Divvy Engineering Exercise

This write up explains my approach and completed work for the Divvy Engineering Exercise which I received on December 29, 2021.

# Setup and Review

Upon receiving the assignment, I forked the repository and created a new branch for my changes. I then began getting my development environment set up on my Windows machine using Docker, IntelliJ IDE w/ Elixir plugin, and Ubuntu (via WSL). I proceeded to review the existing code while also researching Elixir, Phoenix, Postgres, and GraphQL to better understand the application.

The initial setup and review took approximately 10-15 hours, spread out over about 5 days. I wanted to make sure that I understood the basics of how the application and associated technologies worked before diving in and making changes.

# Objective 1: Filtered Search

The first objective I chose to work on was implementing filtered search options for users, merchants, and transactions. I added “find” functions to each of the Users, Merchants, and Transactions modules. The find functions use a SELECT query with a WHERE clause to filter the search results. The query is then passed to Repo.all for processing.

Next, I added “find” queries to GraphQL by updating the schema and resolvers for users, merchants, and transactions. After implementing the queries, I tested their functionality manually by creating data and running queries from my web browser at localhost:8000/graphiql.

Working on this task helped me to better understand the relationship between Postgres and GraphQL. Specifically, it helped me understand how Ecto.Repo provides functions that interact with the Postgres database, which are then used by the GraphQL queries and mutations as defined in the Absinthe.Schema.

This task took around 6-8 hours, spread out over about 3 days. At this point I was still looking up a lot of syntax and getting used to writing in Elixir. I was also trying to match the existing code and documentation style as much as possible, spending the extra time to make sure that my code was readable.

# Objective 2: Add Companies

The next objective I chose to work on was adding companies to the application. I chose to work on this because I knew that it would be challenging and require a significant amount of new code to implement.

I started by creating new modules for companies that were similar in structure to the existing modules for users, merchants, and transactions. I also created an entity relationship diagram (ERD) to help me better understand the relationships between different entities in the database.

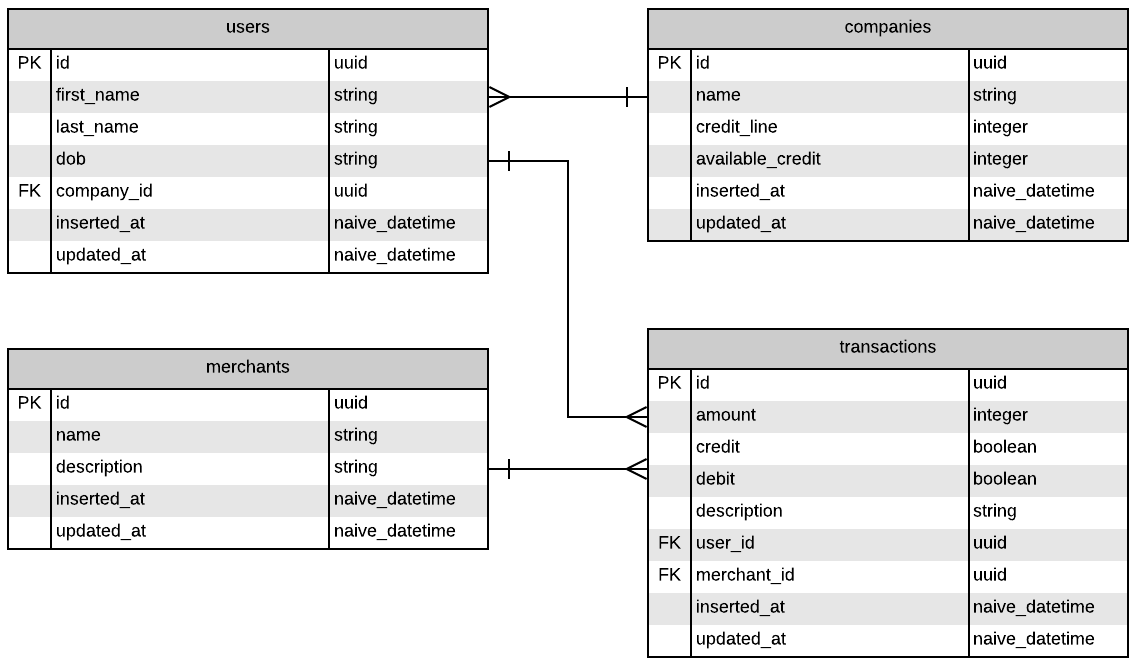


Figure 1: ERD for Homework application

Since users should belong to a company, I added company\_id as a foreign key to the “users” table. However, I chose not to add company\_id to transactions, since the information can already be obtained by joining the “users” and “companies” tables.

The main challenge for this objective was deciding how to link transactions to a company’s available credit. I decided to store available\_credit as an integer amount and update the value each time a new transaction is made. I chose to update the value inside of the Transactions.create\_transaction function. Before adding the transaction to the table, Repo.update\_all is used to update the available\_credit for the company associated with that transaction. I used update\_all in order to make sure that the operation was atomic, avoiding race conditions that would result from using a “read, modify, write” pattern.

While I did handle the main use case for adding a new transaction and updating available credit, I did not handle other use cases such as a transaction being updated or deleted. For an app of this nature, I think it would be better to simply prevent modification of transactions rather than trying to update available\_credit for every possible use case.

This task took around 10-12 hours, split between two days. I found this objective to be the most difficult to implement and test.

# Objective 3: Seed Database

The final objective I chose to work on was seeding the database. I considered both using the seeds.exs file and writing a .sql file that can be ingested by the database. I decided to write a SQL script because it would be faster and more compact when adding a large amount of data. I chose to include 20 companies, 100 users, 20 merchants, and 200 transactions in the seed data. I wanted to make sure there was sufficient data to perform various tests and model more realistic scenarios.

I generated my mock data using mockaroo.com by creating a separate .csv file for each entity (companies, users, merchants, and transactions). I then wrote a python script that would consume the raw data from the .csv files and generate a SQL script using that data. Generating the SQL script programmatically allows me to generate new scripts if a larger data set is needed or if the schema changes in the future.

I verified that the generated SQL script can be consumed by the database using the Postgres command line interface, using the command:

psql postgres -h 127.0.0.1 -d homework\_dev -f priv/repo/seed.sql

I was able to complete this task in one day, and it took me around 6 hours. I did also have to update the SQL script once after changing the schema to add companies. However, it wasn’t too difficult to generate new data and update my python script to create a new SQL script.

# Other Observations

I did find a bug with transactions where the changeset did not include the credit field. As a result, all new transactions and updated transactions would have the value false for credit, regardless of the values passed in by the mutations. Adding credit to the list of attributes in the changeset resolved this issue.

I also found it odd that credit and debit were stored in two separate fields. I would expect that a transaction would be either a credit or a debit (but never both or neither). I would prefer to store the credit\_debit information in a single field (possibly as an Enum), since a transaction will always be one or the other.

# Conclusion

I really enjoyed working through this exercise. It definitely gives me an idea about the types of problems that are solved by Divvy engineers. I was also able to gain a much better understanding of this type of application and the underlying technologies (Phoenix, Postgres, and GraphQL). I am confident that this understanding will help me to complete similar tasks in much less time in the future.