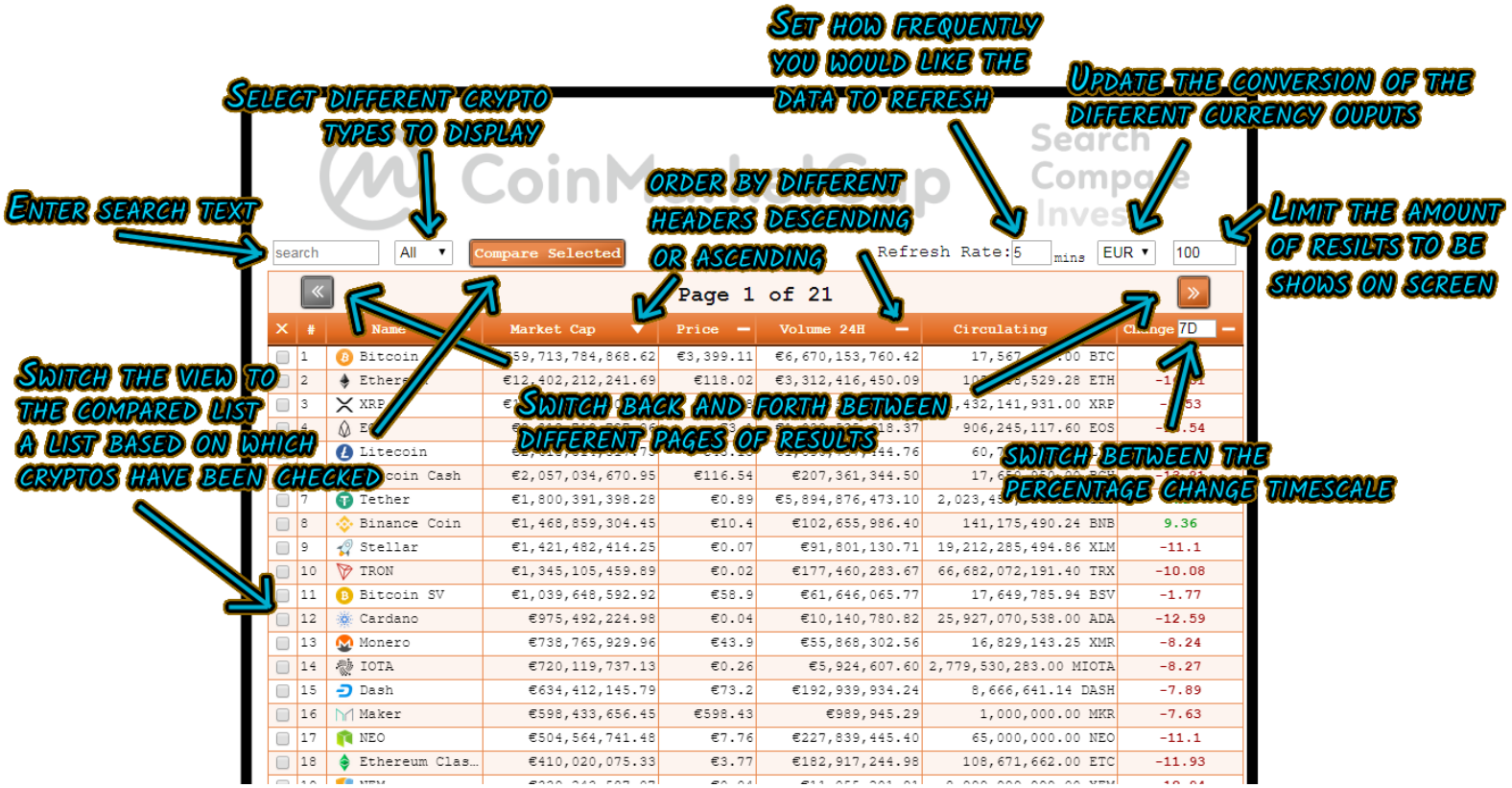

Coin Market Cap App Redux – Ethan Watson

Introduction

My main philosophy for creating this application was to try and rely on the API as little as possible, this would involve bringing in the majority of the JSON data on the initial load of the page and then only call to the API as and when absolutely necessary.

Overview, Options and Decisions Made

The main features of this application include a main view of all listed coins, a search feature, and a compare feature. These 3 features would incorporate different functionality which, when initially considered, I was thought about dealing with through 2 different API calls.

The first of these, being the main view list, would make use of the '*listings/latest/*' API call, taking in a query string of sorting, filtering and conversion options based on the table header and how the user has manipulated it. Either with ascending or descending fields from name through to percentage change over time, along with currency conversion, crypto type (coin or token), limiting page results and paging based on this limit total.

This in theory seemed fine at first for what I had set out to create the application to do, display data and manipulate what data is being displayed based on the user's interactions with sorting options. However I faced 2 road blocks with this. The first and foremost being my initial philosophy, wanting to limit the amount of API calls made so that the application wasn't relying on the API consistently for every re-render upon the user's sorting and filtering requirements.

The second was how I would deal with the search and comparison lists; there were 3 options for implementing these features.

The first was to call the API with the '*quotes/latest/'* API call, using a query string to select certain crypto coin IDs that either matched the search criteria or were found in a list of coins to compare. This however was restricted by how the '*quotes/latest/'* API call responds with data, it's response will order the crypto currencies in the order it is fed to the query string, with no options for how one would like the response to be sorted, meaning that any sorting options then placed upon the search results or comparison list would not manipulate the order of that data.

The second option was to use a similar query string on the '*listings/latest/'* API call, with a list of IDs to return, however, the free Basic account I had for the Coin Market Cap API would not allow me to quote more than one ID at a time using the *'listings/latest/'* API call, so either I would need to call the API multiple times in order of how the table sorting options have been applied, which again goes against my main philosophy for the app, or to sort the results locally with my own functionality based on how the query string has been manipulated by the way that the user is sorting the data, which brings me onto the third option.

The third and ultimately the option I decided upon was to sort the API responses locally and manipulate the data locally when the user updates the way in which they wish the data to be output. Once implemented this functionality would have 2 key advantages, one being that the API was not being called constantly for updating in accordance with how the user was changing the sorting and filter, and the second being that I was able to reuse the table sorting functionality across all different types of view list data; the main list, search results list and comparison list. Meaning that I could restrict the application from calling the API each time the user updates the sorting options and just sort all lists locally.

This still was not without its issues however. Just as the *'listings/latest/'* API call restricted calling specific IDs to only one crypto currency, the conversion between different currencies also only allows for one currency version per call, meaning that I could not call in all currency conversions at once and just manipulate which to output based on the user's decision, I would have to recall the API and update the data in each list.

This however was a fair compromise, instead of calling the API on every single sorting update, it will only recall and update data when the user selects a different currency conversion, which may only happen a handful of times under normal usage.

The other issue here however is based on the accuracy of the information. The API's data is updated with current information once per minute and with the app only updating information when the user updates the currency conversion, there is no saying how relative and accurate the data will still be after a certain amount of time. The way in which I overcame this issue was to set an interval to update all data from the API based on how frequently the user would like to see it updated. The refresh rate is set to 5 minutes initially but can be set as low as 1 minute by the user or higher if desired. This interval will call the *'listings/latest/'* API call in the background and then update all the data pricings accordingly, all within the *'action-api-calls.js'* file. This keeps the data up to date, as frequently as the user needs and also still keeps API calls to a minimum.

So initially, the way in which I first considered how to use the API would have led to calls every time the user updates the sorting options, which could have ranged from several calls per minute depending on how high the user's needs are for consistently sorting the data. However I settled on a process that would take as much reliability away from the API as possible and only call it at a max of once per minute if the user has no need to constantly change the currency conversion option, yet there is still a possibility of more if they do in a worst case scenario, but not as much as if they were sorting the table in many different ways very frequently as well.

Implementation

And so, with these issues and decisions discussed covering how the API would be used, what is the implementation of the features and how does the application deal with the data from the requests?

On the initial loading of the page, the application calls the API twice, first calling in all the currencies of type coin, and then of type tokens, this then creates two list of crypto IDs which will be used for selecting which type of crypto currency to output, instead of calling the API with a query string specifying only one crypto type each time this field is updated by the user.

The responses from these 2 API calls will then be merged and sorted by the *market_cap* field from largest to smallest based on the initial state of the *view_sort_obj*. This sorting object deals with how the data is to be sorted and manipulated, with key value pairs based on the query string that the API call would take, this object is the crux of how the view is sorted.

This *view_sort_obj* is used in conjunction with 3 lists which contain their own coins, *main_view_list*, *search_res_list* and *compare_list* which deal with the main view containing all coins, the results of a search and the crypto currencies selected for conversion respectively.

When a search is made, or the compare selected button is selected, or even if neither are, the *view_sort_obj* upon change will be used by the actions to sort these lists accordingly. The *view_sort_obj* data is manipulated by the sorting options on the header of the table, the navigation and by the crypto type, refresh rate, currency conversion and view limit total options above these.

All options, apart from the refresh rate, will either sort the data or change how the data is output accordingly. For example, selecting to sort by name will rearrange all lists by name within the *'action-sort.js'* file, while changing the *'percent_change'* select box from *7D* to *1H* or changing the number in the *'limit'* text box will alter how the data is shown on the screen when built in the *'coin-market-table-body.js'* file. (And as previously discussed, the currency conversion selection and refresh rate limit will call the API and update the data)

Another example being, when the *'crypto_type'* select box is set to coins, the *'coin-market-table-body.js*' will check which view we are on e.g. main view, search or comparison, and take the according list and create a new one by filtering the it by whether or not each coin's id is found within the crypto type coin arrays created on load and then builds the table based on this list.

Search and Compare Features

As discussed above, the application has 3 main features, to view all coins based on a sort criteria, to search for coins matching a specific given string and to view a list of selected coins by the user for comparison.

The main view's features have mostly been covered above and its sorting and filtering features also translate to the search and comparison features.

The search feature makes use of the Levenshtein Distance, taking a string from the user, this is passed to the *'action-search.js'* file and used to find matches in the main_view_list crypto currency array based on how similar the user's typing matches each coins name. If a match is found then the coin is added to the search_res_list, once the whole list has been compared to the search criteria then the search_res_list reducer will be updated via the *'actions-index.js'* and then built and output on the screen, the *'coin-market-table-body.js'* will check the reducer is_searching, which is updated to true if there is a given search string, and then take the search_res_list as its data to create a table body from. Should sorting or filtering be applied then the search criteria will be output accordingly.

The comparison_list works in a similar fashion except without the searching functionality, when the user checks the checkbox of a specific coin, this coin object form the displayed list (be it main or search results) will be added to the comparison_list reducer. If the user clicks the button 'Compare Selected' then the view will "flip" via some CSS manipulation, while the table is loading and then flip back once loaded.

All sorting and filtering options discussed with the search_res_ list and main_view_list also apply to this list.

Future Maintainability of Code

Despite how the app currently deals with API calls, there is however one key issue that could develop in the future.

The API limits calls to only 5000 crypto objects per call, which while the current total of currencies on the Coin Market Cap servers is limited to only just over $2/5^{th}$ of this total, as the coin total increases more API calls will be needed to get all of the relevant data for each coin. If we were to assume that in a few years that there may be more than 20,000 different currencies then 8 API calls would be needed on the initial load for creating the Coins and Tokens Id lists and then 4 API calls each time the user updated the currency or the refresh is called.

There is also the potential that if there were a very large amount of coins, much higher than 20,000, that the sorting and data update functions/actions could take more and more time to filter and sort data. However current growth rates of the total amount of crypto currencies in the world between 2010 and 2019 would insist that it would be many more years before even the 5000 limit is breached.