Task UFC-176: Refactor the notion of 'Account1/2' to Debits and Credits

Started: May 29, 2020

Completed: May 30, 2020

I really, *really* hate the decision that was made to have the relationship between a transaction and its accounts codified as `account1` and `account2`: it's entirely arbitrary and completely meaningless devoid of context and/or the code that actually defines the relationship (because there isn't any inherently).

As such, I think we need to just bite the bullet and refactor the entire codebase to use the notion of 'debits' and 'credits'.

This shouldn't be a crazy amount of work; just a bunch of find and replace. The biggest risk is dealing with the database migrations and making sure the data doesn't get corrupted (which isn't *too* risky considering our extensive backups).

But on that note, all our old backups become less useful since they'd require the migration to be run on top of them to be useful again. But that's why we just make new backups.

This is also the ideal time to do it since the Backend and Frontend (currently Redesign) services have been migrated to TypeScript and I'm just on the cusp of starting the UI redesign. So now's the time where we'll have the least amount of stuff to change.

As for *how* to rename account 1/2 to debit/credit, I did the following derivation.

Current Implementation

Account Flow

```
case Transaction.INCOME:
    leftAccount = account1;
    rightAccount = account2;
    break;
case Transaction.EXPENSE:
    leftAccount = account2;
    rightAccount = account1;
    break;
case Transaction.DEBT:
    leftAccount = account2;
    rightAccount = account1;
    break;
case Transaction.TRANSFER:
    leftAccount = account1;
    rightAccount = account2;
```

```
    break;
default:
    leftAccount = account1;
    rightAccount = account2;
```

Account Types

```
case Transaction.INCOME:
    account1Types = [Account.INCOME];
    account2Types = [Account.ASSET];
    break;
case Transaction.EXPENSE:
    account1Types = [Account.EXPENSE];
    account2Types = [Account.ASSET];
    break;
case Transaction.DEBT:
    account1Types = [Account.EXPENSE];
    account2Types = [Account.LIABILITY];
    break;
case Transaction.TRANSFER:
    account1Types = [Account.ASSET, Account.LIABILITY];
    account2Types = [Account.ASSET, Account.LIABILITY];
    break;
```

Debit/Credit Implementation

Income:

debitAccount = Asset
creditAccount = Income

Expense:

debitAccount = Expense
creditAccount = Asset

Debt:

debitAccount = Expense
creditAccount = Liability

Transfer:

Asset1 -> Asset2

debitAccount (to) = Asset2
creditAccount (from) = Asset1

Asset -> Liability

debitAccount (to) = Liability
creditAccount (from) = Asset

Liability -> Asset

debitAccount (to) = Asset
creditAccount (from) = Liability

Conclusion

Account1 = Credit Account
Account2 = Debit Account

And the account flows need to tweaked to be the opposite of the ledger. That is, since debits are on the left and credits are on the right, the 'flow' happens (for our purposes) from the credits to the debits (e.g. from an Income (Credit) to an Asset (Debit)).

Account Flows

```
case Transaction.INCOME:
    leftAccount = creditAccount;  (Income)
    rightAccount = debitAccount;  (Asset)
    break;
case Transaction.EXPENSE:
    leftAccount = creditAccount;  (Asset)
    rightAccount = debitAccount;  (Expense)
    break;
case Transaction.DEBT:
    leftAccount = creditAccount;  (Liability)
    rightAccount = debitAccount;  (Expense)
    break;
case Transaction.TRANSFER:
    leftAccount = creditAccount;  (Liability, Asset)
    rightAccount = debitAccount;  (Asset, Liability)
    break;
default:
    leftAccount = creditAccount;
    rightAccount = debitAccount;
```

As such, to model the 'From -> To' relationship, the account flow is literally always this:

leftAccount = creditAccount
rightAccount = debitAccount

Account Types

```
case Transaction.INCOME:
    debitAccountTypes = [Account.ASSET];
    creditAccountTypes = [Account.INCOME];
    break;
case Transaction.EXPENSE:
```

```
    debitAccountTypes = [Account.EXPENSE];
    creditAccountTypes = [Account.ASSET];
    break;
case Transaction.DEBT:
    debitAccountTypes = [Account.EXPENSE];
    creditAccountTypes = [Account.LIABILITY];
    break;
case Transaction.TRANSFER:
    debitAccountTypes = [Account.ASSET, Account.LIABILITY];
    creditAccountTypes = [Account.ASSET, Account.LIABILITY];
    break;
```

Tasks

- Update the models in Redesign/Frontend
- Update the usage of the models throughout Redesign/Frontend
- Update the schema for the models in the Backend
    - Will I need to make multiple copies of the schema now? Because the old migrations will need to have the old column names in place to be accurate.
    - I think so. So what we'll have to do is this:
        a. Either we can have the old schemas in place forever along with a new versions that work with each new migration that takes place. As in, we'll have to keep a history of all the schemas as they change.
            - Pros: Keeps an actual history of all database changes; allows old backups to still be useful since they can be migrated.
            - Cons: Introduces complexity in terms of how schemas and migrations are handled.
        b. Or we cheat. We have two versions of the schema (old and new) along with the migration to rename the columns, but once we've applied the rename migration to the production database (which doesn't even have to happen on production... can do it locally or on a branch environment for all it matters), we can drop the old schema and the rename migration entirely and just use the new schema.
            - Pros: Keeps the schema history clean and tries to keep database complexity down over time.
            - Cons: It's probably not the 'best practices' way of doing things; more risky since the old backups then become *truly* useless, with no way to migrate them to the new schema (short of re-doing this whole procedure, which kinda defeats the point).

    - Decision: Well we have to do 1. regardless, since it's a prerequisite for 2. But since the app is still in it's infancy (with me being the only user), I think I could probably get away with 2. because the old backups aren't particularly useful. And they can always be migrated in an emergency anyways so...
    - WAIT JUST A MINUTE. Doesn't sequelize store data about the migrations in the database itself? If that's the case, in order to maintain integrity, we *have* to *not* do 2. If we do do 2., then we'll end up with a database that has a history of migrations that then no longer exist, which could throw things for a real hoop.
    - OK, so I took a look, and there is indeed a `SequelizeMeta` table in the database. However, all it is is literally a list of the migrations that have been run against the database. Just a single 'name' column, the name of the migration, and a of rows. So... maybe we could just delete the existence of the rename migration?
    - Decision: Try to do 2. and see what happens.
    - WAIT, SHIT
        - So... I just realized that we can't exactly go renaming the accounts in Backend and Redesign... without also doing so in Frontend. :facepalm:
        - So... options? I see two. Either actually go through with renaming it *everywhere* (including the Frontend), or spin up a separate Backend service just for the Redesign service.
            - For renaming it everywhere, we get the option of being able to do Decision 2. from above (cheating the migration system).
            - However, for spinning up a separate Backend service, we can only do Decision 1. Because we'll need to have the migration code in place in the future to migrate whatever data diverges between old production and new production. *Then* we can go ahead with Decision 2.
            - Honestly, I kinda don't want to have two separate Backend services running, if only because I want all production data to be served to the Frontend *and* Redesign services. But it's gonna to be pretty tedious (or just a couple commands...) to rename everything in the Frontend.
            - Yeah, if I'm going to do this rename, it's gonna happen in *all* of the current services. I think it'd be more work to spin up and maintain a separate one than to just rename everything in the Frontend.
- Create a migration to rename the two columns

- Update the usage of the models throughout the Backend
- Test the migrations and updated code on a backup of the production database
- Create a migrated database that doesn't have the rename migration and test it
    - OK, having now gone and done all the renaming, I think it's fine to go with Decision 1. and just leave the old schema in place. I think we can get away with this because of the fact that we didn't actually touch any schemas; we just changed the foreign keys, which are separate from the schemas. As such, we just needed to change the references and now the only things that reference the old names are the first migrations. Which I think is fine for now.
    - Maybe some time in the future, we'll compress everything and remove the renaming migration.

Post Merge Process

Once we merge, we'll have to restore the migrated backup to production. The migrated backup consists of the latest backup (May 30, 2020 at the time of writing) that has had the renaming migration applied to it locally then dumped back to a sql dump file.

The process for restoring it will be as follows:

i. After merging the code and having it deployed, follow the backup restoration process outlined in How to Restore a Postgres Dump.

ii. Once migrated backup has been restored, we now need to reset the Backend pod to pick it up and...

OK, I just realized we *don't* have to do this. That's the whole reason we're going with Decision 1. and not Decision 2.: so that it can migrate the data itself! We're keeping the migrations around, so might as well let them work as intended.

Ignore this entire section.