Story UFC-217: Account Details

The following document will go over the design of the Account Details view, as outlined in Story ticket UFC-217.

Acceptance Criteria

For reference, the following are the acceptance criteria for this ticket:

- I want to see the account's current balance (for the given date range).
- I want to see the account's name and interest.
- I want to view a chart of the account's balance in the current date range.
- I want to see the account's transactions, filtered for the current date range.
- I want to be able to Edit or Delete the account.

Design Brainstorming

So this story is the first introduction of charts into the redesign. Literally everything else about this story is trivial (some text, some buttons, re-using some transaction components, etc), but the chart in of and itself will take the most work.

Why? Cause styling. Getting the data for the chart should be fairly simple. but styling it will be a pain in the butt. Hopefully, with Victory charts, they're customizable enough that we can style them to match some inspiration that we find.

Actually, I take back that everything but the chart is trivial. I completely forgot that the Transactions table is gonna need modifications to support a running balance for the account. So... *two* things that are non-trivial.

Balance Chart

Since we kinda skipped over designing the charts during the design phase, that means that we're gonna need to do it now. As such, here's some design inspiration!

Design Inspiration

- Analytics Chart designs, themes, templates and downloadable...
- Podcast Dashboard
- Report Builder for NLP Platform
- Cryptocurrency Dashboard for Blockchain Exchange Platform

- Analytics (Dashboard UI Kit 3.0)
- UI Movement
- Grafana
- Revenue Dashboard - Baremetrics Demo
- Monthly Recurring Revenue - Baremetrics Demo

Articles

- Data-heavy applications: How to design perfect charts
- 10 rules for better dashboard design
- Material Design
- https://uxdesign.cc/designing-charts-principles-every-designer-should-know-5bd3969a0150
- https://uxdesign.cc/designing-charts-principles-every-designer-should-know-part-2-ce1e06af56fc
- https://medium.com/@AliAftab/how-to-make-simple-line-chart-using-victory-in-react-in-5-minutes-d9c544e7e7ec

Data Formatting

Having done some investigation and research, it seems like a big part of making these charts effective is doing something we didn't do before: aggregating data based on the date range.

For example, when viewing just a month's worth of data, we can show days on the x-axis, and each data point is the account's accumulated balance on that day (at least, for assets and liabilities). But when viewing something like a year's worth of data, the data should be aggregated down so that each point is represents a month.

This will (hopefully) fix the horrid performance of something like the old Net Worth chart, where each point was present and it was *slow*.

Between a year and month, there'll have to be a point where we aggregate down to weeks; this'll probably at several months of data.

OK, let's make this logic concrete (i.e. which unit to aggregate on for data points), so that I know what I'm implementing:

- Less than or equal to 2 months (60 days): Use days
- Greater 2 months (60 days) and less than or equal to 6 months (180 days): Use weeks
- Greater than 6 months (180 days): Use months

Since we support Custom date ranges, but *not* specifically X months, we need to define a 'month' as a certain number of days. Let's just go with 30 days. So every instance of 'month' above can be replaced by '30 days'.

Then we'll need a good algorithm for generating these intervals. That is, we'll need to be able to account for a part of a week or a part of a month.

Finally, we'll need to make sure the tooltip dates (and x-axis dates) get formatted correctly. Here's how they should work:

- Using days:
  - x-axis: Days (e.g. `Jul 3`)
  - tooltip: Day
- Using weeks:
  - x-axis: Days
  - tooltip: Range from start to end date of the week (e.g. `7/3 - 7/9`)
    - Decided that I didn't like this format and just went with `Jul 3 - Jul 9`.
- Using months:
  - x-axis: Months (e.g. `Jul '20`)
  - tooltip: Month

Update: OK, so I just got week level aggregation working. Having tested it over my complete transaction history, it seems to perform well enough.

However, what I'm sceptical about is how to go about implementing month-level aggregation. I don't think, from a UX perspective, that it'd be very good to implement it as a naive '30 days' aggregation. Because users would expect the data to line up with the months on the axis.

So that means that our aggregation would have to only aggregate data in whole months, and then deal with partial months at the start/end of the range.

Aside: Another thing to note is that I think we could get away with increasing the cutoff for when we switch from weeks to months. I'm thinking greater than 1 year, we switch to months. Why? Because, like I said above, week aggregation seems quite performant, so I think we should use it where it works.

Back to month aggregation: OK, so what would the algorithm for month aggregation look like?

i. Set current date to start date.

ii. Reduce over transactions from current date to end of month.

iii. Set current date to start of next month for next loop.

iv. Loop until we hit the last month.

v. In the last month, reduce over transactions from current date to end date.

Data Extraction

OK, so how are we actually going to get the data to this chart? Well first off, we have to content with the fact that the input data to getting balance values is twofold: the date range Context, and the ID from the URL.

Since we're dealing with a date range again, that means that we *have* to go the hook route for querying the data. Additionally, it'll require taking in the account's ID. I think we can just take the ID in as a prop to the chart, since the Accounts scene can pass it down from it's `Route` `match`.

OK, so we have our inputs. Now how do we actually form the data?

Well, we know what the final data has to look like. Let's take a monthly date range as the example. For a monthly date range, there must be a data point for each day in the month (date range). For each data point, there must be a 'balance' and a 'date'.

- Since we know that we need one data point for each day, those can be generated, *not* taken from something like a list of transactions.
- However, for the balance, we'll need to do 2 things:
  - Get the account's balance at the *start* of the date range (more specifically, the day before the date range).
  - Then get all of the transactions inside the date range that belong to the account.
  - Then take all of these transactions (sorted ascending by date) and add the amount's to the balance to calculate the account's balance at each day.
    - Note: This might have been implied before, but I'm making it explicit now: **account balances are end-of-day**. That means, for example, that the start day's balance is equal to the (start day - 1) + amount of transactions on start day.

OK, so to re-iterate, what we need to query up the transactions between the date range and the account balance before the date range.

- We can re-leverage the `selectAccountBetweenDates` selector that we used for `useDateRangeAccount` to select the before-date-range balance.

- We can then use the regular `selectTransactionsBetweenDates` selector to get the transactions in the current date range. We'll need to then filter them to get the one's that apply to the current account.
- Finally, we can use the `balanceReducer` to calculate the balances.

An implementation detail that I thought of was whether to iterate the transactions or the dates to do the summing.

- I think it's easiest to do iterate the dates. But I think that means that we need to index the transactions by date again so that during each loop of the dates, we can:
  - Check if there's any transactions on the date.
    - Add their amounts to the previous date's balance if there is and set the current date to the new balance.
    - Set the current date's balance to the previous date's if there isn't.
- What if we were to iterate the transactions? We know that when we get back the list of transactions, they're already sorted by date. So then if we iterate them, we'd do the following:
  - Take the transaction's date and lookup the balance in the data points.
  - Add the transaction's amount to the balance in the data points.
  - When we go to the next transaction, backfill any data points that didn't have any transactions with the previous date's balance?
- Hmm, I don't like that as much.
- What if we were able to get the transactions from the store as a date index structure, instead of an array? That would (marginally) speed up date iteration.
  - The only real decision here would be whether to return them in an actual date index (i.e. the same structure that the store uses, with year → month → day indices) or in a flat index with a string date index (i.e. "year-month-day" → [ids]).
  - I *think*, for the purposes of the chart data, it'd be easier if it was just string dates → [ids], since we're using timestamps/date objects for the data points (strictly speaking, we *must* use date objects in the final data format, since that's what Victory expects, but we can use whatever as an intermediate format).
  - OR, we just start with the naive implementation of re-indexing it by date at the hook level and optimize it later.

OK, so then I don't think we're adding any new selectors. So what's gonna be the name of this new hook? `useAccountBalanceChart`? Yeah, I guess we don't need to have this be a top-level hook, so it can just be in the hook logic of the chart itself.

Running Balance Transactions Table/List

OK, so how are we gonna modify the Transactions table/list to support running account balances? Let's discuss each component individually, since they're *slightly* different:

Table

So what's the interface we want to have to enable this functionality? Well, what information are we going to need?

Because the rows are completely independent from the table (i.e. they only receive the transaction ID), it's gonna be be tricky to calculate a running balance. Obviously, we need to pass in a starting balance to the table, to set a base-line for this date range's account balance (obviously, this is 0 for income/expense accounts).

But then what? Because the table rows are completely independent, at best they can calculate the balance + themselves.

But what if we just calculated out the running balance for each transaction at the table level, then passed in the right balance for the right transaction to the right row, when we generate the rows? That way, the rows wouldn't have to do any calculations themselves.

Yeah, I guess that'd work.

Update: Oh boy, have I hit a snag.

See, when we generate the running balances, they can't just be generated using the filtered transactions list; they *have* to be generated using the unfiltered (i.e. un-paginated), *un-sorted* (i.e. sorted by date *ascending*) transactions. Otherwise, the running balance is wrong.

So I guess what we have to do is generate the running balances on the "date ascending, un-filtered" transactions, and then generate an `ID → running balance` map instead of an array. This way, when we generate the rows, we can just use the ID to lookup the balance.

Obviously, the running balance column will be nonsensical looking if the user sorts by anything other than date, but that's kind of a given. At least the data will be accurate.

List

TODO

Component Breakdown

Atoms

- [modification] Button (add a 'secondary' variant for the grey background buttons)

Molecules

- [modification] TransactionsTableRow (add option for running account balance)
- [modification] TransactionsTableColumnHeaders (add option for running account balance)

Organisms

- AccountBalanceChart
- [modification] TransactionsList (add option for running account balance)

Scenes

- [modification] Accounts (modify it to display the Account Details; will need a lot of media query trickery to get it to work well between mobile and desktop)

Tasks

- Build the `AccountBalanceChart`.
- Modify the `TransactionsTable` and `TransactionsList` to support a running balance.
- Build the Account Details view into the `Accounts` scene.
- Make sure there is an 'invalid account' state for the Account Details.