Project Experience Report

Investment Grade

Hayden Jin, Erika McCluskey

Table of Contents

Project Objectives and Planning

Project Initialization

Our goal at the start of the semester was to create a cross-platform mobile application to be able to teach people how to invest. Our primary audience was young adults who were interested in starting but were not motivated enough to read a textbook, watch videos, etc. Our primary goal was to bring down the barrier of entry and friction of initially starting to invest. At the start of the project, we created many pieces of documentation to help us stick to our timeline, one of which was the milestone-based schedule. Initially, we planned out all of our tasks and proposed completion times within this document for the entire length of the project (8 months). As we progressed, we adjusted completion times as needed since certain items would take more or less time than we originally anticipated. We also utilized a Kanban board to keep track of tasks, as this helped to further break down roles and responsibilities. The first two months of the project were dedicated to planning. This included deciding on a tech stack, database planning, lo-fi / high-fi diagrams, and more. Planning was essential to us, as we have experienced the effects of a rushed planning process in previous projects.

Team

Our team consisted of Erika and Hayden who both worked as full-stack developers. We tried to split tasks up so work was divided as equally as possible, helping each other whenever there were issues. Erika did more of the work focusing on back-end infrastructure, while Hayden did more work on the content side of the application, including lessons, course ordering, etc.

Results and Conclusions

Tech Stack

For our tech stack, we chose to go with a React Native front end, with a variety of AWS services supporting our back end. We chose to go with React Native due to the fact that it is a cross-platform framework, enabling us to create an IOS and Android app with only one code base. We decided to use React Native over Flutter since React Native has been around for longer, with more documentation and form posts supporting it. Our back end consists of multiple services hosted on AWS, including Amplify, RDS, LAMBDA, and API gateway. Amplify is used for user authentication, and it handles sign-in, sign-up, email confirmation, password resets, and more. RDS is where we host our PostgreSQL database. We chose to go with a relational database mainly due to our previous experience and familiarity with relational databases. To facilitate communication between our front and back end, we used a REST API hosted by API gateway and LAMBDA. This enabled us to separate out our front end and back end, so changes to one would be contained and not affect the other. This also meant that the size of our application was smaller, and updates to only the back end would not require an additional app store release.

Final Result

Looking at our final product, we were quite happy with what we were able to create in such a short amount of time. We were able to implement all of our desired features while also being on time to get the application app store listed in time for project day. Our user interface and user experience were especially well received, with many comments saying that it looked very well put together.

The Good

One aspect that we executed very well was our planning and project management throughout the project. Due to our extensive planning before actual development, we did not run into any issues involving database issues, or having to majorly refactor models. Our low-fi and high-fi prototypes were also quite well thought out right from the start with our final product largely resembling what we envisioned initially. Another area that we did well on was project management, largely thanks to our Kanban board. By utilizing the board properly, we were able to keep track of what needed to be done, which tasks we were personally responsible for, and varying priorities. This enabled us to keep the project on track and on time even as the semester got more hectic.

Potential Improvements and Learnings

Future Improvements

One future improvement that needs to be implemented is securing the AWS API. Currently, we do not have any security setup within AWS. This means that anyone with our AWS URL can query our endpoints if they have the correct URL. We kept the API URL in a .env file and avoided committing it to source control to keep the URL private. In addition, it would be incredibly difficult for someone to get the correct UUIDs in the URL to execute commands which is why we chose UUIDs as our primary keys over self-incrementing ids. However, this is still obviously not enough security as we expand. We need to add authentication to all API calls. Since none of our data is sensitive data

and given the time constraints, configuring the security rules was not our top priority but it is something that should be done in the future.

As far as product-specific improvements, we plan to add additional features and lessons to the application. Some promising features include company scores based on how good of an investment it may be, and a modelling feature to perform a discounted cash flow calculation simply and easily. In addition, we plan to add more lessons to the application and more courses.

Lessons learned

One main lesson that we learned was that you truly do get what you pay for. This relates primarily to our cost from our stock data API, which is our biggest expense at $140 per month. Initially, we chose to go with a provider called StockData.org which offered the data that we required for $50 per month. Due to the low cost and limitations of the API, we had to store the stock prices within our own database. This meant that we had to create a LAMBDA service within AWS to query prices from the API, insert that data into our own database, then query price data from our database to finally display the prices on our application. There were a few issues with this process: the main concern that we had was the amount of data we would need to store, and the scalability of this process as we added more stocks to the list. Our other concern was that there were now multiple points of failure within this chain, as a failure of the stock API, LAMBDA, or database would all result in our application not receiving any data.

Luckily, an interruption caused by the stock API early on caused us to reconsider, and we ended up switching to a new data provider. This new API had no query limit which meant that we could connect it directly to our front-end application. This change

significantly decreased the complexity of our process, as we were able to drop the Lambda as well as all of the tables relating to stock prices within our database. While this was a great change, we unfortunately are still having some issues due to the supposed low price of the API service. After emailing to ask about a potential start-up discount, we were informed that there are additional fees related to displaying data to the public. This was not mentioned anywhere on the website, so it is a fair bit of false advertising on their part, however, we should have read the fine print. All in all, the primary lesson is that if all other offerings cost around $600 per month, there is probably something missing from the $50 and $150 offerings.

Another lesson that we learned was that promoting and getting people to use your product is way harder than anticipated. We were hoping to get at least a couple hundred downloads from project day, but we actually received less than ten. This goes to show that business-to-consumer (B to C) applications are a tough market to get into, and that advertisement is needed to get your product recognized.