Airline Mile Finder

Shane Panchot

Contents

Abstract

Airline loyalty programs offer miles to their members for redemption on award flights. Airline miles are commonly earned by traveling with the airline or making purchases with a cobranded credit card. A lesser-known method is shopping through the loyalty program's online portal. When a member accesses a participating retailer's site, their purchase is tracked via a cookie. The member then earns a rebate in the form of miles. For example, a student shopping for textbooks might access abebooks.com through the portal and earn 1 American Airlines mile per dollar spent. The burden of accessing the portal before making a purchase creates missed opportunities to earn miles toward free flights. The goal of this project is to create a browser extension that alerts the user when they are on a page with an available promotion with any of the 4 largest US airlines. This paper describes the design and implementation of the application including the data structures, algorithms, design patterns and frameworks utilized.

Narrative

The goal for this project is to create a web extension to notify a user when the website they are on has promotions available with an airline shopping portal. The finished product should aggregate data from the major US airline airlines into one interactive menu. That menu should provide links to each portals webpage to activate the promotion.

Initial Research and Design Considerations

Dataset

For this project, the availability of a usable dataset was critically important. I first determined there were no published APIs and therefore turned to web scraping. I started with just American Airlines and identified their web page listing all available deals without user authentication. However, the data I needed was rendered by the client so fetching the raw html from the URL was not an option. I used the inspector to monitor XHR request and found a URL that returned a static HTML file with details about all the promotions. The HTML file used nested span elements with classes identifying the data. It was not something I had seen before but seemed to be structured in a logical way that would make scraping easier than expected.

I then examined the other airline portals. To my surprise, they all used the same platform, and the data was accessible and structured the same way. With the consistency of the dataset between each airline, I decided to move forward with the plan to support multiple airlines in the project.

At this point, the usable data included each merchant name, the link to the promotional page, and the description of the promotion. The problem of identifying the merchant from the user's page still needed to be solved.

Inspiration for this project came from the existing Chrome extensions available for each airline's portal. I knew they must be matching the page to a merchant, so I tried to examine their extension. Unfortunately, the source code was minified and obfuscated. However, I was able to use the inspector again to monitor the network traffic coming from the extension. I found the URL to an API and with some manipulation of the query parameters, was able to return the full list of merchants with their name and a domain match pattern.

I now had all the data elements I would need coming from two data sources with the name being a common identifier to link them. This allowed me to move forward with the rest of the project.

Algorithms and Data Structures

When considering data structures and algorithms, I started by defining the requirements. Above all, the data structure needed to have fast retrieval of data. The order of the data within the data structures was not important.

The first data structure I thought of was using a hash table. However, the data included a match pattern for the domain, not the actual domain for each merchant. After some research to confirm, there was no option to do a pattern match once something is hashed. Looking for an alternative, I thought about using an array or linked list, but that would require linear search. To avoid this, I went back to the hash table and considered alternative keys. Ultimately, I decided I could extract the domain from the match pattern and use that as the key. I could do the same with the URL from the user and have a valid key. A

hash table was the most logical choice and would be possible with the data, so I moved forward with this option.

Application Architecture

The primary consideration for the application architecture was ensuring the most reasonably up to date data with the best possible performance. One option would be everything running in the browser. The significant performance impact of scraping new data with every page load meant the data would have to be stored locally. This introduced the issue of ensuring the data is updated at a reasonable frequency within the user's browser which was not going to be easy. This approach seems simple at first but there were too many disadvantages.

The second option is to move the data processing and storage to a server and expose a REST API that each user's browser can connect to. With this option I can schedule the data to be updated a few times per day from a single source available to all users of the extension. This approach is also preferable if the data source I'm scraping is changed and breaks my scripting. Updates would only be required on the server, not to each users installed extension. Having an API could also provide data for additional applications in the future. For these reasons, I decided to move forward with the REST API.

Implementation

Front End

Developing a Firefox extension uses the Web Extension API that I used for this project. To start, all extensions must contain a manifest.json file that defines the extension. The manifest includes the name of the extension, file paths to icons, requested permissions and the files and scripts that will run. In this section I will break down the components for this project.

Content Script

A content script is loaded into the webpage but only if the proper permissions are set. For my project, I defined this in the manifest file and had it run on all webpages. Because the content script runs as a part of the page, it has full access to the DOM but it doesn't have access to most of the Web Extension API and the rest of the browser. The one part of the Web Extension API it can access is the ability to send a message to other parts of the extension. For this project, when the page loads it sends a message requesting the promotions for the page. The message is replied to with a Promise that resolves to the available promotions to be accessed later. It also sets up a message listener should any part of the extension send a message requesting the promotion details about that page. I'll discuss this more in the next sections.

Background.js

Extensions run in their own hidden tab. The background script can run persistently in that tab and allow the extension to access the primary functions of the Web Extension API. This script has access to the browser and most of the Web Extension API but notably can't access the DOM. Instead, the background script listens for the message about a page coming from the content script.

When a message is received, it contains information about the sender including the tabs URL and an ID number for the tab within the browser session. The script extracts the domain from the URL using a regex pattern and makes a call to the REST API. If a 200 response is received, it then updates the button

in the tool bar called a browser action with a badge alerting the user promotions had been found. The listener also returns a promise to the content script containing information about the response.

Browser Action / Popup

The button in the browser tool bar is called a browser action by the Web Extension API. The icon is defined by the manifest and badge alerts are handled by the background script. When a user clicks the browser action, a popup is a launched. This behavior and the files to create the popup are also defined in the manifest.

A popup is created with HTML, CSS and JavaScript. The default appearance of the popup displays that no promotions found. When the popup script runs it sends a message to the current tab requesting the promotions for the current tab. The listener for this message was described in the content script section. Using the reply to the message, the DOM for the popup is updated with the available promotions.

The popup only remains visible until the user clicks away. If the user moves to another tab, the popup will have to load again requesting new information from that tab. The badge alert on the browser action is set by the background script in context of the active tab. This means new messages and request are not needed to update the badge alert if the user is just moving between tabs.

Backend

As discussed in the design consideration, a REST API was implemented to handle the data processing and storage. For this I used the Express.js Node framework. I hosted it on Heroku. Data was stored in memory for the time being.

Data Layer

The data layer is where I implemented the data processing and storage. While it will run as part of the express application, express was not a dependency for this portion. I implemented custom data structures for a hash table and a linked list to handle collisions within the underlying arraay. The hash function will be discussed in the testing section of this paper. I also implemented model classes for a merchant and a deal. Each merchant would be stored only once and then the deal for each airline would be added to a list within the merchant object.

For the data processing, I used the Axios library to make promise-based HTTP request to my data sources. I also installed a dependency for an HTML parser that allowed the use of a querySelectorAll method to filter the HTML data as needed. The first step was to process the HTML files that had the information about the deals and required the merchant name to be the key in the hash table. Going one airline at a time, I treated through the list of merchants and checked the hash table to see if they existed. If the merchant did not exist, a new merchant was added to the table, if they did exist, the deal was added to the existing merchant. This way I did not have any duplicate merchant keys.

After processing the html files for the deals, I needed to add the domains as an additional key. To do this, I created a second hash table to function as an index. Iterating through JSON data with merchant names and the domains, I added these values to the hash table.

I created a façade to interact with the data using a getMerchantByDomain() function. The logic behind this function checks the index table first to find the merchant name, if it does not exist, returns false. If it does exist, it checks the primary table using the name as the key. While this adds an extra step to the lookup process, it still maintains an O(1) average lookup. By implementing a façade here, the underlying

data structure can be updated in the future to use a database or any other data structure if it becomes more appropriate.

This process uses asynchronous JavaScript and returns a promise. These scripts run with the application starts and that same promise can be used for multiple requests. A new HashTable is not generated each time.

Express.JS

Express is a backend framework for Node. The framework implements the chain or responsibility pattern by linking multiple functions called middleware. I used a generator to create the initial template. This included setting some app level middleware functions such as loggers. It also had a driver script that normalized the port for local development or using the port from a hosted environment. The primary logic I had to implement were the routes and controllers. Only one route was needed for this project, but I implemented an MVC pattern to allow for future functionality to be added if needed. That route returns a merchant with a parameter for the domain. From the route, I called methods from the MerchantController. These controllers handled the logic of retrieving the merchant if it exists and then sending the response back to the client with an appropriate response. If the merchant was found, the reply is sent with status 200 and a JSON representation of the merchant. If the merchant is not found, the reply is sent with a status of 204.

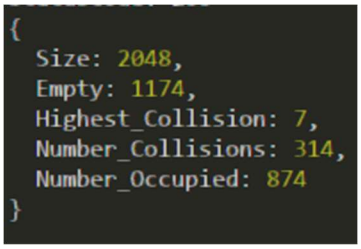
Heroku

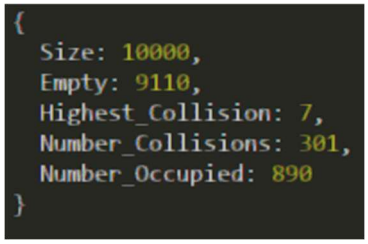
Heroku provides free dynos that would meet the needs for this project. Deploying the node application was simple and can be done with git. At this time, I have not implemented any persistent storage for the data or any jobs to schedule when the data is updated. Instead, I am relying on the ephemeral nature of Heroku to reload my application periodically which will lead to my data structures being updated. It is well documented that you cannot store anything permanently within the application dyno. The documentation says memory will likely be cleared once per day. While scheduling a job would be more reliable and predictable, for now this solution is working for what I need. In the future, I can connect addons to run cron jobs and connect to a persistent storage solution.

Testing

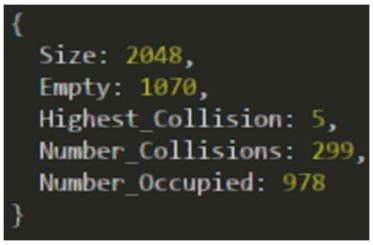
Data Structures

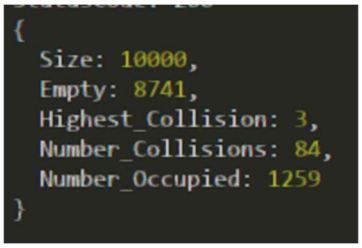
With the hash table, I initially started with a simple algorithm that would add up the ASCII character codes within a string. I then returned the modulus of that value by the size of the underlying array to determine the index of the hash. To understand the effectiveness of this hash function, I created a method to analyze the distribution of values within my hash table. I found that for my data, there were only 874 occupied slots in the array and 314 of them contained a collision. Increasing the size of the underlying array did not help distribute the values any better. The below images show the results of the analysis for this hash function:





To fix this, I found an alternative hash function called djb2. This function uses 33 to the power of each character code and multiplies it by the hash up to the point. You then complete a bit shift operation on the final value and take the modulus of the underlying array size to prevent overflows in the array. With this function, I still had a similar number of collisions with the array size 2048 as the previous function, however by increasing the size of the underlying array I saw significant improvement in the number of collisions. The next images show the results of my analysis:





Conclusion

The project was successfully implemented and includes all the features needed to provide a minimum viable product. Some additional improvements would a user settings page to allow filtering of which airlines are displayed. Also, implementing persistent data storage that is updated with a cron job rather than relying on the unpredictable behavior of Heroku. Additionally, I considered supporting multiple browsers but discovered that there were many differences between the Web Extension API for each browser. I also found that iOS 15 now includes support for extensions in Safari. To my knowledge nothing like this exists yet in the iOS app store. There is likely demand for the features of this extension on iOS. Some potential users don't make any purchases in a desktop browser and only use their mobile phone.

Additional Resources

Frontend GitHub Repository: https://github.com/kctraveler/PointFinder

Backend GitHub Repository: https://github.com/kctraveler/PointFinder-Express

Demonstration Video: https://youtu.be/2IqaQ1hJmkE