CSE 134B Homework 4 Write-up

Lines of code: 1,453 (not including the VueJS and Firebase libraries)
Hours taken: ~7 hours

VueJS:
We chose to build our app around the VueJS framework.

Pros

- Provides a lot of functionality baked in, reducing the amount we have to do manually ourselves versus a VanillaJS or jQuery-based app.
- VanillaJS and jQuery don't provide much structure for building an app, which can lead to messy/spaghetti code as application complexity increases. VueJS provides concepts like components that help us compartmentalize our app implementation and lend themselves to loose coupling.
    - HTML, JavaScript, and CSS can be bundled together for each component and isolated from the rest of the app, helping separation of concerns and reducing the number of places to visit when making changes to a component.
        - Note: we took our CSS directly from the vanilla templates we made for HW3, and kept it in a single global stylesheet file instead of splitting it out over our VueJS component files. Since we'd already written it this way, it would have been a lot of work to go back and carve it up between components now.
    - In our vanilla templates for HW3, we had a set of "game widget" markup that was duplicated across many different static HTML files, and making changes to it required making the same updates across all of those files. When converting over to a VueJS component setup, we were able to factor out our game widget into a single reusable component that can be embedded in different site pages.
- Data binding makes building interactive views and keeping state in sync much easier than directly setting attributes and listening for change events via a DOM API.
- VueJS's Webpack integration provides live editing and hot code reloading without page refreshes, giving us rapid feedback as we worked on our prototype.

Cons

- More concepts to learn. Specialized attributes like v-on for events, v-if conditionals, and so on are all specific to the VueJS library and increase cognitive burden of getting up to speed with the app.
- Larger total code size delivered to the web browser, counting the VueJS framework code.
- Harder to do deep performance optimizations.
    - Lots of code you don't have control over.
    - While all the baked-in functionality is useful, it's more code running in your app that you don't totally understand. More technical debt.

- Framework structure can be limiting, implementing particular functionality may be harder to do in the “proper” VueJS way.
- VueJS is relatively new and doesn’t have the same level of backing as, say, jQuery, React, or AngularJS. Then there’s some risk involved in building your app around it: maintainers may disappear.

Neutral

- Integrating with the Firebase SDK required some lower-level coding in our VueJS components. To listen to authentication events in a component, for example, we passed in a firebase.auth reference as a prop, registered a listener in the component’s ‘mounted’ hook, and saved the unsubscribe reference to unregister the listener in the component’s ‘beforeDestroy’ hook. In a real-world setting, with fewer restrictions on library usage, this could have been smoothed over by using a plugin like ‘vuefire’, which provides native integration between VueJS and the Firebase SDK.

Firebase:
We used the Firebase Web SDK to handle authentication, database, and image storage for board game entry photos.

For the purposes of this project, Firebase was ideal for getting a backend up and running quickly. After setting up traditional email/password authentication, plugging in Google authentication was near painless, for example. Setting up database servers, putting together file storage infrastructure, and tying it together with a server-side HTTP interface would have dramatically increased the length of time taken by this assignment.

In a real-world setting, however, one would want to more critically analyze the benefits versus the cost of using Firebase. Firebase ends up controlling all your application data and, critically, user accounts, increasing vendor lock-in and exposing one to risk of Google cutting off the service or shifting around the pricing model.

The schema-less JSON storage model of Firebase made it easy for us to start pushing data from our app, but in the long run the lack of constraints may lead to data corruption or painful manual migrations as the application evolves. By default, all authenticated users can access all data in the database, push malformed data, and vandalize or destroy existing data. Preventing this requires learning Firebase’s security rules system, which entirely differs between their Database and Storage offerings, and carefully constructing and maintaining rules sets without accidentally leaving something improperly exposed.

During development, we noted that Firebase Storage does not offer any way to resize images uploaded by users, always serving back the original, potentially excessively large, files. It can also take a short delay for Firebase to return the public URL for a requested file reference. To speed up time-to-content for the user, we observed the typical format for these URLs and wrote

code that guesses the URL for a game photo before receiving the definitive version from the Firebase API.

Finally, the always-on socket model used by Firebase is overkill for the CRUD app we were building, which would map more directly to a request-response model. Dynamically loading content this way through Firebase's Web SDK may lead to issues with search indexing and precludes operation on environments without JavaScript capabilities.

Performance:
Our web application was tested on a BLU R1 HD device owned by one of our group members, which happens to be the same device recommended by instructor Powell as an example of a low-end Android phone. Subjectively, the app performs quite well even under this resource-constrained environment. From a clean cache, the initial page load (visiting the application index, which downloads the application HTML, JS, and CSS source code) took less time than loading a New York Times article. Page transitions--clicking the Login button, for example, and navigating to the Login page--happen instantaneously on the client. Uploading a large photo for a board game entry can slow down the load time of its view page; unfortunately, Firebase Storage does not offer a way to resize uploaded image files.

Our results for testing with the R1 HD are all subjective because we were unable to get Chrome Dev Tools to recognize the device for remote debugging. We obtained the following timing and byte count data from testing with an old Samsung Galaxy S5 Active, a 3-year-old device which has seen better days.

Load times and byte counts (full page loads, from a clear cache each time)

- Logged-out homepage: download 210ms, DOMContentLoaded 900ms, bytes downloaded 139kb
- Login form: download 250ms, DOMContendLoaded 990ms, bytes downloaded 139kb
- Registration form: download 225ms, DOMContentLoaded 1s, bytes downloaded 139kb
- Logged-in homepage: download 200ms, DOMContentLoaded 900ms, bytes downloaded 139kb
- Add/edit game form: download 125ms, DOMContentLoaded 975ms, bytes downloaded 139kb
- View game page: download 1.5s (including uploaded game photo), DOMContentLoaded 1s, bytes downloaded 178kb (including uploaded game photo)

Note that the above are all measurements of worst case situations for page loads, as each page was fully refreshed from a clear cache, redownloading all assets and starting up the JavaScript code from scratch each time. After the web application is "booted up" from a first visit, page transitions are instantaneous save for dynamic content and image asset fetches from Firebase, which happen in the background.