University of Illinois at Urbana Champaign

CS 411 Final Report

Team 010 - Aurora Aura

Watch.io

Based on our project proposal for Watch.io, we were mainly on track with our initial proposal as we wanted a watch collector website that holds the database of watches for users to save their own collection or browse other watches. We wanted to connect different watch collectors together and build a community for them by having interactive features on our website. From the project proposal, we wanted a filter option for the search in which we were able to complete that and return all watches based on what the user typed into the search bar. On top of this, we were able to utilize the brand market values for different watches which allowed users to see real time prices on specific watches they search for which is something we had in the project proposal.

We believe that our application has been successful regarding its usefulness. From Watch.io, users are able to add watches to their collection and gather the price for their watches to let users know how much value they hold. On top of this, users are also able to see other collections to see what users own and be able to rate it as well. These ratings will notify the user if their collection needs to be improved or if it is impressive to other users. On top of this, users are able to filter their searches, making the search function easier for customers to find watches they are looking for.

We also made a small change in the direction of our project, which is that we changed the name of our app from TimeConnect to Watch.io. We did this because Watch.io resembles a slightly more techy and entrepreneurial representation of our product and we believed that it would make our product more marketable.

In terms of the source of our data for our application, we did not change the source of our data as we still stuck with datasets from Kaggle, Chrono24, as well as another kaggle dataset for the brand information. Through these datasets, we were able to gather prices for each watch as well as more detailed information about the watch, such as the brand, brand information like the year founded, etc. The information provided was sufficient enough for us to complete Watch.io and it was enough information to show the user about their watch or other watches.

We did not change our ER diagram / table implementation as we believed that everything suited each other very well and made sense. The information that was presented was connected to another dataset in one way or another therefore we made almost no changes to our diagram / table implementations.

We added multiple functionalities to our website. To start off, we added a brand summary page with the average market values per brand. With this, users are able to see which brand has the more valuable or expensive watches as well as which brand has less valuable or cheaper watches so users can track which brand is most sought after. On top of this, we have a feature where our user collection dashboard highlights their collection as well as listing their collection. This allows users to show their collection to the public for other users to see and rate to showcase what they have and see how valuable their collection is. We also added a page to add watches to the users collection so users can build their portfolio of watches and track the prices and value of their items, keeping track of what goes on with their watches.

Our advanced database programs complement our application through having more advanced search / filtering capabilities. For our query, we find the average price of watches per brand which helps users to see on average which brand has more value and is more expensive. We also have a query where users can see watches that exceed the retail price. This lets the user see which watch is resellable for a profit and just how much value it has over the retail price. The more expensive it is, the more sought after it is and the more cheaper it is, the less sought after it is, clearly distinguishing what watches are valuable and which aren't. These are just some of the advanced database programs that we have in our application.

Some future work that we think that the application can be improved on other than the interface are to potentially add more filtering options. For example, if we found a review dataset for each watch, we could be able to filter the option in which users can see watches that surpass a certain number of reviews like a 3 out of 5 stars and such. On top of this, we could potentially add a marketplace connector in which users can see completed sale prices and a watch-watcher alert for when a specific model hits the users target price so users get notified and can be able to purchase it. This can be the same functionality as something like StockX where users can see when an item has a new lowest ask so users can purchase it for the price they desire.

Sohum

One of the trickiest technical challenges our team wrestled with was stitching together Firebase Authentication—great for secure sign-in but agnostic about our domain data—with our relational customer schema and the FlutterFlow front-end that consumes it. Firebase only guarantees a unique UID and, depending on how the user creates their account, sometimes nothing more than a display name. Meanwhile, our MySQL schema expects a fully populated customers table keyed by an auto-incrementing customer_id that foreign-keys into nearly every other table (watch_collections, transactions, social_graph, etc.). The pain point was mapping a freshly authenticated Firebase user to a persistent row in customers fast enough that the front-end could immediately issue authorized API calls that rely on that customer_id. We discovered early on that trying to lazily create the customer record on the first write caused race conditions, 401s, and null-FK errors, because many screens fetch data the instant the auth state changes. Our pragmatic—if imperfect—solution was to treat "sign-in" as a two-step process: after Firebase returns an ID token we synchronously hit /customers/register, insert a new row with the Firebase UID, capture the customer_id from the response, and store it in a global app-state singleton. All subsequent REST calls read that value and include it either in the path or payload, eliminating look-ups and letting us de-couple the SQL primary key from the auth provider's UID. For future teams, design a single canonical identifier up front or embed customer_id inside the JWT using custom claims to avoid this extra round-trip.

Ashit

For me, one technical challenge that I faced in this project was being able to connect my flask backend to the database which was hosted in the GCP. One thing I wasn't aware of when initially working on this was the fact that GCP only allows connections from authorized IP addresses. This caused issues where I couldn't figure out where the error was coming from. For

the future, I believe if there were more resources we could use in understanding GCP that would've made the process much smoother. Another thing I would like to address in the feedback is having a method of highlighting important information within campuswire posts as a lot of key information got lost within all the posts…

Rahul
One of the main technical challenges I faced in this project was implementing the stored procedures correctly. Initially, I wasn't sure how to structure the procedures to work cleanly with my Flask backend, especially since I needed them to perform specific aggregations like finding the most valuable watch, calculating total collection value, and identifying a user's favorite brand. Additionally, deploying the procedures to the database was trickier than expected. I encountered issues with SQL syntax, particularly because tools like Cloud SQL do not allow DELIMITER commands, which are commonly used when writing procedures locally. This required me to carefully adjust the syntax to fit the database environment and ensure each procedure executed correctly. Integrating the stored procedures with my Flask routes also involved modifying how parameters were passed and results were handled, making it a key technical hurdle that significantly improved the performance and maintainability of the application once solved.

Tony
For me, one technical challenge that I had encountered was working with GCP and just overall trying to make the advanced queries that would be useful in our application and just figuring out how to use GCP in general. Connecting to the VM was difficult in a sense that the website was hard to navigate, considering the fact that we had to use the in browser console to create the advanced subqueries and ensure that they worked. On top of this, sometimes the names on the original dataset was completely different from what GCP was telling us, which made it tedious to create these queries as well as ensure that the data was uploaded properly. One thing that really did help me was to follow the tutorials given on the side. From here, I was able to see how to connect to the VM as well as access the console to create these queries. On top of this, I was able to figure out how to view the dataset table names and column names which helped me properly write the advanced queries with the correct name so we have access to the data. From here, I was able to write the queries as well as make sure the data was uploaded properly by writing basic commands to display the whole dataset for whichever one I was wanting to read.

Division of labor and teamwork
Overall, we functioned effectively as a team and were able to divide work in a way that made us more efficient. For stage 4, Sohum took charge of the frontend while Ashit and Rahul worked on backend and integrating the API. Tony created the report and compared the current state of the project to the initial proposal to see how things were progressing. We regularly communicated and met in-person to figure out next steps to ensure we were on the right track. This was an exciting and rewarding experience and we are happy to have a finished final product to showcase to people.