Tony Ly
Mark Santa Presca
Alvin Yan

Homework 5: Specifications

Development Overview: CRUD Implementation

The most important part of our application was the structure the Firebase database. First, we used Firebase's authentication and Google authentication to create a login and signup page. When a person signed up for an account, a unique user ID would be created and stored to the database under a User array. That would be used to hold the person's custom amiibo as well as hold the checklist data of the original amiibo (whether they had it or not). Also in the database is static data for the original amiibo, amiibo that were officially released by Nintendo that we stored information for. After we built the structure of the database, we moved onto the CRUD functionality and asset management.

For the CRUD functionality of **creation**, it was dealt with in creating users and custom amiibo for each user. Custom amiibo would be added to a user's custom amiibo list in the database. But a problem we would have to face before being able to add to a user's amiibo list would be the ability the add photos. To do this, we utilized Firebase's storage of photos. We created a Custom-Amiibo folder that held UIDs that held photos for each person for our asset management portion. There was another problem that we faced, which was the pushing of two photos of the same name. The later photo would invalidate the link the the earlier photo, essentially not allowing users to upload multiple photos of the same name. To avoid this, we added a timestamp to the file name to avoid conflicting photo names. Other than that, adding to Firebase using the Firebase API was simple enough.

For the CRUD functionality of **reading**, it was dealt with by reading the original amiibo data, reading their have or have not status, and reading from the database to display a user's custom amiibo. The only problems with reading from the database was setting the src of the image element, as images took the longest to load when loading from the database. A minor setback in reading was setting Firebase rules for who was allowed to read or write to the database, which we were unaware of and caused some problems.

For the CRUD functionality of **updating,** it was dealt with by updating the user's have or have not status for both their custom amiibo and updating the have or have not status of the original amiibo list. Updating was also implemented in the edit page, where users could update a certain attribute of their custom amiibo, for example, uploading the custom amiibo's name or picture. However, when updating an amiibo's picture, I had to add to Firebase's storage before I was able to update the src of the image, somewhat leading to a double dipping for creation.

For the CRUD functionality of **delete**, it was dealt with by deleting from the database the custom amiibo's reference on Firebase using Firebase's API. It was a straightforward process for deleting.

Development Overview: Method Implementations

For AmiiboDex, all of our main methods was created in vanilla JavaScript for its quick speed due to its asynchronous nature. We had many methods that seemed like a repeat, however, were necessary to differentiate custom amiibo from the original amiibo already in the database. For example, the markOwnership() and markOwnershipCustom() methods have a relation For all pages, we checked whether the user was logged in and redirected to its proper place. The function signUpUserData() and google_login_in() both sign up a user and create new UIDs. It then calls writeNewUserData(password, authId) which wrote to the database a custom list as well as their personal checklist of original amiibo. We checked if a user was logged in using the API call firebase.auth().onAuthStateChanged(). Whenever a user checked a checkbox, we called updateOwnership() to update the database according to amiibo that they changed the have status. The forEachFunction(item, index) is used to display all points of data for each point of data "item." The function addAmiibo(authId) is called when the user clicks the "Add Amiibo" button when logged in. It will take their UID and add all the input data into a custom amiibo. The function displayCustomEdit(object) will display all the user's custom amiibo from the database. The function editAmiibo(object) will change in the database the amiibo that had it's data changed. Users are able to go back to the main page by clicking our banner that calls the function backPressed(). Finally, users are able to log out with the function signOutUserData().

Performance, Optimization Overview, and User Experience

An optimization we made from our last homework was to incorporate modals in which would reduce the amount of HTML files as well as improve our user flow. Instead of leading to a page that would be just a single form, we created modals with the functionality already implemented to create a better way to create custom amiibo, sign in, sign out, or sign up. This inspiration came from many websites having modals as a way to keep user flow continuous and less chunky, for example, from Facebook where you add a status in a modal.

With a lot of data to display, our first method to increase performance and create a better user experience was to load five of the elements at a time of a table filled with empty rows, however, this would lead to an empty list that slowly filled up. After talking to the professor, we optimized further by adding a progress bar that showed how far along loading the table it was, then showing all the elements when it was done. However this was extremely slow because of the loading of all elements at one time. Our next optimization for user experience was creating a JQuery table that included pagination, number of elements to show, sorting, and also searching within the table. However, the problem with the JQuery table would be that all elements would be needed to be loaded before we would be able to change the HTML table into a JQuery table.

Our final optimization was to first display a single page in a “fast table” while loading all the other elements in a pagination table while the users were still looking at the first page of the “fast table.” This gave the perception that we loaded the full table quickly, while we were buying time for the rest of the table to load. With all these optimizations combined for user experience, this lead to the final product of a progress bar that showed the loading time of a fast table, which then lead to the load of the full table with pagination. This way, we are able to keep the user engage with the website, as it provided a responsive feature that seemed quick on the first load.

Progressive Web Application Changes

Using the Google Chrome extension Lighthouse, we were able to analyze our page and make it into a progressive web application with the help of service workers and a manifest.json file. The service worker helped us maintain our site when there was spotty internet connection and would work with Firebase to finish a task whenever the internet went out and resumed that task when reconnected. The service worker also helped us manage an offline page which would help users connect when possible. The service worker also helped cache files so that revisits of the same site would load almost instantly because so many files were cached. This also lowered revisit load times. The usefulness of a manifest.json was to create a “Add to Homescreen” button to Android phones that would help frequent users of the application have a way to quickly access our site.

Lighthouse also gave us a good look into our load times, from a fresh load to multiple loads from our cached files from our service worker. Though for this application we tried to increase our page load performance, it really did not have a good initial performance because of the time it takes to upload the main JavaScript file. After the main JavaScript file is uploaded however, the application runs quickly. When revisiting the page, the caching from the service worker increases load speeds immensely.

Future Overview and Setbacks

Looking at our application, something we would like to improve on would be the mobile friendliness of our application. Instead of a very wide table that would require side scrolling, we would have liked to make a minimized table which removed a lot of the columns if the website detected you were on a mobile device. For example, instead of showing the ten columns in our AmiiboDex, it would only show four: Number, name, picture, and have or have not. This would have the same functionality we wanted, however presented in a more mobile friendly way.

Another thing we would like to improve on would be the initial load times, based off of what Lighthouse gave us. It seemed that the bottleneck for that test would be the JavaScript loading time for the first visit. However, after the load speeds are very fast. If we had more time, we would try to implement a first print that would be available without JavaScript. Though it may be tough because all of our data is in the database and JavaScript is required to access data in the database.

A large setback was JavaScript’s asynchronous nature. We had many for loops that did not finish retrieving data, so we had a function that would call setTimeout() to try and wait for the for loops to finish all iterations. This happened many times and was annoying to deal with, however, the JavaScript ran very fast.