CSE**134B**

Group 8 Dream Team - BullionTracker
Final Report & Developers Guide

Greetings Developer!

Welcome to the developers guide to our application, BullionTracker! We hope this information will help you continue work where we left off and to ease any transition pains that would normally occur. Extra information on documentation of the applicatino can be find in the README.md file located in the root folder of the application. For a quick glance at this guide, here's a table of contents of what will be covered (clicking a blue link will take you directly to that section):

Status of the Application

BullionTracker is currently capable of the following core functions:

1. Market Price Tracking. The graphs on each respective metal page and the home page display average market prices for 1oz of that metal over the last 31 days.
2. Bullion Collection Tracking. Graphs on each respective metal page also contain a graph of each user's collection, represented by net worth. This is calculated through the use of the market's price per unit weight over time with the amount of precious metal owned by the user over time. What this allows is for a user to add bullions that were purchased in the past and still see an estimated graph over time of how that collected bullion fluctuated in value.
3. Bullion Adding. Users are able to add to the collection, providing an optional image, quantity, and weight per unit. From there, the application automatically calculates important values like total price, total weight in the metal, and stores that in the database.
4. Bullion Editing. Users are able to edit and delete items from their collection with the same automatic calculations that are provided in Bullion Adding.
5. User Authentication. Users are currently able to login and authenticate conveniently with Google, Facebook, or Twitter. They can also create their own custom login with an email and password if they wish.
6. User Settings. Users who created an account with a custom email and password are able to change email/password information via the settings button.

Overview of Technologies Used

HTML

Concactus

http://jesseqin.com/concactus/

Concactus, written by one of our members, was used to write common HTML fragments into multiple pages that shared the same fragment. As a python that script that simply modifies the actual served HTML files, this causes zero penalty on end-user efficiency and costs nothing

in terms of extra requests as opposed to server-side includes. Documentation, demo, and repo can be found at the link.

CSS

SCSS - Bourbon

http://bourbon.io/docs/

We wrote all of our CSS using SASS that compiled into CSS. We used SASS in a way that the syntax was identical to that of simply using CSS. The only additions we utilized SASS for were as follows:

- SASS Variables - easier refactoring of color themes
- SASS Nesting - easier for the developer to nest attributes, mainly used as a security fall-back against conflicting styling as we were merging files and work
- SASS Mixins - For any CSS attributes that needed all the extra prefix declarations for moz, webkit, etc., we used simple @include mixins from SASS to avoid having to type all the prefixes ourselves.

Otherwise, all of our SASS was written exactly the same as CSS. We simply employed the SASS to save some time and organize common themes easier.

Uglifycss

https://github.com/fmarcia/UglifyCSS

This is the CSS compressor we used to minify the application's CSS code. Using this, we managed to compress the entire application's CSS down into a single file of size 21 KB.

Javascript

Firebase

https://www.firebase.com/

We used Firebase as our backend data store for use in CRUD operations on the application. We also utilized it for user authentication. Essentially, all data regarding a user's coin collection, settings, and login information was all stored via Firebase.

ChartJS

http://www.chartjs.org/

Creating the graphs was done using a library called Chart.js. It seemed to work fine for our purposes in terms of efficiency at this small level scale and fit in well with the design theme so we decided to stick with the same library.

jQuery
https://jquery.com/
jQuery was utilized for some quicker targeting and access to some shortcut methods just to simplify the lives of the developers. While we initially started out in pure javascript with writing our own functions, we found that the trade off to the extra file space was worth some ease of coding at this scale.

TrackJS
https://trackjs.com/
TrackJS was used for tracking Javascript errors that might end up occurring on the user side. This would help us determine any errors that were not caught during development that end users could trigger in their frenzy of playing around with the application. We found this to be an extremely valuable tool for mediating any catastrophic end-user errors as soon as possible.

Mixpanel
https://mixpanel.com/
Mixpanel was used for user analytics. This would help us determine user behavior and track how users are interacting with the application.

Uglifyjs
https://github.com/mishoo/UglifyJS2
This is the Javascript compressor we used to minify the application's Javascript code. Using this, we managed to compress the entire application's Javascript (libraries AND developed code) down into a single file of size 350 KB.

File Organization

1. All **fragment HTML** files for common import are located in the concactus folder. If you change any fragments, be sure to run the concactus script before deploying the production HTML files.
2. All **production HTML** files are located in the root directory of the application. You can edit them directly and deploy.
3. The **development CSS** file is located in sass/style.scss. When editing this file, make sure the swatch.sh script is running. This will generate the intermediary processed css file in style/style.css.
4. The **production CSS** file is main.min.css, generated by running the minify.sh script.
5. The **development Javascript** files are located in in js/. unifty.js is the place to work for development code while libraries.js is the place to add any new libraries.
6. The **production Javascript** file is main.min.js, generated by running the minify.sh script.
7. All **production font files** are located in /font and **production graphic assets** are located in /assets.

Installations and Local Testing

The only installation required is for the the Uglifycss and Uglifyjs packages. These installations are handled on an initial run of the minify.sh script.

For local testing, a simple HTTP server is required: There are two simple options that we generally default to that involve either Python or Node.js.

The Python option:
python -m SimpleHTTPServer 8000

The Node.js option:
http-server -p 8000

In both examples, the URL you would use to open the application locally would be http://localhost:8000/

HOW TO: Engage in Current Setup Workflow

Below will be an example description of development workflow for the BullionTracker application, as utilized by the previous iteration of developers.

Firing up the local server

As per the previous section, this can be done using either the Python option or the Node.js option. Just be sure that the port specified corresponds with the URL you input in the browser. Sometimes, you may encounter caching issues with the local server URL not reflecting changes you made in your local files. If this happens, simply shut down the local server using CTRL+C and firing up a new server at a different port location. Then, change the URL in the browser to reflect that new port and all changes should be refreshed.

Editing HTML files

Editing HTML files is straightforward. There is only one set of HTML files in the root directory and those are the production HTML files. Changing any of the common navigation elements should be dealt with by heading into the concactus folder, changing the fragment there, and then running the concactus.py script again to repopulate the production HTML files.

Editing the stylesheet

Editing the stylesheet actually happens in a three-step process. First, you need to work in the /sass/style.scss Sassy CSS file. You may feel free to use pure CSS syntax and not use any scss; either syntax will work when compiled. If you begin editing the scss file, be sure to have the /scss/swatch.sh script running. This will automatically detect changes in the scss file and process into pure css located at style/style.css. To push that to the compressed stylesheet, run the minify.sh script in the root directory. Note that this is required to see any stylesheet changes both locally and in production deployment. Stylesheets for each individual page is organized by large header comments. You should follow this standard for best workflow between developers.

Editing the Javascript code

Editing the Javascript happens in a two-step process. If you're adding or removing any libraries, edit the js/libraries.js file. Otherwise, you'll likely be

primarily working in the js/unify.js file. This contains all the javascript for every single HTML page. After making any changes to those javascript files, you'll also need to run the minify.sh script in the root directory to push the compressed javascript file, main.min.js. Note that this is required to see any javascript changes both locally and in production deployment. Like the stylesheet, the Javascript is organized internally as if there were different files, signified by large comment headers. This workflow is also recommended for keeping things well-kept between developers.

Development Difficulties

Avoiding conflicting stylesheets

At the very beginning, we encountered some struggles with having multiple developers working on the stylesheet and HTML pages, as it may have been possible to conflict with ID or class names, or even general selectors. We solved this issue by utilizing SCSS nesting in the pre-processed stylesheet. This ensured that changes we made the stylesheet would be within the scope of the page we were editing. In the future, saving a little bit of space could be possible by purging the HTML files and stylesheet for any conflicting selectors, allowing for the removal of nesting. At the end of the day, didn't find the penalty hit for using nesting to be any reason not to use it for ease of development in the team.

Doing the research on Firebase

This one is a little embarrassing. As we were first setting up Firebase, nothing would work for the longest time. We were absolutely sure our code was right and even redid the tutorial at least 5 times. After what appeared to be several hours, we realized that run test Firebase locally, an HTTP server was required. This resulted in a round of exasperated sighs but at least we were able to move forward. Moral of the story here for us was to do our research on any new tools we were about to include; setup should never be that difficult and if it is, we were probably forgetting a crucial step.

Common HTML imports

For common elements like the top navigation and side navigation, we wanted a nice way to include those on multiple pages without having to edit every single one. In addition, we wanted to be able to test this locally and avoid any unneeded fetches to files for those imports. At the beginning, we used a

hacky approach with javascript document print statements, but with that came with a host of problems. One of which was that changing that common HTML import meant changing some HTML and converting it all to Javascript prints and *then* finally adding it to the import function. Additionally, this would cause huge problems if anyone disabled Javascript. To get around this, one of members wrote an in-house Python script for common HTML imports that writes directly to the production HTML files. This was one of the issues we faced that produced some cool results and learning experiences.

Asynchronous Ajax Calls

This was actually a pretty big headache for the developers as fetching data via the Ajax calls were asynchronous. This presented a development pain when trying to populate certain areas of our application pages. Essentially, we would keep forgetting that we would need to utilize callbacks correctly and not attempt to use any variables modified in the Ajax call outside of that scope. We never really came up with a very clean solution for handling that code-design wise and sort of manually dealt with each case when we needed to fetch data. For the future, it was probably be wise to set up a better system for populating and changing the application from data fetched by asynchronous ajax calls.

Issues and Bugs

Firebase Performance

The loading time for Firebase data is significantly slow at certain times. In order to improve this, caching or cookies can be helpful for potential use in the future. The slow performance of Firebase might also make users think that certain features are not working. It may be wise to include UI/UX indications of loading in the future so as to not frustrate the user.

AdBlock and Error Tracking

There are some cases we tested where the error tracking code would not work if the user had AdBlock. This is definitely something to look into in the future to work around. Error tracking is extremely important and should be available for as analysis on as many end-users as possible

The Odd Device

The BullionTracker device does not work on an iPhone 5. This is also something to be looked into as the population of iPhone 5 users is likely to be a significant statistic.

Font Issues

The monstserrat-hairline-webfont.tff font does not load correctly in Firefox and has some slight issues. Nothing build breaking, however.

Firebase User Creation

Sometimes Firebase won't create a user object properly. This can be due to XMLHttpRequest or Firebase's security rule issue. When moving forward it would be strongly suggested to look at other back-end store systems and only use Firebase as a temporary development tool.

Safari Private Browsing

Login does not work in Safari private mode. Likely also associated with Firebase security policies.

Local Work without a Server

Login does not work if browsing the files entirely locally without any http server.

Stray Firefox Warning

Firefox throws "The connection to wss://s-dal5-nss-28.firebaseio.com/.ws?v=5&ns=134b-dreamteam was interrupted while the page was loading."

Moving Forward

UI/UX Popout Notifications

This was one feature that we never made it around to implement. Giving the user notifications of when something changes on the page, particularly with CRUD operations would be an extreme benefit to usability. Little popout notifications saying things like "coin added, click here to undo," or "coin deleted, click here to undo" would definitely make the overall flow the

application provide a much stronger connection to the cognitive processing of the end-user.

Community/Market Capabilities

The application was initial conceived and set to be able to house an entire community. Currently, it is still very single-user oriented with a simple login and data store for that user's needs. There is still no ability to community with any other users of the application, let alone be able to initiate trades with them. When looking to the future, it is highly encouraged to consider this facet of the application as community can really change the end-user experience. However, if that's not the main goal you end up having in mind for the application, it's perfectly fine to not take that route.

More Reliable Backend Database

Firebase was a great introductory resource for learning purposes, but we didn't really like the fickle properties of the service. Interactions with Firebase ended up being a little bit unstable, which in turn is way too unstable for any production deployed application. It is highly recommended that you look into other possible tools for handling a backend database.

Handle Unsupported SVG

The previous team utilized a lot of SVG icons for ease of implementation and fast performance. However, there are certain browsers like the low versions of Internet Explorer that do not support that type of graphic. We encourage that you produce some spritesheets to serve as backups when the SVG icons are not rendered by some browser. This would ensure that some users do not end up confused with blank boxes or blank areas in the application. We anticipate that this will be a needed change because of the age demographic of the target population of this application.

Thank you!

We would like to take this time to thank you for walking through with us the development process and guidelines for BullionTracker. It is our hope that this guide will aid in all future development, whether it follows the same workflow, employs the same technologies, or none of that at all. Feel free to contact us at any point in time with any legacy questions and we'll be sure to get back to you as soon as we can! *Cheers! - The CSE134B Dream Team.*