

# Find & Replace Extension for Text Editing

RegEx Search & Replace Extension for Chrome and Firefox browsers.

**Project Website:** <https://find-and-replace-f6588.firebaseio.com/>

**Chrome Web Store Link:** <https://chrome.google.com/webstore/detail/find-replace-for-text-edi/jajhdmnpicpbpnlpejbgmpijgmoknnl>

**Firefox Add-ons Store Link:** <https://addons.mozilla.org/en-US/firefox/addon/find-replace-for-text-editing/>

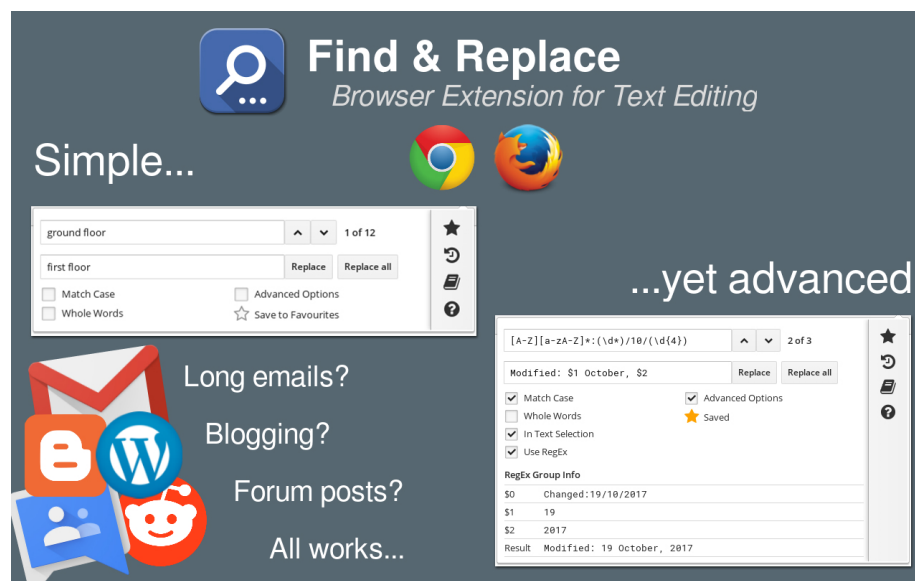


Figure 1: Find & Replace Browser Extension - Web Store Screenshot A

## Project Specification

### Principal Goal

Develop a Chrome/Firefox browser plug-in to allow Search & Replace with regular expressions in text input fields of web pages.

### Description

The project will develop a Chrome browser plug-in that would provide a search & replace functionality in text input fields of web pages. An important part of the project is to provide support for the use of popular regular expression

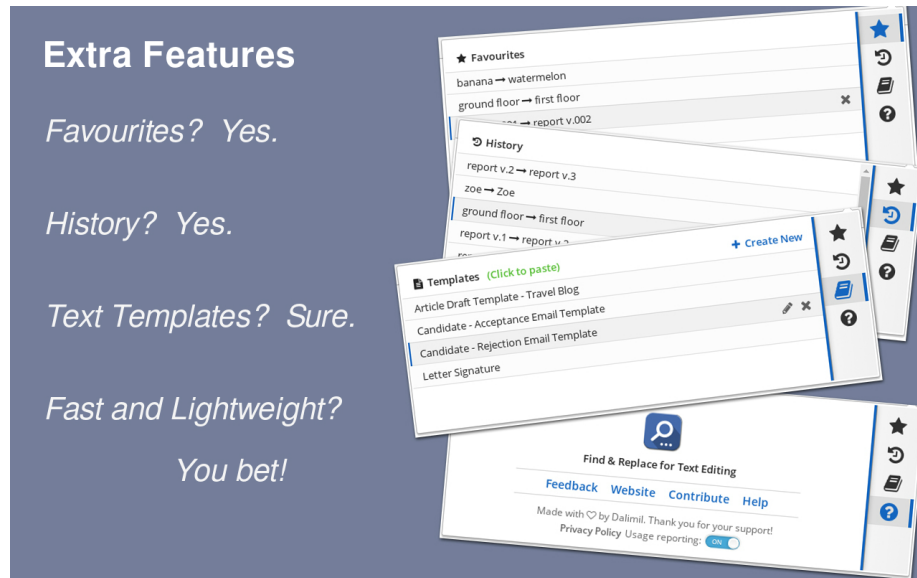


Figure 2: Find & Replace Browser Extension - Web Store Screenshot B

constructs, a functionality which is missing in extensions available today. Other desirable features include:

- Ability to restrict search & replace to the current input field (i.e. where the cursor is) or run through all input fields on the page.
- Restrict search & replace to the selected text only
- Support for templates (i.e. a small “library” of commonly occurring patterns to search-and-replace with one click).

In addition to the software development work, it is expected that the student will do at least one round (ideally two) of user testing, collecting the feedback, and improving the tool based on the responses & bug reports. An ideal way to do this is by having a stable beta version of the plugin available on the Chrome Web Store well before the thesis is due.

## Motivation ~ Why would this be useful?

- Weird Characters In Blog Entries – This sometimes happens in WordPress immediately after an upgrade. You may end with weird characters such as “â€”, “â€™” or “??”. Imagine the agony of having to replace each occurrence manually.
- Renaming A Word – Sometimes you may want to rename a word that occurs several times in an entire blog post, forum post or email. For example, you could decide to use a synonym which sounds better or brings

out the meaning more clearly.

- Fixing A Typo – You may also realize that you have been spelling a word the wrong way and it happens to reoccur several times in your blog post, forum post or email.

Search and replace would be extremely useful on these websites: WordPress, Gmail, Hotmail, Blogger, Facebook, any forums, online blogging platforms, and web email clients.

You could always paste the content into your word processor, fix it and paste it back into the Chrome input field but that is a lengthy and time consuming process, and it will most likely lose all your formatting.

With more advanced features, one could also see the history of previous find & replace operations, save favourite queries and also define paste-only templates (for instance various email templates).

## What already exists?

Web browsers support standard search functionality for any text on a page but no browsers have the find & replace functionality (Google Chrome Help Forum post asking for the feature: <https://productforums.google.com/forum/#!topic/chrome/Y4UORlpdYfo>)

*Google Chrome browser:*

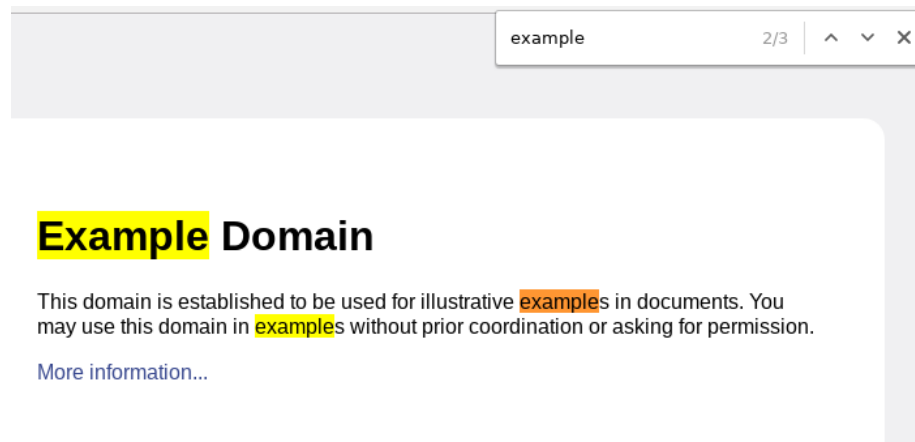


Figure 3: chrome-find

*Mozilla Firefox browser:*

There have been several attempts to implement this functionality via an extension. Most of them either don't work, are missing functionality (particularly regex), are limited to certain websites, or are unintuitive and hard to use in general.

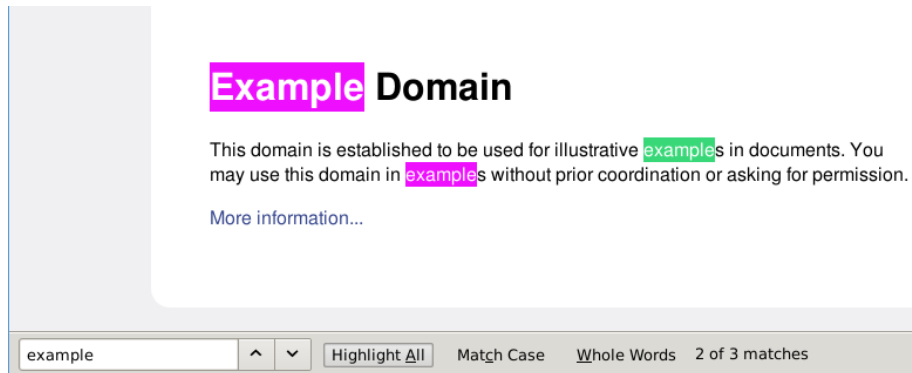


Figure 4: firefox-find

## Chrome

- <https://chrome.google.com/webstore/detail/search-and-replace/bldchfkhnkoimaciljpilanilmbnofo>
- This is the dominant one with over 52,000 users (as of Dec 2017)
- It has only 3.1/5 stars and many reviews are complaining it doesn't work or that it messes up other content of the current page
- It has usability issues (control checkboxes partially hidden) and it doesn't work in many places such as Blogger, Facebook,...
- This is clearly a very bad and simplistic solution that is mostly broken but benefits from a larger userbase because it came early (appeared in 2013)
- <https://chrome.google.com/webstore/detail/find-replace/cfjmfciolkikfodjfdmdpdpmpfbjdofek>
- This extension requires copy-pasting your desired text into the provided box (plain-text only, so formatting is lost).
- It has almost 1000 users (as of Dec 2017), but it has been around since 2013
- It is unable to perform the search and replace in browser
- This is no better than copy pasting text into a text editor and performing the operation there
- <https://chrome.google.com/webstore/detail/findr/bidnaaogcagbdidehabnjfedabckhdgc>
- It has almost 2000 users and 3.9/5 stars. It has been around since April 2016.
- This extension is used for replacing any HTML text in the page, rather than text in input fields - for this purpose it seems to work, but when one tries to use it only for text input fields, things break and the extension stops working (highlighting and match indicator both disappear and replace button stops working)
- It also requires permission to "Read and change all your data on the websites that you visit", which might be a privacy issue (the extension can read everything even when the user isn't using it)
- <https://chrome.google.com/webstore/detail/easy-replace/ojoejfegihohnkjlfoonbnailkohkce>

- It has over 3000 users but only 2.6/5 stars
- Most of them report it doesn't work because it only focuses on plain-text textareas and completely ignores more advanced editable HTML elements that most sites use these days

## Firefox

- <https://addons.mozilla.org/en-US/firefox/addon/find-and-replace-for-firefox>
- Old add-on, not compatible with the latest Firefox 57.
- It has almost 3000 users but only 3.5/5 stars
- It's not working for most users, has almost no options (no RegEx, no highlighting, etc.)
- It was last updated in 2012 and is not maintained
- <https://addons.mozilla.org/en-US/firefox/addon/foxreplace/>
- This extension provides different functionality - it asks the user to predefine a list of substitutions and then it automatically applies them globally across the text in newly loaded web pages
- FoxReplace has over 7000 users (Dec 2017) but targets different audience

## Other browsers

Approximate values for current market share for desktop are ([https://en.wikipedia.org/wiki/Usage\\_share\\_of\\_v](https://en.wikipedia.org/wiki/Usage_share_of_v) Chrome 55%, Safari 15%, Firefox 10%, Edge < 5%)

This project will focus on Chrome and Firefox, which mostly follow the same Extension API. Safari, although widely used, has its own extension API and is in general more involved as it requires dealing with Apple's developer libraries and licenses.

## Development

### Naming and SEO

Based on the research mentioned above, these extension names already exist: 'Easy Replace', 'Search and Replace', 'FindR', 'Find Replace', 'FoxReplace'. Using any of these existing names would be bad for SEO and discoverability. At the same time, we want to clearly indicate that the extension is used for input fields and editable content rather than HTML source code.

People are likely to search for browser extensions by typing in the functionality that they need. In our case, that might look something like 'find and replace in text input fields extension'. Stating the extensions purpose in its title and description should help us do better in search results. I therefore avoided any

newly-invented words and named it **Find & Replace for Text Editing**. We are not trying to trademark a new brand name, we are simply trying to match what people might search for.

## Search and Replace UI Components

- ‘Find’ input field
- ‘Replace’ input field
- **Action buttons** (see below)
- Replace (~ pressing ENTER in ‘Replace’ field)
- Replace All
- Find next (~ pressing ENTER in ‘Find’ field)
- Find previous
- Close widget
- Save to favourites
- **Options** (see below)
- Match Case (Aa)
- Use Regex (.\*)
- Whole Word (Ab|)
- In Text Selection
- ‘X of Y’ or ‘No Results’ indicator
- Regex groups indicator (for regex search only)
- **Panel buttons** (see below)
- Favourites
- History
- Templates
- Help/Info/Feedback

To some extent, we would like to follow the current standard of find & replace toolbars. Many of these can be seen in more advanced text editors:

*Android Studio:*

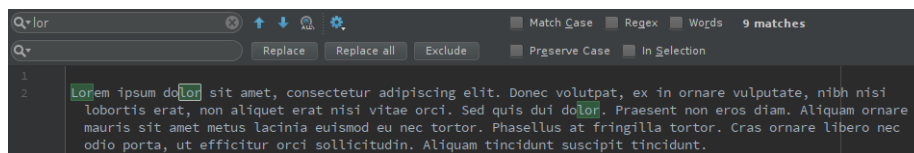


Figure 5: Android Studio

*Google Docs:*

*Visual Studio Code:*

*Sublime Text 3:*

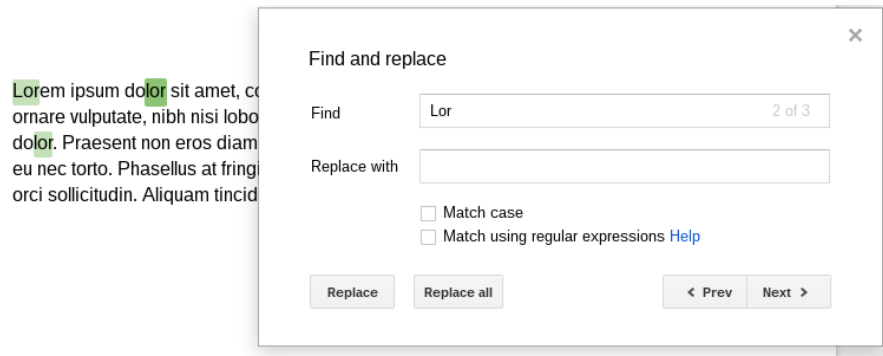


Figure 6: Google Docs

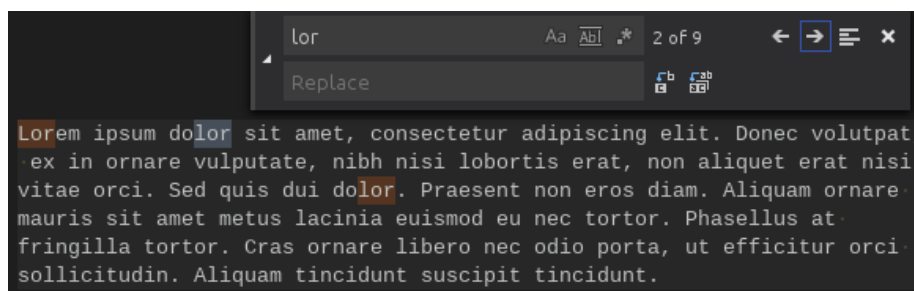


Figure 7: Visual Studio Code

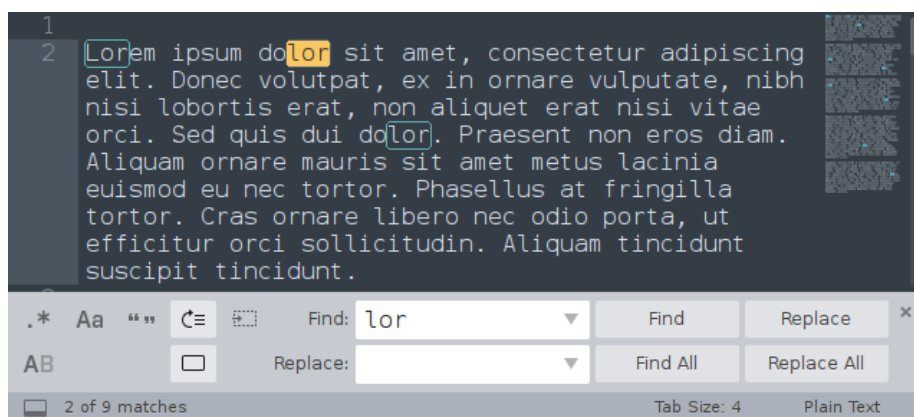


Figure 8: Sublime Text 3

At the same time, we should not assume that regular users are familiar with regular expression or more advanced search functions. Therefore, the UI design of some of these editor widgets should only be used as an inspiration - regular users are not developers and the number of options in this extension must not feel overwhelming.

## Accessibility

### Keyboard

Launch the toolbar: **Ctrl+Shift+F** (**Command+Shift+F** on Mac) Note that all **Ctrl+F**, **Ctrl+R**, and **Ctrl+Shift+R** are already predefined browser shortcuts, so we cannot use those.

Once the search widget appears: - Next match: **Enter** in *Find* input field - Previous match: **Shift+Enter** in *Find* input field - Replace: **Enter** in *Replace* input field - Replace All: **Shift+Enter** in *Replace* input field

When user jumps between occurrences we also want to keep the text in screen viewport and we therefore always check the current element position and scroll the window accordingly in case it is out of view.

### Context Menu

User can select text on the page and, after right-clicking the selection, search for the text using the extension. This should open the extension search widget and/or replace the current 'Find' input field with the selected text.

UPDATE: The background page cannot open the pop-up(<https://stackoverflow.com/questions/5544256/chrome-extension-how-to-pragmatically-open-the-popup-window-from-background-html>). It cannot be opened programmatically - the user must click on the browser action or press the keyboard shortcut to open it. But we can inject a content script that creates a floating div in the page with a hint text suggesting the user should press the keyboard shortcut to open the find & replace widget.

### Scope of Search

**<input type="text">**

For a short single line of text, HTML **<input>** element is often used. There are other types of input fields (many new were added with HTML5), such as date, email, number, tel, time, and similar, but text is the standard one. <https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input/text>

However, due to the short length (20 characters by default), this is not the kind of input that needs the find & replace functionality. The primary use case of



our extension are large multi-line text areas, so this short single-line input kind can be skipped.

**<textarea></textarea>**

Multi-line plain-text input space. This should be a common target for find & replace. It is used by many sites to allow users compose longer pieces of text, one of them is new post creation on Reddit.

<https://developer.mozilla.org/en-US/docs/Web/HTML/Element/textarea>

**<div contenteditable="true"></div>**

Enabling rich text formatting by allowing HTML tags inside the text area, **contenteditable** elements are used in Gmail, Facebook posts, Facebook Messenger, GitHub editor, Twitter, and many other sites. Note that **contenteditable** is a global attribute and is therefore not limited to **div** tags. [https://developer.mozilla.org/en-US/docs/Web/HTML/Global\\_attributes/contenteditable](https://developer.mozilla.org/en-US/docs/Web/HTML/Global_attributes/contenteditable)

**contenteditable tag inside an <iframe></iframe>**

Blogger.com is an example of a site that isolates the main contenteditable area in an iframe. When performing find & replace we must consider the scenario where we're dealing with elements inside an **<iframe>** on the page.

Also see: <http://w3c.github.io/html/editing.html#user-interaction-editing>

## Other DOM

This extension isn't meant to modify (find & replace) the raw HTML text of the page's source. It is limited to finding occurrences in text areas that are modifiable by users.

There are certainly sites that might try to avoid all the options discussed above and implement their own text editor functionality. One notable example is Google Docs, which is using static DOM but listen to user's keyboard events to modify it internally in JavaScript. Implementing your own online text editor from scratch without using contenteditable or textareas involves a lot of work, and such editor should probably include its own find & replace functionality, which is what Google Docs do.

At this point, it seems reasonable to limit the implementation to only cover the choices discussed above and wait for the user feedback to see if there are any widely-used sites containing their own implementation of text input areas.

## User Interface for Search Widget

### UI Design

TODO: Explain that extensions use a pop-up widget with `browserAction` icon

We split the UI layout to two types - simple and advanced. Because displaying all search options in one widget might feel overwhelming for regular users, there should be a way of switching the search UI to the 'advanced' state that would include regex options and helpful previews of matched regex groups etc.

At the same time, we initially disable the action buttons when the user hasn't typed in anything into the search field yet. This is in order to draw attention to the active input field element, rather than overwhelming the user with options that cannot be used. Google Docs use the exact same design pattern.

We add a box shadow highlight to the currently selected input area. If no single text area is selected, we highlight all text areas in the page. We do this to indicate which text is going to be affected by the search and replace operation and thus improve usability. Without the visual indicator people might complain that the extension doesn't work, while in fact they might just have a different text area selected. The search widget will also indicate if there are no editable text areas in the page.

### Why not Material Design

For the general look and feel, I decided not to use Google's increasingly popular Material Design for several reasons. First, Material Design works well when there's a lot of space and all the elements can be spread out. Unfortunately, this extension's user interface is a small widget with very limited space and many condensed components.

Secondly, Material Design likes to add its *ripple effect* to most interactions (such as clicking a button). I believe this looks great for actions that have large impact (e.g. navigating to a new page, or submitting a form), but for our purposes we need something less flashy, as most buttons are going to be pressed very often (Find prev/next, and Replace button) and too many effects or animations would cause too much distraction for the user.

I believe that, to a large extent, an extension (the pop-up) should feel like it is part of the browser by perhaps matching some of the UI styles. We don't want to create an extension that gives users the feeling that it completely doesn't belong because of its wild user interface and styles.

### UI Implementation

To implement the search UI widget, we could simply create DOM for all the input components and listen to any changes as the user interacts with the UI.

Unfortunately, all input components manage their own state - a better approach would be to have the search parameters state in one central place/datastore and have the UI inputs reflect this data. Therefore, we are going to use the React.js library to implement the search UI.

React has become popular in recent years - one reason is that it enforces this pattern of always reflecting the current application state in the UI. Without it, we would have to manage all the inputs separately and this could create many UI inconsistencies - incorrect update of our internal data might create a state of the application where our search parameters are set to certain values internally but display different state externally via our UI. As we're dealing with a lot of different inputs (many search parameters as well as the simple and advanced modes of the search layout), using React seems to be a wise choice.

TODO: explain React input handling (<https://facebook.github.io/react/docs/forms.html>) and technologies used (<https://facebook.github.io/react/docs/installation.html>) and my search-widget project setup + tech stack in general  
TODO: mention component separation for easier development

## API Design

There should be an extension background page with a content script that is programmatically injected into the page whenever the user triggers 'find & replace'.

TODO: Explain the reasoning behind this and how extensions work in general (<https://developer.chrome.com/extensions>). And also: [https://developer.mozilla.org/en-US/Add-ons/WebExtensions/Anatomy\\_of\\_a\\_WebExtension#Options\\_pages](https://developer.mozilla.org/en-US/Add-ons/WebExtensions/Anatomy_of_a_WebExtension#Options_pages)

TODO: Explain security scopes, Chrome API being available from the background page, page content being accessible only from content scripts.

TODO: Explain permissions set in manifest and motivation behind the `activeTab` permission <https://developer.chrome.com/extensions/activeTab#motivation> (we are not requesting `chrome.tabs` permission)

Instead of searching for a one text occurrence at a time, we want to match the behaviour of modern text editors and show the user all matches at once using text highlighting. We therefore only perform the search once (meaning every-time search parameters change), because later actions (find next/prev, replace one/all) will simply work with the text occurrences we already found.

## Component Lifecycle

Our background page is only a single JavaScript file that sets up all required events and starts listening to incoming message connections. Whenever the extension icon is clicked (or the launch keyboard shortcut pressed), our UI widget pops up. The widget can be closed/destroyed by the user anytime, so it

first registers itself with our background page, so that the background page can see when the message port disconnects (when the widget is closed).

The widget needs to communicate with the content script to manage highlighting in the page. Right after the widget connects to the background page, the background page checks if the content scripts have been injected already (the widget may have been previously closed and reopened). If they have not been injected yet, it injects them. If, on the other hand, they have been injected previously, it simply sends a message to the content script to reconnect (restart its port connection).

How does the background page check if the content scripts have been previously injected for a given tab? One solution I considered was storing tab IDs and checking a given tab ID against our stored list of IDs to decide if the content scripts are already injected in this tab. This approach might be buggy when we move tabs between more than one browser windows and is also not very efficient.

Instead, to check if our content scripts have been injected, I came up with a different approach. I created a new very simple content script, that gets injected into the page first before anything else, and simply detects the presence of the code directly in the page (it executes in the same sandboxed namespace so we can check for existence of our JavaScript variables) and returns this boolean result to the background page, which acts accordingly.

Once our main content scripts are injected, they broadcast a port connection - both the background page and UI widget are listening for this event. Background page needs to connect to the content script to see when the message port disconnects (user may have navigated to a different page and we thus lost the injected code). Search widget needs the content script connection for all its API actions - this is going to be the most frequent message passing channel.

### **Why is the lifecycle complicated**

We are managing three separate component - the search widget, the background page, and the context of the webpage itself (via content scripts). None of these components are permanent - The search widget can be destroyed/closed anytime. The content scripts in the page are lost whenever the user navigates to a different website. And finally the background page isn't persistent either - it simply sets up a bunch of event listeners that can wake it up in the future, and shuts itself down.

Could we do everything in the content scripts? Content scripts run in the content of the webpage and for security reasons they don't have access to the Chrome APIs - for a large portion of the extension functionality, and for the search widget integration, we need the background page.

## Storing State

TODO:

In Widget - Search params, all inputs;

In Content Script: Refs to all DOM textarea elements, Find-next number (because it keeps dynamically changing and search widget should only receive it from content script messages passed to it) In Background: Nothing - here we just set up events

TODO: Explain chrome.storage <https://developer.chrome.com/extensions/storage> and describe challenges faced when syncing state within the app - abstracting with Promises and chaining async operations once finished + observer pattern - subscribing to updates when another part of the app changes storage

## Favourites Panel

TODO: explain hashing search state to check if the item is in favourites

## History Panel

## Templates Panel

## Help Panel

## User actions

- Update search query or options
- 'Find' input field content changes
- One of the search options is toggled
- Find next/previous
- Replace current/all
- Close the widget
- Click one of the panel buttons
- Show Favourites
- Show History
- Show Templates
- Show Help/Info

## Actions API

User actions specified above directly translate to types of messages that need to be passed between our widget and the content scripts in the page. We define the following API, where each message has the prototype { `action: string`, `data: Object` } where `data` is optional:

**action: shutdown** - Clean up all active highlights when widget is closed

**action: restart** - Trigger port reconnect action, on widget re-open

**action: log** - Logs data

**action: updateSearch** - Uses `data` to update search parameters - Returns: { `invalidRegex`: boolean, `invalidSelection`: boolean, `searchIndex`: number, `searchCount`: number, `currentMatch`: object }

**action: findNext** - Finds next match - Returns: { `searchIndex`: number, `searchCount`: number, `currentMatch`: object }

**action: findPrev** - Find previous match - Returns: { `searchIndex`: number, `searchCount`: number, `currentMatch`: object }

**action: replaceCurrent** - Replaces current match with `data` contents - Returns: { `searchIndex`: number, `searchCount`: number, `currentMatch`: object }

**action: replaceAll** - Replaces all matches with `data` contents - Returns: { `searchIndex`: number, `searchCount`: number, `currentMatch`: object }

**action: insertTemplate** - Inserts template text `data` at current cursor - Returns: { `noCursorPosition`: boolean }

Object `currentMatch` is used to display advanced regex information about matched regex groups. It has the following structure: { `groups`: Array<string>, `replace`: string }

## RegEx Search

JavaScript contains native support for regular expressions ([https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular\\_Expressions](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions)). Without using any additional libraries, we can simply create new `RegExp` objects and execute search methods on regular strings to find matches for the given regular expression query.

When choosing the string that we want to search in, we need to only consider the text content in our `HTML` nodes - finding matches in `innerHTML` would be a mistake since we need to ignore the website's source markup and only consider the text that users see.

## Highlighting Found Matches

### Highlighting in `contenteditable`

Highlighting inside a `contenteditable` element should not be a problem - we can simply inject our own `span` element with our custom class into the element's

DOM. Contenteditable elements are designed to contain any HTML nodes so no problem here.

For the highlighting, I'm going to use the mark.js plugin. We could certainly implement the highlighting ourselves (wrapping each search occurrence in `<span>` elements with custom styling), but there are many tricky cases that we need to handle. For instance, HTML `<div>John <b>Smith</b></div>` matches the query `John S` but simply inserting a `<span>` at the start and end would violate HTML rules, and instead we'd need to create two pairs of `<span>` elements - for `John` and another for `S` inside the `<b>` tags.

There are more tricky cases like this. Therefore, it's wiser not trying to reinvent the wheel and instead use a plugin that is actively maintained and used by many people.

### Highlighting in `<textarea>`

Highlighting text in `<textarea>` is more tricky because it only allows plain text to be displayed inside it. Any styled markup will not render as expected. To overcome this, one must create an overlay that exactly matches the textarea element and then highlight text in this new element. Further, there are many browser-specific quirks and one must also take care of synchronizing scrolling and handling textareas that are resizable by the user. In other words, it is a lot of work.

<http://codersblock.com/blog/highlight-text-inside-a-textarea/>

Fortunately, there have been a few attempts to implement this. The most successful version I found was the following jQuery plugin: <https://github.com/lonেকorean/highlight-within-textarea/> because it also supports resizable textareas (other plugins I found did not).

Even though this plugin does better than all the other plugins that exist, it still has many issues, particularly when it comes to transferring all necessary CSS styles. When I was testing it, it failed to properly align text for several textareas with particular styling. Therefore, I contributed to the development of this open-source plugin by fixing all the bugs I found and creating a pull-request on GitHub.

<https://github.com/lonেকorean/highlight-within-textarea/pull/19> - Submitted: Sep 24 - Reviewed by the author: Sep 25 - Review comments addressed: Sep 25 - After 2nd review the discussion still continues, there are backward-compatibility issues and conflicting views on the design and purpose of the plugin. As I'm injecting the code dynamically I have a slightly different use case and might need to leave my forked version unmerged.

For highlighting text in the `<div>` mirroring our textarea, I'll use **mark.js** again. This is mostly for making things consistent and having a single unified function interface (finding and replacing the plaintext separately would also be an option, but we are using the library already so there's no need to make things

more complicated).

### RegExp Replace

In the replace input string, user can specify special symbols to refer to the found occurrence.

- **\$n** - Where n is a positive integer, inserts the n-th parenthesized submatch string.
- **\$0** or **\$&** - Inserts the matched substring.

These are commonly used to extend find & replace functionality in advanced text editors. Inspiration also taken from Mozilla's spec on MDN website:

[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/String/replace#Specifying\\_a\\_string\\_to\\_replace](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/replace#Specifying_a_string_to_replace)

### Handling iframes

TODO: explain recursive search for active element

TODO: explain the concept of my Context object - performing all dom operations (in particular jQuery functions) with respect to a specific Window and Document objects (such as window.getSelection, window.scroll etc.)

### Google Analytics

In order to collect statistics on how users use our extension and how they interact with different UI elements, we are going to implement Google Analytics into our extension.

Collecting this data will require additional manifest permissions and also requires us to provide a way for the users to be able to opt-out from collecting them. This might dissuade a small percentage of potential users from installing the extension in the first place, but I believe this will only be a small percentage.

### Implementing Google Analytics

The standard way of implementing Google Analytics is by inserting a short script into a web page, which then pulls more JavaScript code from Analytics servers, to collect user data, as well provide a developer interface to send additional tracking events.

This method will not work in Mozilla Firefox extensions due to their review policies (<https://blog.mozilla.org/addons/2016/05/31/using-google-analytics-in-extensions/>) - *The most popular way to do this is to inject the Google Analytics script into your codebase as if it were a web page. However, this is*



*incompatible with our review policies. Injecting remote scripts into privileged code – or even content – is very dangerous and it’s not allowed.*

As a workaround we need to use Google Analytics’ Measurement Protocol (<https://developers.google.com/analytics/devguides/collection/protocol/v1/>), which is a simple API that accepts HTTP requests with analytics data. It collects data (and aggregates it for displaying dashboards and statistics via the same Analytics web interface) without relying on the Google Analytics JavaScript library running on the client, so we are essentially asked to implement the library ourselves and only send requests in the appropriate format to this API.

TODO: Explain how session is measured: <https://support.google.com/analytics/answer/2731565>  
- 30 minute inactivity is a reasonable measure

### **General Tracking**

I am going to embed the Google Analytics code into our extension’s widget, so that we can track how many times users open the search widget, as well as how many unique users we have, and what kind of population they represent (along with additional user details).

### **Pageview Tracking**

Our search widget is essentially a single page application, but we would like to track how many times users open different widget tabs (history, templates, etc.). So we send a **pageview** event every time a user opens a tab, and set the pageview path to the name of the tab. That way we can look at opening tabs as an actual page navigation. Suggestions taken from <https://developers.google.com/analytics/devguides/collection/gtagjs/single-page-applications>.

### **Event Tracking**

In addition to the basic user and page view tracking we would like to know how users interact with our extension and in particular, if they use certain functionality significantly less/more. This might suggest potential ways we can improve our extension - for instance, if nobody ever uses the templates panel, perhaps we need to educate users on how to use it more effectively, or perhaps reconsider its overall usability.

### **Events**

- Analytics enabled/disabled clicked - If the user has allowed Analytics to be used for a week and then opts-out, we want to know they did it (rather than thinking they stopped using the extension)

- Added to/Removed from Favourites - We can get more insight by looking at how often users use the star button. If they've never used it, perhaps it's not intuitive enough.
- New Template Created - Enables us to see how often users create new templates. If the numbers are very low, perhaps they don't know how to use this feature.
- Template Successfully Pasted - Enables us to see how many users have successfully inserted a template into a web page and thus how many know how to use them.
- Advanced Search enabled/disabled - We would like to know how often users use the advanced search functionality (regex search and in-selection search)

In the Google Analytics Dashboard we can plot how many of the total number of users triggered that given event in a given time period:

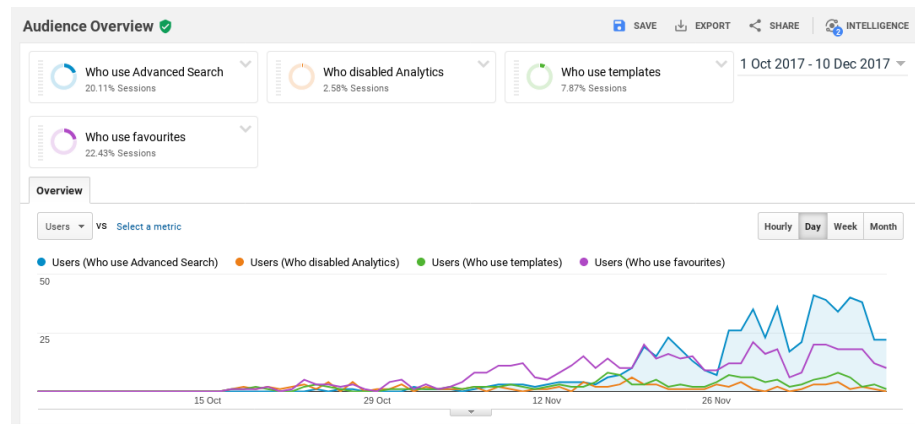


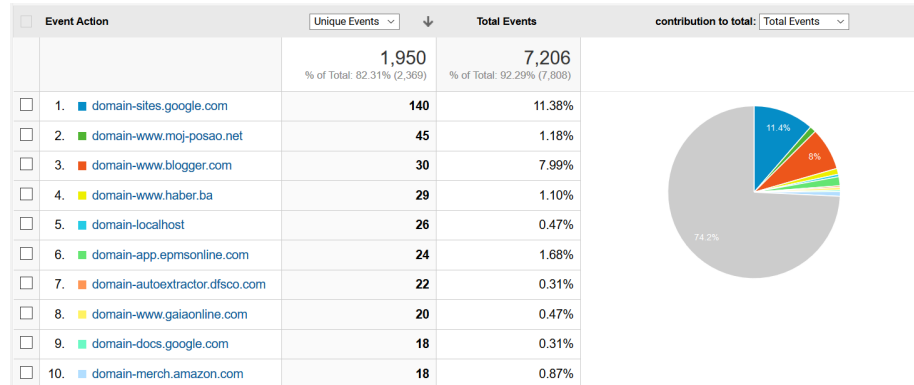
Figure 9: User Behaviour

### Domain Tracking

A lot of the feedback messages that are submitted after someone uninstalls the extension are not very useful and often look something like 'not working'. This is not very helpful as it doesn't tell me on which sites they tried to run it that didn't work for them. The analytics that I have been collecting so far give me insight into which features of the extension are more used than others but tell me nothing about the sites where the extension is actually used.

To get a better idea of which sites are the most frequently used and which should therefore receive more attention (to make sure everything works as expected), we would like to anonymously collect domain names (not URLs, only host names, for privacy issues). This would allow us to construct a list of the top domain names where the extension is used most frequently, and this could also help us with user acquisition and targeted marketing in the future. We therefore add a

Google Analytics event that reports the domain name whenever the extension is used. We can then combine unique events from each user to create the list.



Top domains after 2 weeks of reports. 13th January 2018

## Testing

How can we test our extension? The Chrome Extension Guides contain a section on debugging ([https://developer.chrome.com/extensions/tut\\_debugging](https://developer.chrome.com/extensions/tut_debugging)) but do not mention any extension testing methods.

For testing web applications in general we have several kinds of tests (Sources: <https://medium.com/powtoon-engineering/a-complete-guide-to-testing-javascript-in-2017-a217b4cd5a2a>):

- Unit Tests - testing individual functions
- Integration Tests - testing several modules working together
- Functional Tests - testing a scenario on the whole product (in-browser interaction)

### Unit Tests

Our content script contains a lot of individual functions that we can test separately. I'm going to use Mocha.js for unit testing of individual methods in the content script and search widget. The coverage will be very limited however, because most functions need to interact with the Document and Window objects, which are supplied by the web browser and will be undefined when running a command line test. We therefore shift our main focus to integration tests and functional tests.

### Integration Tests

We would like to test if multiple modules or parts of our code can work well together. We split this into tests for the search widget, and tests for the con-

tent script. Testing the extension working as a whole will be addressed in the Functional Tests section.

### **Search Widget Tests**

Our UI search widget is mostly focusing on keeping the state of UI consistent. Here we are more interested in the widget being rendered correctly as a whole, and that the UI doesn't accidentally change in unexpected ways. Because of the way we separated the search widget development (described earlier), we could simply mount our React root on a standard website DOM and test basic user interaction. There will be no content script for it to communicate with, but this is not the focus at this point.

Because the search widget code is written using React, we can use React's testing tool called Jest, which makes it very easy to test React components. It is using a specifically designed test renderer that instead of spinning up an actual browser instance to render a visual component, it creates a snapshot of the resulting HTML and writes it into a file. These snapshots are regenerated on subsequent test runs and compared to the old versions. That way it can quickly discover and report any changes (and mark that particular test as failed in such case).

Also see Jest Snapshot Testing

### **Content Script Tests**

What we are interested here is, given a simple website, can the injected functions from our content script work together, so that when triggered by a mocked API call, they successfully perform the find operation and highlight occurrences.

We do this by creating a simple HTML page that already includes all the code that the extension normally injects into a web page. In addition to that, we include one more script file that contains the actual test suite.

Unfortunately, the content-script hides its scope in an isolated JavaScript sandbox, so simply including it in a local HTML page will not allow us to interact with it in any way, because the connection API that it uses is only made available to the script when it is injected via an extension. We therefore need to detect when the content script is running in the debug mode and not as a part of an installed extension (the state when the runtime connection API is undefined is a good indicator of that) and we expose the API message handling function when this is the case.

Content Script Testing Demo - YouTube link

### **Functional Tests**

To implement functional tests, we would need to drive a browser that installs our extension and interacts with it. There is a tool called PhantomJS which is commonly used for headless WebKit testing. Unfortunately, it is not based on Chromium, so we cannot load Chrome extensions (<https://stackoverflow.com/a/23643111>).

To be able to install and test the extension as a whole in Chrome and Firefox, we are going to use Selenium, which is a browser automation tool that can directly control a browser instance and allows us to install browser extensions as a part of the testing process <https://stackoverflow.com/questions/15005833/browser-plugin-testing-with-selenium/17117849>).

We are going to install Selenium WebDriver to control Chrome and Firefox browsers. The Selenium WebDriver accepts commands via a client API and sends them to a browser ([https://en.wikipedia.org/wiki/Selenium\\_\(software\)#Selenium\\_WebDriver](https://en.wikipedia.org/wiki/Selenium_(software)#Selenium_WebDriver)). The client API has several implementations in various programming languages. I decided to use JavaScript to keep my whole codebase consistent.

The Selenium project have their own JavaScript implementation for the client API, but after comparing it to alternatives, I decided to use WebDriver.IO, which is another JavaScript implementation of the (Selenium 2.0) WebDriver API, but it has much simpler and more readable syntax (<https://github.com/webdriverio/webdriverio#syntax-example>).

### **Sites that need test coverage**

The following sites have been tested manually so far and it would be great if we could automate their testing.

- Gmail
- Blogger
- WordPress
- Reddit
- Stack Overflow
- Google Groups
- LinkedIn (article editor)

### **Setting up Selenium tests**

I successfully updated our configuration to install our extension once an automated browser instance is launched. It correctly opens a new tab with the Help text (our User guide). First functional test that I created tests that this actually happens (new tab opened on install with the extension user guide).

Unfortunately the Selenium driver do not allow direct mouse interactions with the extension - there is no way to launch it from the toolbar (Mentioned here: <https://releasematic.wordpress.com/2013/12/29/automation-testing-chrome-extensions/> and here: <https://github.com/webdriverio/webdriverio/issues/2108>).

At the same time, I designed the extension with the minimal set of permission, and so the extension doesn't execute any code and doesn't inject anything into to page unless the user launches the widget first.

The only thing we can do is open the extension's HTML page directly. So due to API limitations we won't be able to set up tests for the specific sites listed above. The only way we could test the highlighting for these sites would be to create a new extension that forces the injection of the content script code in every page the browser visits and also automatically starts sending the exact data payloads mocking the API coming from the (never running) search widget. Besides it being loads of work to implement, this would not actually be the full functional tests that we were aiming for - it would be testing the content script alone but none of the actual component interaction and communication with the search widget. We abandon this idea, because for the sole purpose of testing of the content script functionality we can find an easier and more efficient way to test things. We addressed this in the Integration Tests section.

## Manual Tests

Due to some of the limitations mentioned above and also to speed up initial development of the core functionality. I created a demo HTML page that contains all possible combinations of editable text areas - all types (plain textarea, contenteditable, and single-line inputs), all sorts of styling, various states such as with a scrollbar, or in a scrollable container, as well as text areas inside nested iframes, and various other combinations.

It is very quick to open this HTML page in a browser and use the already installed extension to see if everything is working as expected. It also allows for efficient debugging, and focusing on a specific element category when things don't work.

## Sites currently known to be broken

- Facebook - highlighting and replace works but is reverted to original onClick
- Quora - completely broken - inserting `<mark>` violates their `<span>` format and scatters original text (inserts newlines)
- Sites using the CodeMirror plugin, including Jupyter Notebook on local-host

## Why is it broken

Facebook, Quora, and several other sites use contenteditables and keep the text content separately in JavaScript variables. When I insert my markup, and replace text, their JavaScript immediately restores the previous state (switches back to the original text). When I detach their JavaScript listeners by cloning the

contenteditable DOM node, I am able to highlight and replace text successfully and the user can continue editing, but when they click the post or submit button, all changes made after the last search & replace operation was made are lost. This is because the text that is posted is the content of their JavaScript variables, and not the current contenteditable content.

### How to fix broken sites:

There would need to be an alternative way of adding/changing the editable text (besides directly changing `textContent/innerHTML/innerText`). Sending keyboard events does not seem to work.

1. Help sought on Stack Overflow: [https://stackoverflow.com/questions/46981226/edit-text-in-facebook-messenger-contenteditable-div?noredirect=1#comment80915743\\_46981226](https://stackoverflow.com/questions/46981226/edit-text-in-facebook-messenger-contenteditable-div?noredirect=1#comment80915743_46981226)
2. Marked as off-topic - but someone commented with a possible hint in time
  - Potential fix: *Focus the editor element and use `document.execCommand` in 'insertText' or 'insertHTML' mode.*
3. Discovered `document.execCommand()` API <https://developer.mozilla.org/en-US/docs/Web/API/Document/execCommand>

### Product Development Iteration with `document.execCommand()`

TODO: Explain the API, that it originated from IE, but now is part of W3C spec (+add link)

After switching to `execCommand` for manipulating `contenteditable` suddenly insertion started working on Facebook: - It fixed template insertion in Facebook contenteditable

However, for replacement (where the original content must be deleted first) it proved very unreliable, because `execCommand` operates on the active cursor selection, which can be programmatically set by manipulating the `Selection` object, but since there's no way to tell it to select only the text content itself it ended up deleting whole DOM tags and even after a lot of testing it wasn't reliably able to replace parts of text in a post and pass through all the automatic DOM manipulation that Facebook implements. Therefore, we disable the extension on Facebook and Messenger for now.

LinkedIn: All inserted content and tags are filtered so we cannot insert `<mark>`s or `<span>`s. We check which tags are allowed - `<p>`, `<strong>`, `<em>`, `<u>`, so we can use the underline (`<u>`) instead of `<mark>`. Solved.

Quora: It doesn't allow us to insert marks or any formatting as all inserted content is immediately filtered. The only tag which is not filtered out is `<div>` and Quora uses a complicated hierarchy with various class name swapping on new text insertion. This is very tricky if not impossible to fix. When users try to use the extension on Quora, simply let them know that it does not work there.


[Questions](#)
[Developer Jobs](#)
[Tags](#)
[Users](#)

## Edit text in Facebook Messenger contenteditable <div> [closed]

▲

-1

▼

★

To compose messages in Messenger, Facebook uses `<div contenteditable="true"></div>` as the input element. When I open browser devtools and change the text of the message I'm currently composing, it immediately reverts back to the previous text.

I'm making a Find & Replace extension (<https://find-and-replace-f6588.firebaseio.com/>) that needs to be able to insert text into the `contenteditable` but simply changing the DOM node is immediately reverted by their JavaScript.


Is there any other way to simulate typing in the input text (using a browser extension or JavaScript in general)?

UPDATE: I can duplicate the DOM node and then the JavaScript listeners are removed, but then the internal JavaScript variables are never updated with my new text.

[javascript](#)
[html](#)
[web](#)
[google-chrome-extension](#)
[contenteditable](#)

[share](#)
[edit](#)
[delete](#)
[flag](#)

asked Oct 27 '17 at 18:18



Dalimil Hajek

1 ● 1

deleted by [Community](#) • Nov 7 '17 at 3:00 (RemoveAbandonedClosed)

This question was automatically deleted. [Please see the help center for more information.](#)

closed as off-topic by [Makyeen](#), [EJoshuaS](#), [Machavity](#), [Ken White](#), [Paul Roub](#) Oct 28 '17 at 4:04

This question appears to be off-topic for this site. While what's on- and off-topic is not always intuitive, you can learn more about it by reading the [help center](#). The users who voted to close gave this specific reason:

- "Questions seeking debugging help ("why isn't this code working?") must include the desired behavior, a *specific problem or error* and the *shortest code necessary* to reproduce it in the question itself. Questions without a **clear problem statement** are not useful to other readers. See: [How to create a Minimal, Complete, and Verifiable example](#)." – Makyeen, EJoshuaS, Machavity, Ken White, Paul Roub

If this question can be reworded to fit the rules in the [help center](#), please [edit your question](#).

I just set `el.innerHTML = 'test';` and it worked for me without reverting. Make sure you're targeting the lowest-level `div` element that has the `data-editor` attribute. – [Alex W](#) Oct 27 '17 at 18:24

You can destroy the underlying structure like this, but when you post the message it posts the original text (changes are not saved, unless you continued editing manually afterwards) – [Dalimil Hajek](#) Oct 27 '17 at 18:56

Focus the editor element and use `document.execCommand` in 'insertText' or 'insertHTML' mode. – [wOxxOm](#) Oct 27 '17 at 22:23

[comments disabled on deleted / locked posts / reviews](#)

Figure 10: stackoverflow question about contenteditables



## How to fix CodeMirror (and thus Jupyter Notebook)

CodeMirror is using hidden textareas for input from the user and then they are shadowing this in a formatted DOM that is displayed to the user. This is how CodeMirror 2 is implemented, the previous version was using contenteditable and Document design mode to display formatted text, but this turned out to be very inconsistent across browsers, and buggy in general (See <http://codemirror.net/1/story.html> and <http://codemirror.net/doc/internals.html>).

If we had a single mirrored textarea, our extension would work just fine. However, CodeMirror has been around for several years and the current 5th version implements things differently (<http://codemirror.net>). After the plugin is initialized it waits for the user to click somewhere in the editable area, which is in fact non-editable DOM, that adds a virtual cursor animation to appear as one. It then spawns an empty textarea in that section of a line and waits for the user's keyboard input. This small textarea is immediately cleared and reset when the plugin area loses focus or when the user selects a different line (or in fact any mouse click), with the typed content being committed into the non-editable DOM to visually reflect changes.

This behaviour prevents our extension from working because there is no content in the spawned textarea, and it doesn't have access to the non-editable DOM that is displayed (because we limit our extension to editable text areas, rather than consider any HTML). Therefore fixing CodeMirror (and thus all sites using it, including Jupyter Notebook) would mean implementing find & replace for any HTML - something that we decided not to do in the initial software specification.

## Distribution & Marketing

Posted on Reddit, HackerNews, Quora.

## Feedback & Iteration

### Changelog (releases)

- v1.3.9 - 14th Jan - Support local debug mode for testing
- v1.3.8 - 14th Jan - Fix 'No Target' indicator
- v1.3.7 - 13th Jan - Add Google Docs warning notification
- v1.3.6 - 2nd Jan - Add support for single-line input fields
- v1.3.5 - 31st Dec - Add hostname to GA for domain analysis
- v1.3.4 - 19th Dec - Fix 'Replace all' with regex groups
- v1.3.3 - 19th Dec - Fix restored favorites not storing state
- v1.3.2 - 18th Dec - Fix undefined regexp groups substitution
- v1.3.1 - 27th Nov - Add info notification for broken sites
- v1.3.0 - 27th Nov - Implement dynamic mark tags based on site

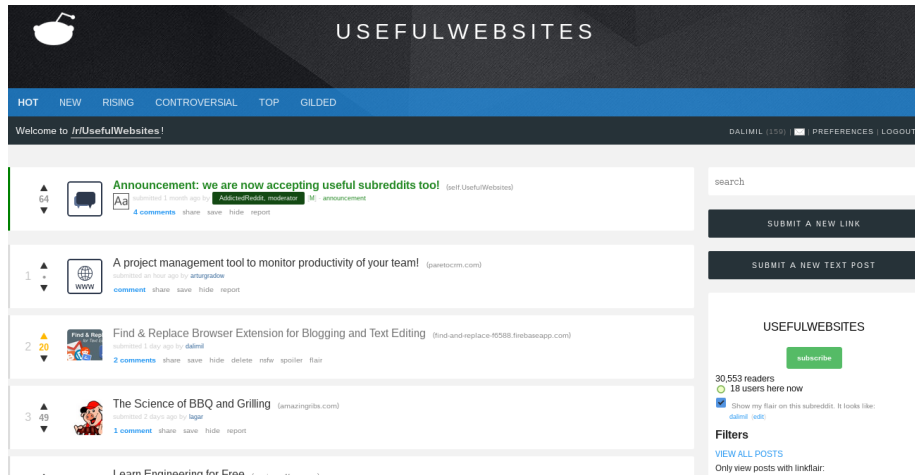


Figure 11: Reddit Marketing

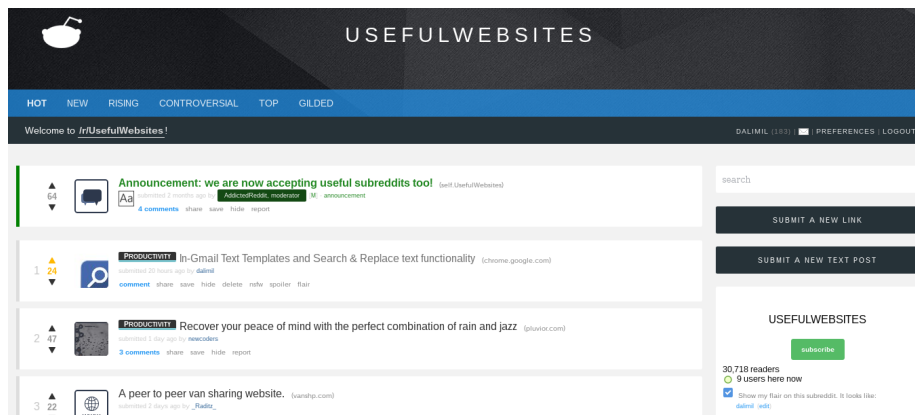


Figure 12: Reddit Marketing 2

- v1.2.1 - 18th Nov - Add more events to GAnalytics
- v1.2.0 - 18th Nov - Change templates to execCommand API
- v1.1.3 - 25th Oct - Fix bugs in replace offsets
- v1.1.2 - 23th Oct - Add help text links
- v1.1.1 - 23th Oct - Add UI hints
- v1.1 - 20th Oct - Disable all debug logs
- v1.0 - 18th Oct - First prototype released
- v0.1 - 13th Sep - Raw dev work started

### Thanks To

- Boris Grot - for supervising the whole project, and making important feature suggestions leading up to the first official releases of the extension on the web store
- Michael O'Boyle - for making suggestions, especially regarding Google Analytics
- Christoph Metze - for finding several important bugs that subsequently led to releases 1.3.2, 1.3.3, and 1.3.4
- Daniel Tomberlin - for pointing out a use case when trying to search across multiple single-line inputs, and for updating his web store rating and review after I implemented it in 1.3.6
- @MarkRH - for finding a bug that was later fixed in 1.1.3
- User wOxxOm on Stack Overflow - for suggesting `Document.execCommand` API that I used to fix issues with templates in 1.2.0

And also thanks to all those people who submitted user feedback or reviews.

### Extending scope to include `<input>` elements

Why? A lot of people are complaining about the extension not working for single-line text inputs. The argument for not implementing it for `<input>` elements was that these are only very short pieces of text (typically a few words but typically less than 100 characters).

It turns out that the scenario that users face is this: Given a very large number of single line text inputs in a single page, they need to search and replace a phrase across all of them at once.

The input types that contain standard text and would therefore be a good target of our extension are the following: `<input>`, `<input type="text">`, `<input type="search">`, `<input type="url">`, `<input type="email">`. So we are going to find these using the following CSS selector: `input:not([type]), input[type="text"], input[type="search"], input[type="url"], input[type="email"]`

The problem is that there might potentially be a large number of these input fields in a single webpage - websites have plenty of search bars or input fields

scattered around, often away from the main content - so if this functionality is implemented it should only be added as a switchable option, most likely in the existing advanced options in the search widget.

### **How to make it work**

We can reuse some of the code for mirroring and overlaying textareas, some of the styling will be different, because the input is vertically centered by default, and instead of the text wrapping to the next line, it instead continues as a single line.

The horizontal scroll itself is somewhat problematic - Firefox fires the scroll event for us and when user scrolls the input box, it preserves the scroll position after the element is defocused. We can therefore capture the scroll event and shift our overlay highlighting by the same scroll amount.

Chrome on the other hand doesn't fire the scroll event for us, but since it automatically resets the scroll position to zero after the element is defocused the initial highlighting will always be correctly aligned. The only issue arises when the user tries to horizontally scroll the input box with the extension search widget open, because the highlights will not move. This can only be done with a touchpad however, as clicking into the page would dismiss the search widget. Since single line inputs with a large text overflow are quite uncommon in the first place, we ignore this specific case, as it presents only a slight visual flaw, that will most likely not occur.

### **Support website**

TODO: explain Firebase hosting: <https://firebase.google.com/docs/hosting/>

TODO: explain Firebase functions: <https://firebase.google.com/docs/functions/write-firebase-functions>

There are various ways we could collect feedback. We could have a website that provides a contact email address or asks users for a feedback message and sends an email automatically. This email inbox could get very cluttered and hard to manage, especially since we want to collect various forms of feedback (e.g. did user just uninstall the extension, or are they happy and simply want to propose new features?).

I decided to go with an alternative solution - we store all submitted feedback messages in a database. This database can be later queried and filters based on various criteria (time of submission, feedback type, ...).

### **More ideas/TODOs**

- Look at structure of previous dissertation projects

- Find a good Markdown to Latex converter
- Use we or I in text