Sorting 400+ Chrome tabs in seconds

Source: blog.entropy.observer
44 points by thecupisblue 6 hours ago on hackernews | 46 comments

Feb 16, 2023

I'm a serial tabbist. I admit it.

Currently, I have about 460 tabs open across 5 brave windows. Let's not even get started on the bookmarks.

> *"B-b-but, they're all necessary! So much knowledge! So many good links!"*
> - My inner hoarder

Yeah, I'm like an information hamster. I just keep hoarding all the tabs until I can find enough time to read *everything* - and open even more of them on the way. And as one can assume, having so many tabs can be quite overwhelming, either when I need to find something and it's lost beyond the borders of the tab bar or when I'm just looking at the screen and getting the anxious feeling of "having so much to do" - even when there is nothing to be done.

So, being the lazy hacker I am, instead of actually sorting them, cleaning them up or **gulp** simply closing them all, I wondered - why not just let the machine do the job? Can I have a 1-click solution to all my woes? Can I Marie-Kondo my inner hoarder into submission by using code?

Luckily for us, there is a giant language model worth billions of dollars just waiting to eagerly do the job. The idea is simple: Give GPT3 a list of items and ask it to return a list of categories those items belong to. Wrap all that up into a chrome extension and let the magic happen.

So, let's crack our fingers and get coding.. or.. oh... wait..

The sweet taste of complexity

Let's backpedal a bit. So, our plan sounds simple enough. But as it usually goes in software, we missed out on some key details that are going to blow up our scope and budget if we don't think about them properly.

Some of the key issues to think about before we dive into code head first and find ourselves in a world of regret are:

- **Prompt token limits**
  OpenAI's language models have token limits - 2048 or 4096 tokens.
  Since each token is about 4 characters, that limits our prompt and response size to 8192/16384 characters respectively.

  There are a few ways we can get around this problem (we'll cover all of them):
  - Cutting our prompt into consumable chunks
  - Optimising the data sent to reduce token count
  - Fine-tuning a model for our task

- **API Key security**
  Since OpenAI API charges API calls by tokens used, our API key needs to be hidden somewhere safe. Hardcoding it in our extension is a no-no - unless we really want to pay OpenAI

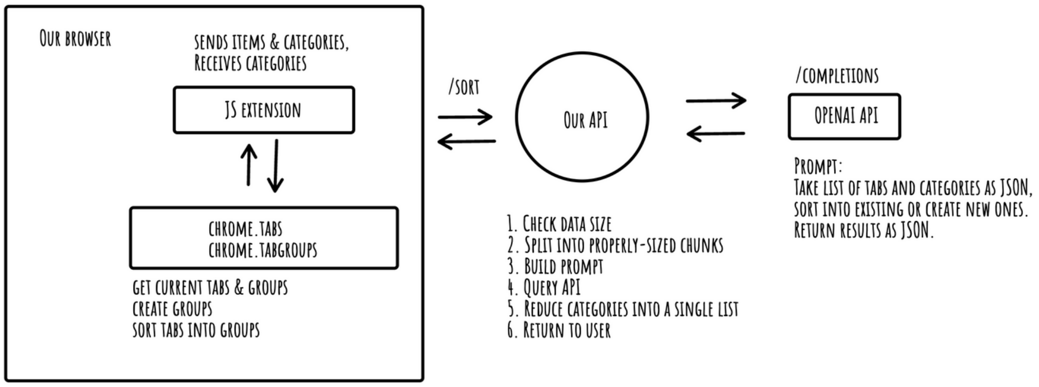
millions of dollars in bills because some bored script kiddy decided to scrape our key.
- **User privacy**
  Tab titles and URL's can reveal sensitive things - private documents,
  links, session ID's and a lot of data about a person. We want users to be able to trust the extension, so we want to open-source it, have it build and deploy from that source and make it easy to deploy for others.
- **Ease of update**
  Since LLM's can be fickle with their responses and OpenAI API could incur us insane usage costs due to simple mistakes, we want to have control over updates instead of letting the users do it at their whim. That means our most important code cannot reside in the extension.

How do we solve those issues?

We'll take a simple route - instead of writing all of the logic in the extension itself, we'll hide it behind an API - we'll build a simple backend service that will receive the tab data from the extension, chunk our prompts, communicate with OpenAI's API and reduce the data back into a single response. This enables us to both secure our keys, control our updates and open-source the extension without giving our secret token away.

To do this, we'll be using Rust - with Axum as our backend framework, Shuttle as our deployment platform and Github Actions as our CI.

So, before we get into code, let's do some napkin sketches to get an overview of what we're building:


Our browser
Sends items & categories, Receives categories
JS extension
chrome.tabs
chrome.tabGroups
Get current tabs & groups
Create groups
Sort tabs into groups
/sort
Our API
1. Check data size
2. Split into properly-sized chunks
3. Build prompt
4. Query API
5. Reduce categories into a single list
6. Return to user
/completions
OpenAI API
Prompt:
Take list of tabs and categories as JSON,
sort into existing or create new ones.
Return results as JSON.

(Not a real napkin - made with okso.app, an amazing whiteboarding app made by Oleksii Trekhleb)

Step 1: Building the Extension

Chromium extension are quite simple to build - they're basically just tiny webpages that live inside your browser and (with proper permissions) are given access to your browser by using your browser's API. We'll be relying on the Chrome API - it's the API Google Chrome uses - and which many Chromium project based browsers expose (such as Brave, which I'm using, and even Edge, tho with a different namespace). Other browsers, like Firefox or Safari aren't built off of the Chromium project, but provide a quite similar extension API. If you want to know more about the differences between them, I'd suggest this MDN article.

Specifically we'll be focusing on these two API's:

- `chrome.tabs` - enables us to query tabs our user currently has opened
- `chrome.tabGroups` - enables us to query existing groups, create new ones and move tabs inside them

So let's get to building. To bootstrap our extension, we'll be using Chrome extension CLI - it will generate the initial project structure we need.
So, hit the terminal with:

```
npm install -g chrome-extension-cli
chrome-extension-cli bookie-js
cd bookie-js
```

Follow the instructions at the end and load the build folder as an extension - it will allow you to load and test your extension via hot reload, so every change will be immediately visible.

Now, take a peek inside the structure it generated - most of it is self-explanatory,

```
├── README.md
├── config
│   ├── paths.js
│   ├── webpack.common.js
│   └── webpack.config.js
├── node_modules
├── package-lock.json
├── package.json
├── pbcopy
├── public
│   ├── icons
│   ├── manifest.json
│   └── popup.html
└── src
    ├── background.js
    ├── contentScript.js
    ├── popup.css
    └── popup.js
```

We're mostly interested in only three files for now:

public/manifest.json

The manifest is a JSON file which provides the browser with information about your extension, such as name, it's capabilities, how it's started, which file to display, scripts to run on pages and many more. A few fields to note there for us:

- `default_popup` - the HTML file to show when the extension icon is clicked
- `permissions` - we need them to access certain parts of Chrome API
- `host_permissions` - a set of URL patterns your extension can access

For now, we'll leave it all as it is and come back to it later.

src/popup.html

The starting point of our UI. This HTML pops up when we click the extension button in the browser, so we'll use it to build a simple interface here.
We'll have a 'Sort' button that calls our API's /sort endpoint and returns the result, a loading bar and a simple error box in case anything goes wrong.
For debugging, we can also have a "Show tabs" button that will show as a list of all of our tabs. So let's write some simple HTML for it:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <title>Bookie JS</title>
    <link rel="stylesheet" href="popup.css" />
  </head>
  <body>
    <div class="app">
      <div class="button-container">
        <!-- This will call our API -->
      <button id="sortBtn" class="button">Sort my mes
      <div id="loading" class="loading"></div>
      <div id="error" class="error"></div>
    </div>
  </div>
  <script src="popup.js"></script>
```

```
</body>
</html>
```

src/popup.js

This is where our JS will reside. We ain't gonna use no fancy *bulletproof cybernetically CRISPR'd SSSR JavaScript framework*, it's going to be our plain ol' vanilla JS. To update the UI, we will rely on a simple `render(state)` function that manipulates DOM elements using some simple `show` and `hide` functions (by changing `element.style.display` to `block`/`none`).

Now, let's write our thought process down by writing it into functions:

```
'use strict';

import './popup.css';

(function () {

const SORT_BTN = 'sortBtn';
const LOADING = 'loading';
const ERROR = 'error';

// get tabs & groups from the API
async function getTabsAndGroups(){};

// call backend with the data
async function callBackendToSort(tabsAndGroups){};

// apply result to browser
async function applySort(sortedCategories){};

//runs our app
async function run(){
 //get tabs
 let tabsAndGroups = await getTabsAndGroups();
 render({loading: false, error: null}

 let btn = document.getElementById('sortBtn')

 //on click, call the API, show loading and apply the
 btn.addEventListener('click',async ()=> {
     render({loading: true, error: null}
      try {
        let result = await callBackendToSort(tabsAndG
        await applySort(result)
        render({loading: false, error: undefined})
      }catch (e){
        render({loading: false, error: e})
      }
 })
}

//load our run function when the content loads
document.addEventListener('DOMContentLoaded', run);

})();
```

Our first step will be querying the Chrome API for tabs and groups. As we can see in the docs, we can use `chrome.tabs.query` to achieve this.

So, let's try it:

```
async function getTabsAndGroups() {
    let chromeTabs = await chrome.tabs.query({})
    console.log(chromeTabs)
  }
```

Not working? Now, remember that `public/manifest.json` file? And the `permissions` object?

Well, to access tabs, their titles and groups, we'll need to add matching permissions to it. So open up the `manifest.json` and under `permissions` add `"tabs", "tabGroups"`. Now when installing, chrome can check your extensions permissions and let the user know what you're accessing.
But, to be able to access the tabs API, we'll need one other special permission called `host-permissions`. It tells the user which websites the extension is enabled to run on, so if we want to be able to use it on all tabs we'll need to add the proper URL pattern. So add a new property to the `manifest.json` called `host-permissions` with a pattern allowing it to match all URL's such as `"host_permissions": ["*://*/*"]`. Finally, now we are able to access all of the user's tabs and groups.

Now that it's working, the data the `chrome.tabs.query` method returns will contain a few things we'll need: `id`, `title` and `groupId`. We'll be using `id` and `title` for sorting, and `groupId` to query existing groups, so first, we'll map the returned object to a simplified version of it, using only the properties we need.

To get more data about groups, we'll create `tabsForGroups` function which will find all the unique groups and query Chrome API by using `chrome.tabGroups.get(id)` to get the title of each group.

```
async function tabsToGroups(tabs){
  //get all existing groupIds from tabs
  let groupIds = tabs
      .map( (it)=>it.groupId)
      .filter((it)=>it!==null && it!==undefined && it

  //push them into a set to get unique ones
  let groups = new Set(groupIds)

  //query chrome API for data about each tab group
  return await Promise.all([...groups]
      .map(async (it) => {
      let item = await chrome.tabGroups.get(it)
        return {
          id: item.id,
          title: item.title
        }
    }));
  }

// now our function can return us all of our tabs anc
async function getTabsAndGroups() {
    let chromeTabs = await chrome.tabs.query({})
    let tabs = await mapTabs(chromeTabs)
    let tabsWithGroups = await tabsToGroups(tabs)
    let groups =  tabsWithGroups.filter((it)=>it.titl
    return {
      items: tabs,
      categories: groups
    }
  }
```

Boom, in a few simple steps we have the list of our existing groups and tabs.
The API calling function is also quite simple. Since our API doesn't exist yet,
we'll just write a generic POST request to localhost:

```
async function callBackendToSort(data){
 return await fetch('http://127.0.0.1:8000/sort',{
      method: 'POST',
      headers: {'Content-Type': 'application/json'},
      body: JSON.stringify({
        items: data.items,
        categories: data.categories
      })
    })
}
```

Our render function is quite simple too - we just check the state and change our UI accordingly.

```
function render(state){
    if(state.loading){
      show(LOADING)
      hide(SORT_BTN)
      hide(ERROR)
    }else{
      hide(LOADING)
      show(SORT_BTN,true)
    }
    if(state.loading!==true &&
      (state.error!==undefined && state.error!=null))
      show(ERROR)
      showError(state.error)
    }else
```

```
        hide(ERROR)
}
```

All that's now left to do is implement the `applySort` function which will apply our new categories to the browser itself.

The idea is:

- Check if the group exists
- If it doesnt, create it
- Update it's tabs list and title

For this, we have a bit of API research to do - the documentation covering this part is a bit confusing. You'd expect to be able to have something like `chrome.tabGroups.create` or `chrome.tabGroups.update` which would change tabs in the group, but... that's naive thinking.

To create a group we use the API call `chrome.tabs.group` by *NOT* passing the `chrome.tabs.group` a `groupId`. Then, the group will be created and the new `groupId` returned to you. This is kind of a weird call by the chrome team - if groups are just containers of tabs, why would tabs have knowledge and control over them?

Shouldn't the groups be created and managed via groups API?

Oh also, if you want to add tabs to the group, you use the same call and pass it the array of tabs via `tabIds`. "Hey can I pass in the title too since we're already creating and updating the object via this API call?" No, for that you'll use `chrome.tabGroups.update` API call.

I assumed this weird syntax is because groups were a later addon in chrome so support was retrofitted into the tabs API itself. So let's test that assumption. Looking at the commit that added groups to the Tabs API, we can find the same discussion in the comments, leading us to the Tab Group API proposal. It seems the team decided to split the responsibilities between *tab management* and *group management*. Since moving a tab is *tab management*, it's responsibility belongs in the Tabs API.

The alternative proposal was also discussed (putting that responsibility in the TabGroups API), along with it's pros and cons:

From my perspective (as the user of the API), the cons list doesn't seem that bad. Tabs wouldn't need to know about groups, user security would be increased (extensions would only need `tabGroups` permission, reducing the potential area for malicious abuse by extensions) and it would *hide the implementation details, replacing them with an intuitive API, which is what abstractions are all about*. Weird decision none the less.

But enough talking about the spaghetti, let's write some down.

```
function applySort(sortedCategories){

/* The response object we want looks like:
{ categories: [
        { category_id: int, category_title: string, i
    ] }
*/



  for (i = 0; i < sortedCategories.categories.length;
     let category = sortedCategories.categories[i]
     let categoryId = category.category_id
     //check if the group with ID exists
     let groupExists = await chrome.tabGroups.get(cat
                                              .catch((e)=>u
      let groupId;
      if(groupExists === undefined)
         //if it doesnt, the chrome.tabs.group returr
         groupId = await chrome.tabs.group({ tabIds:
      else {

        //if it does, we use the existing one
        groupId = groupExists.id
```

```
          await chrome.tabs.group({groupId: groupId,
                                   tabIds: category.item
        }

        // Set the title of all groups and collapse the
        await chrome.tabGroups.update(groupId, {
          collapsed: true,
          title: category.title
        });


    })
}
```

With this, our JS extension MVP is done.
- We collect the tabs and groups
- We send them to the API
- We apply the returned sort.

Now, we don't have an API yet, so how do we test it?
We should write down some unit tests, but let's leave that for another day (no really - a few posts down we'll look into testing a chrome extension with Jest). For now, we can fake the return of `callBackendToSort` function to include a few categories and a few tab id's - something like this (but with your tab id's):

```
{
        "categories": [{
                "category_id": 837293848,
                "category_name": "Hacker News",
                "items": [1322973609, 1322973620]
        }, {
                "category_id": 837293850,
                "category_name": "Science",
                "items": [1322973618, 1322973617, 132
        }, {
                "category_id": 837293851,
                "category_name": "GitHub",
                "items": [1322973619]
        }, {
                "category_id": 837293852,
                "category_name": "Web Development",
                "items": [1322973612, 1322973613, 132
        }, {
                "category_id": 837293853,
                "category_name": "Web APIs",
                "items": [1322973646]
        }]
}
```

Now we can move on to the fun parts - building that API, prompt optimisations, GPT timeouts and fixing mistakes we'll make in the days of the future past.
Oh and we'll also be adding some more complexity and feature creep, but more on that later.

Stay tuned for Part 2 where we'll continue our adventure with everyone's favourite crab - Rust.

Rusty the crab cute illustration, simple, clean, 2022 (The Artist Is A Machine)