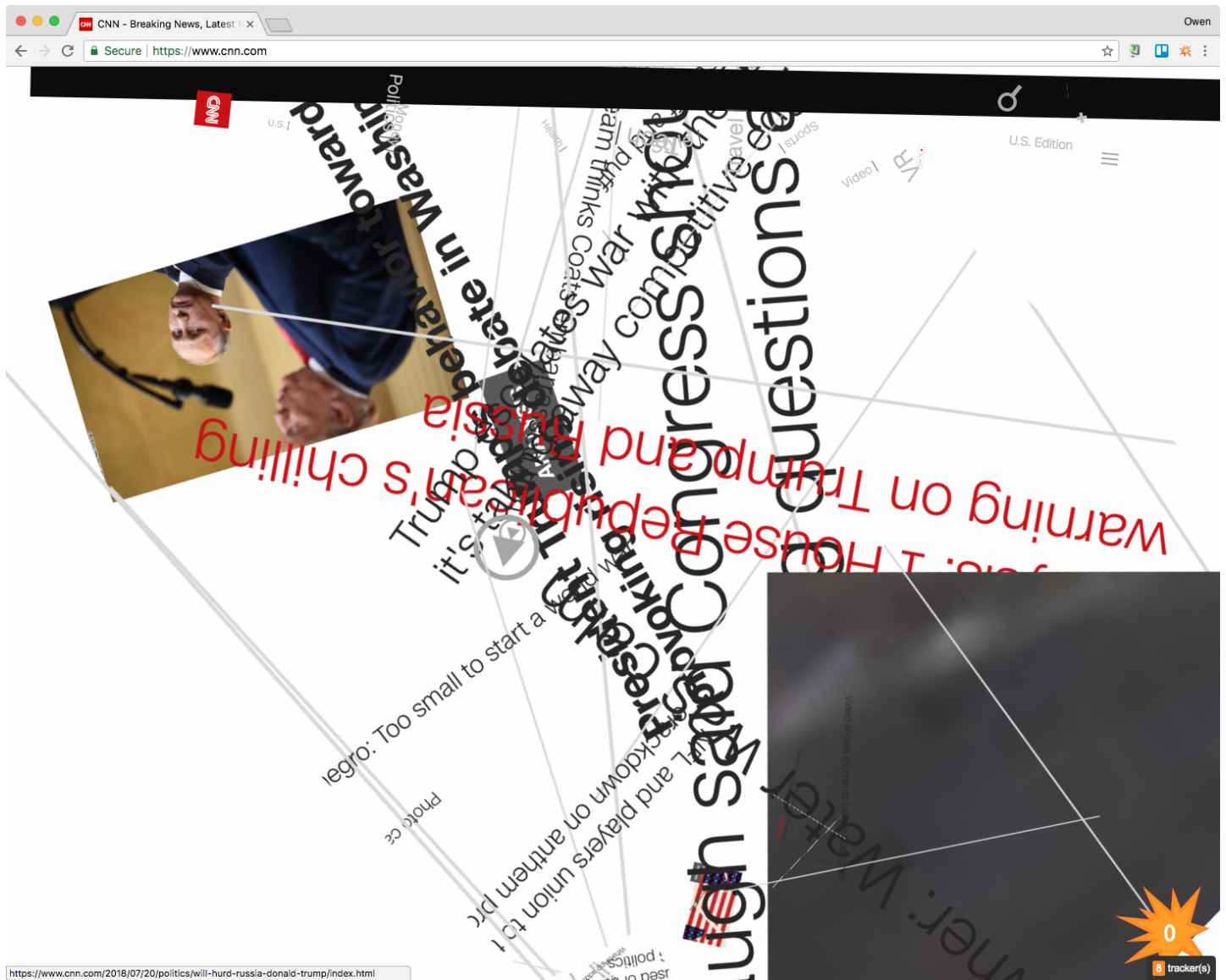


Figure 4.12

You can see the bulk of the work in making our extension was writing Javascript code. And like many coded ideas, ours is just one of many ways it could have been executed. Now that we've shown you how to build a cross-browser extension, you can remix ours, or come up with a new idea and create your own. Once you do, publish it [online](#) and [let us know!](#)

Conclusion

In conclusion, we've seen examples of browser extensions, how they work, and how to build one. We've also learned that trackers are everywhere, but thanks to the open nature of the web, there are ways to deny them your data. And, as this Explode the Web! extension, and our new browser game, Tally (due to launch in April 2020!!!) will show, there are many ways to resist the surveillance economy that are both creative and practical.

References (Chicago)

"Choosing the right filterlist." Adblock Plus. Accessed July 23, 2018. https://adblockplus.org/en/getting_started#subscription

Atwood, Jeff. "The Power of 'View Source'." Coding Horror (blog). August 17, 2006. Accessed July 23, 2018. <https://blog.codinghorror.com/the-power-of-view-source/>

"US Ad Blocking User Penetration, 2014-2018 (% of internet users)." eMarketer. February 22, 2017. Accessed July 23, 2018. <https://www.emarketer.com/Chart/US-Ad-Blocking-User-Penetration-2014-2018-of-internet-users/204561>

Ghostery. 2016. Why #Google AMP Is So Much Faster in Three Dramatic Charts [#ghostery #trackermap @adage](http://mygho.st/LQ). <https://twitter.com/Ghostery/status/702935141920993280>

Madden, Mary and Lee Rainie. "Americans' Attitudes About Privacy, Security and Surveillance." Pew Research Center. May 20, 2015. Accessed July 23, 2018. <http://www.pewinternet.org/2015/05/20/americans-attitudes-about-privacy-security-and-surveillance/>

Stallman, Richard. "Why Open Source misses the point of Free Software." November 18, 2016. Accessed July 23, 2018. <http://www.gnu.org/philosophy/open-source-misses-the-point.html>

Surman, Mark and Jason Diceman. "Choosing Open Source: A decision-making guide for civil society organizations." January 2004. Accessed July 23, 2018. <https://marksurman.commons.ca/publications/choosing-open-source-a-decision-making-guide-for-civil-society-organizations/full-text/>

References (Chicago)

Tee, Gillian. "Google Alarm' plug-in tries to wake the world up to privacy issues." CNN. August 6, 2010. Accessed July 23, 2018. <http://www.cnn.com/2010/TECH/web/08/06/google.alarm/index.html>

Turow, Joseph, Michael Hennessey and Nora Draper. "The Tradeoff Fallacy: How Marketers Are Misrepresenting American Consumers And Opening Them Up to Exploitation." The Annenberg School for Communication at the University of Pennsylvania. June 2015. Accessed July 23, 2018. https://www.asc.upenn.edu/sites/default/files/TradeoffFallacy_1.pdf

Voon, Claire. "Cat Art from the Met Museum Makes the Purrfect Browser Plug-in." Hyperallergic. July 16, 2015. Accessed July 23, 2018. <https://hyperallergic.com/222538/cat-art-from-the-met-museum-makes-the-purrfect-browser-plug-in/>

Additional Recommended Resources Not Referenced in Paper (Chicago)

Singer, Natasha. "Sharing Data, but Not Happily." New York Times. June 5, 2015. Accessed July 23, 2018. <https://www.nytimes.com/2015/06/05/technology/consumers-conflict-ed-over-data-mining-policies-report-finds.html>

Simanowski, Roberto. "Data Love : The Seduction and Betrayal of Digital Technologies." La Vergne: Columbia University Press, 2016.