**Implementation Document**

**Repo Structure**

The code is mainly seperated into client and server side code. The client side main uses php , css to render the html pages. Whereas the server side code has pages that perform server side render, components acting as API to parse requests , a functions folder which has the important functions that execute the SQL , and get called by the API's to retrieve data from the database. We also came up with a master database method in functions/db\_connection.php called executePreparedQuery($query,$params) . We called this function by all other functions to execute our SQL Queries. Since this handled preparing the connection and binding the parameters as well, which simplified executing the remaining queries for us.

We also wrote out unit tests for almost all of our important functions in the src/tests/ repo, which we automated them running through github actions as seen in ~/.github/workflows/ which ensured that the team was easily able to debug most of the problems if some functionality was not working. These tests can also be seen through the github actions tab.

**Initial Home.php layout explained**

We wrapped each item in a <section> tags , and then within that section used <aside> tag to create the image description , and then used the <article> tag to display the right half which consisted of all the comments. This idea was taken from this [W3 Schools Article](https://www.w3schools.com/html/html_layout.asp) .

On top we designed the Header and Footer Seperately , and included them on rest of our pages.

**Authentication**

For the authentication mechanism, the client side code sends MD5 hashed messages to the server side code. From the Client Side we made use of the [Crypto-js](https://www.npmjs.com/package/crypto-js) library , to convert the text string into Hashed Password. Which were then submitted through the form, to the server functions.

**Authorization**

To ensure that normal users cannot access admin based information , we made sure to use variables $\_SESSION['ADMIN\_EMAIL'] & $\_SESSION['USER\_EMAIL'] that render certain information based off if they were set or not. Then once the user logs out we unset all the session variables, and delete the session.

**Adding items through file upload**

The add bulk items through Admin Console , utilized javascript to parse the text file (client side) then sends multiple POST requests to the add Items function (Server side), which then adds the individual items into the database.

**Database Functions**

Located in src/server/functions/db\_connection.php , first parses the right Env variables to get the appropriate then establish the connection.

ExecutePreparedQuery Function - Expects a SQL Query , and an array of the binding params , with the first element a string of chars that represent the variable type of the corresponding (sample usage given below). We primarily made use of This function , in almost all of our other functions to query the database. Except some functions that required the usage of SQL Transacations.

public static function itemExists($ITEM\_ID){  
 $query = "SELECT \* FROM ITEMS WHERE ITEM\_ID = ?;";  
 try {  
 $response = executePreparedQuery($query, array('i', $ITEM\_ID)); /  
 if ($response[0]) {  
 if ($response[1] === "NO\_DATA\_RETURNED") {  
 return "ITEM\_NOT\_EXISTS";  
 } elseif (is\_array($response[1]) && count($response[1]) >= 1) {  
 return "ITEM\_EXISTS";  
 }  
 }  
 } catch (Exception $e) {  
 echo "Error occurred, when using Database function to try to validate User.<br>";  
 echo $e->getMessage();  
 }  
 return "USER\_NOT\_EXISTS";   
 }

We also utilized Environment variables stored in local.env and server.env files. The idea was that on the server we would delete the local.env file and it would automatically pick up the environment variables from .

This did not actually end up working , so we ended up manually hardcoding the credentials.

**Mobile User Requirements**

Mobile user requirements were achieved primarily using changes in font size. Each client side page included a CSS media query file called css/mobile/global.css which was loaded if the window resolution met a certain threshold (under 500px). This page was responsible for targeting elements with adjusted sizes. When possible, larger parent containers were initially targeted with a smaller font size (usually 50% of regular one) and their individual elements were then adjusted with em units. By doing this, the child elements would scale in accordance with their parent containers.

In order to ensure that the website renders appropriately on mobile websites, we created seperate mobile class css stylesheets that we imported into our home page , after we had imported the main stylesheet. Ensuring that the styles from the mobile stylesheet are applied on top of the main stylesheet . We imported the mobile stylesheet after the main stylesheet onto the html page. Also the tag media="screen and (max-width: 650px)" , will ensure that .Example given below :

<link rel="stylesheet" href="css/home.css" />  
 <link rel="stylesheet" href="css/mobile/global.css" media="screen and (max-width: 650px)" />

**Search and Hot Threads**

This feature primarly works by Loading all the results from the server, then filtering them on client side to filter results by search text. The filtering happens in the scripts/home.js file which filters all the results using the filterStoreItems() function. The Only time the Client Side requests new data from the server side is when the client switches between different stores.

**Switching Between Different forms when Adding items**

The add Items page gives the admin option to add items in bulk , add items one by one , and also add a new category. The way we switch between these forms is by Hiding and Showing the appropriate forms by using jQuery Show and hide functions , upon selection of the appropriate radio buttons.

**Upvote Item**

Since every time a user / admin upvotes an item , it submits a GET request which updates the new upvote count in the database. And Since the home.php is setup in a way that it always displays records in Descending order as per the SQL query , upon reloading of the page the items with the highest upvote will bubble upto to the top.

**Price Chart**

To generate the price chart we utilized ChartJs in server/priceChart.js which expects a 3D array of prices and charts which is returned by the parsed\_GetAllPrices($itemID) function. Then with additional formatting we generate the chart in server/priceChart.php .

Documentation for the version of chart JS we used can be found [here](https://www.chartjs.org/docs/2.9.4/charts/line.html) .

**Weather API**

The server/weather.php utilizes the functions written in server/functions/weather.php file, to display the weather on home.php page. Everytime getWeather($cityName) is called, it checks if the weather has been updated in the weather table , in the past 15 mins (which can be changed from server/weather.php file). If the weather for the request city has not been updated in the past 15 mins, then we update it in the database and then return the updated weather. This ensures that we do not make too many API calls , since we get limited number of free API requests per day , and ensures we still display the latest weather.

**Page Security**

To ensure that normal users cannot simply change the url to access admin based informaiton. We added these lines of code to the top of every page to ensure that if an Admin was not logged in, it will redirect the person to the Bad navigation page.

if (!isset($\_SESSION['ADMIN\_EMAIL'])) {  
 header('Location: ./bad\_navigation.php');  
}

**Breadcrumbs**

From the server/breadcrumbs.php we render the breadcrumbs on the client side. The Buildbreadcrumbs method , takes the current url from $\_SERVER['REQUEST\_URL'] , then remove the baseURL = qfinooch/src/client from the URI array. Then we attach a *start breadcrumb* address , which we choose as the login page, and then every single subdirectory is added as a breadcrumb unit with the right address.

Therefore, we decided to put all the pages that extend from home in the directory client/home/ so that all those pages have the parent breadcrumb home. And automatically every page will have the breadcrumb Login at the beginning of them.

**Ban User Functionality Toggle**

To switch between Banned and Not Banned Functionality we implemented a GET API , that recieves the user\_id of the account it was intended to ban , and toggled between the banned status of that. Therefore Upon page reload of the display all users page , it simply toggled between the status , and then displayed the new status.

**Asynchronous Updates**

To ensure that our site , checks for latest comments regularly. We enclosed our main get UpdateFilteredItemsList() function in an automatic timer that runs every 60 seconds , and also remember the last requested list. To ensure that our comments, and items are updated every 60 seconds. This will not only ensure that all item comments are updated every 60 seconds , incase a different user adds a new comment. But also will show refresh the items at the store , in case an admin adds a new item.

setInterval(function () {  
        updateFilteredItemList();  
   }, 60000);

**Final Remarks**

We were also able to host the website on our server, ensured that all the functionality reflected the right information in our database. We also created an ER diagram for our database in ~/database/er\_diagram.png , for anyone to have a look at how we designed the database.