Story UFC-191: Transactions List

The following document will go over the design of the (basic) transactions, as outlined in Story ticket UFC-191.

Acceptance Criteria

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

- I can click on 'Transactions' in the header to go see an overview of my transactions.
- On the 'Transactions' page, on desktop, I can see a *table* of my transactions with the transactions' type, date, amount, description, and accounts.
- I can see that only a certain number of transactions are shown at a time.
- I can see that I can change between pages of transactions that are shown.
- I can sort the table ascending or descending by clicking on a column.
- The table is sorted, by default, by date.
- On the 'Transactions' page, on mobile, I can see a *list* of my transactions with the transactions' type, date, amount, description, and accounts.
- I can still see that only a certain number of transactions are shown at a time.
- I can still see that I can change between pages of transactions that are shown.
- There is no way to sort the list; it is only sorted by date.

Design Brainstorming

Table Design

OK, so the hardest part of this ticket is definitely getting the `TransactionsTable` just right. There are a couple different stages of responsiveness that the table needs to deal with before it can change to the simpler `TransactionsList`, not to mention that tables in general are more complicated than lists. Especially because we'll have to learn the new API for `react-table` v7, which is all headless.

[I've just completed UFC-223, the mobile TransactionsList; time to design the table]

Just a thought: I think I completely forgot that we should only be passing the IDs of objects around when dealing with lists. For example, right now the `TransactionsList` (and the `AccountsList`) are receiving the full object data for rendering. But we should change it so that the map operation in `useMemo` is only reliant on the IDs. Otherwise, we'll eventually run into rendering performance (i.e. once we start having to edit transaction data live).

OK, back to the table design.

Should we use react-table again?

So I've briefly looked over the docs for v7 of `react-table`, the version ahead of what we used in the old Frontend that is now completely headless. And.. honestly, I don't think it has enough benefit for us. I legit think it'd make more sense for us to just code the table ourseleves.

If we look at what v7 offers, it's really just a bunch of advanced table logic and no UI. However, it still forces to construct our table in a certain way (since we need to be able to map over the rows/cells/etc to use the props generated by `useTable`), and it isn't really the way I want to build the table.

This might end up being misguided in hindsight, but I want a completely separate `TransactionsTableRow` component . That way it can be tested (and storied) in isolation. However, due to the limitations I stated above, this seems like it'd be harder to do. This is the first reason I'm apprehensive of adopting `react-table` again.

The second reason is probably the more logical one: we don't need all of the functionality that it offers. Really, the only thing that we'd be using immediately is the column sorting. However, of all the table functionality we could need, that's like the *easiest* to implement ourselves.

In terms of other table functionality, we're already doing the pagination ourselves. I don't foresee a need for grouping columns or doing or modifications to the columns (re-ordering them, hiding them, resizing them, etc). We're gonna be doing the filtering all outside of the table component anyways, so we don't need that. Row selection will also be custom. We won't have expanding rows. Etc.

As such, I think it's best we just start out building the table ourselves. We can re-evaluate in the future if we ever get to a point where it'd make more sense to go with a library like this again. But I think the flexibility we gain by building it custom will be worth it in the long run.

Row Design

So, like I mentioned above, I want to have a completely separate `TransactionsTableRow` component that encapsulates all of the logic and UI needed to render a single row. All it should need is a transaction ID and it should be good to go.

Currently, we have 3 different row designs, plus a fourth for editable cells.

The first design is the most generic: each piece of data is one column.

The second design is the stop-gap responsive design before the layout would change to the `TransactionsList`: The accounts have been collapsed into a single line beneath the transaction, just like the `TransactionsList` items, which means that a single column represents three pieces of data (accordingly, the header for the column -- `Description, From, To` -- must support sorting on all three pieces of data, independently).

The third design is a custom design for the Account Details view: really, we're just adding on a `Balance` column to the end of the second design. Why the second design? Because the Account Details view is more constrained horizontally. Theoretically, however, a user could have the room for the full width table design (the first design) *and* the `Balance` column, so we should really model adding this column as just *adding a column*; nothing should tie the column to the second design.

- One thing that gets tricky with this `Balance` column is *how* we calculate the data for it. Because the row component only receives the ID of the transaction, we have a couple of options:

    1. Don't restrict ourselves to only passing the ID! One option would be to have the Account Details scene (at some level) calculate the balances for each transaction and then pass them along to the rows as a prop.

        - And since this balance value is just a number, memoizing on it is *much* easier.

    2. Ya know what, my second option was gonna be something like "have the row connect back to the store to derive the balance itself", but I feel like that's *really* overloading what the row should do. Not to mention that feels like something where a *lot* of calculations would be duplicated, since each row would have to do something like "lookup the opening balance, find the balance at the start of the period, then look up *all the other transactions* for the period, *then* calculate the rows balance based on all of that data". In reality, all those steps should be done at the Account Details scene level.

- So, looks like we have our solution to the `Balance` column! It'll just be a dumb value pass through the row conditionally renders, and *calculating* the value will be up to the calling scene.

So, after all that, we now just have to figure out how to work with actual HTML tables! I haven't actually done much of this before, so it'll definitely be a learning experience.

I think the most 'complicated' thing will be how to model the second design (having the description + accounts in one column) in a semantic manner.

- Having done some research, it seems like something like a `colspan` that might help with this.

Also, I came across this article: Flexible data tables with CSS Grid on using CSS Grid with a HTML tables.

- Basically, the technique described is to use `display: grid` on the `table`, but then use `display: contents` on the `thead`, `tbody`, and `tr` elements so that the grid affects the `td` and `th` (i.e. the cells) directly. This way, `grid-template-columns` can be applied with a repeating `minmax` property to have responsive columns that have a min/max width.

- However, for our purposes, we can probably skip applying `display: grid` to the `table` and apply it directly to `tr`, which really means that we'd do the layout at the `TransactionsTableRow` component level with `grid-template-columns`, and then the `grid-template-columns` layout for the column headers (which would done at the table component level).
  - Note that all of the above is for the first design.
- For the second design, we'd have to bring in some breakpoints and (presumably) a `colspan` to both change the order of the columns being rendered (at the table level) and combine the `Description`, `From`, and `To` columns together.
  - Actually, it seems like `colgroup` would be the more fitting element to use. We'd use it to declare each of the columns, with a column that spans multiple columns.
  - So for the second design, the `colgroup` would look like this:

    -
      ```
      <table>
        <colgroup>
          <col> // This is the Date column
          <col span="3"> // This is Description, From, and To
          <col> // Amount
        </colgroup>

        <tr>
          <th>Date</th>
          <th>Description</th>
          <th>From</th>
          <th>To</th>
          <th>Amount</>
        </tr>
      </table>
      ```
    - For reference on some `colgroup` examples: <table>: The Table element
    - Then we'd have to use CSS to finagle the Description, From, and To columns to look like one column. We'd probably also have to use the `after` element to add in the commas. Although we could also do that in HTML with some `aria-hidden` elements.
  - BTW, the whole reason we're looking into how to do this properly is so that the table is as accessible as possible. As long as the resulting markup looks good, we can do whatever to style it to look how we want.

    - At the row component level, we'd probably just have three consecutive `td` elements for the Description, From, and To cells, but then we'd lay them out specially to meet our design needs.
        - This way the markup is semantically correct for screen readers, but it looks how we want it.
        - We, presumably, have to be careful about using extra `divs` to wrap things when trying to align the description/from/to, but we'll get there when we get there.

Filtering Design

The other 'hard' part of this ticket is deciding on a design for how we're gonna handle all of the filtering of the `TransactionsTable` in the context of the transactions scene, but also in the context of being able to drop the table anywhere else (in particular, Account Details). And when I say 'filtering', for this ticket, that specifically means pagination, but in the future that'll mean things like type filtering and date range filtering.

There are a couple of different designs that I've had in mind, and it basically comes down to "put it in the store" or "don't put it in the store".

For the store approach, this would mean something like having the 'pagination' data in the store so that the `PaginationSwitcher` can just update the store and then the `TransactionsTable` can connect directly to the store where selectors can filter the transactions data (in this case, by page).

The 'not store' approach would involve either something simple like local state at the scene level (since the scene = table + pagination) such that the scene would connect to the store to pull in all of the transactions, then it would filter them based on its internal state for pagination (which it would pass to the switcher).

- The alternative to local state that I've been thinking about is to finally dip my toe into using React Context. A bit more lightweight than using the store, but certainly heavier than local state at the parent level.
- OK, having looked into how to use Context a bit more, it *looks* like this is *feasible*, just certainly not the most optimal way to go about it.
- For the record, the way I was thinking of using Context was to have something like a global 'pagination context' (global in the sense that that's how Context is defined; it's not tied to a single component). Then different scenes could have make use of separate instances of the Context Provider, while something like the `PaginationSwitcher` would just consume the context, not know under *which* provider it is. This mostly works, it just feels like the mechanisms for updating the Context value from the consumer (pass a setter as a piece of the Context value) is a bit janky, and not really how it feels like it was intended to be used.

  - OK, apparently my gut feeling on this being janky was *right*, because after reading the article linked below, it seems like a good practice is to have *2* contexts: one for the value, and one for the 'dispatch' (or setter, in this case).
  - So, I don't feel so bad about it.
- Of course, another downside of Context is that it makes components less reusable, but eh...
- Here's a really good article of using Context well: How to use React Context effectively

If we remember the constraints we're working under (namely, as much logic as possible in the store), then it'd seem like doing the store approach is a no-brainer, but because we want to be able to reuse the table component in multiple scenes (again, most notably Account Details), it's less clear cut since then we'd have to decide to either have one piece of store state for all transactions table pagination (i.e. it's shared by all instances but maybe reset whenever the route changes to 'simulate' independent state) or a separate piece of store state for each instance of the table, where we'd then have to configure each instance of the table (+ other components like the switcher) to use different pieces of store state.

You know what? Context and hooks are reusable in react native, and I think using the store here is really overkill. So I think, to try something new and see how it works, we'll go with the Context approach. Here's the flow:

- Connect table to store to get transactions.
- In scene (which houses the table + switcher), store the pagination state locally and provide it using the Context Provider.
- The switcher consumes the state using Context, and can update it.
- The table consumes the state using Context, and it uses it (inside `useMemo`) to filter the transactions it got from the store at the component level.

Hmm, but then I think that maybe we shouldn't connect the table directly to the store, because we *know* that we're gonna have to change where it gets it's data (Account Details connection is different than Transactions scene connection). As such, I think the table should take transactions in as a prop, *on top of* all of the filtering contexts, so that the table can encapsulate the filtering logic.

With this setup, dropping a transactions table somewhere else just means wrapping 'somewhere else' in the Context Provider, using the table component normally and passing it whatever transactions it needs.

For the future, this means that things like Date Ranges and Type filters would be separate contexts.

We'll see how this plays out and if it's any better than having independent store states.

Pagination.. through URL?

A thought occurred to me: should the pagination state be stored in the URL?

The main reason we might want to do this is to support page navigation using browser buttons -- would provide a more native feeling experience.

However, I feel like this might complicate the pagination implementation. First of all, what URL would make sense for the transactions in Account Details? `/accounts/:id/page/:page`? It's not the cleanest thing.

Although I just thought of something: could we use hashes (e.g. `#page-1`) ? That would give us a more general interface than a URL.

If we go this route, that means we can basically throw out the reducer + state used for the pagination state. Instead, it's all just stored in the URL. Then, the Provider can be a component wrapped with `withRouter` that checks for hashes and updates the Context accordingly.

You know, because the Provider completely encapsulates the state (+ reducer), I wonder if we could have swappable Providers, if only for testing. Could have the one Provider we currently have (with reducer + internal state) and then have a separate Provider that maintains state through URL hashes.

The only problem with this schema is that the hook for using dispatching would have to be a bit different. Because `dispatch` wouldn't exist with the URL hash scheme, we'd have to move the callback generation up a level to the Provider, such that the `Dispatch` context returns the callbacks directly, instead of being derived in the hook.

I like it! This seems like it could work.

Re-thinking Context and Providers

So I just started implementing the Transactions scene -- got the TransactionsList and the PaginationSwitcher in and changed TransactionsList so that it just receives its transactions as a prop and the Transactions scene is connected to grab them.

However, once I plopped the Pagination provider in, I had a sudden 'oh shit' moment when I realized that, since the Provider is rendered *by* the scene, the scene itself can't access the Pagination information.

So then my thoughts were "ok, so who's gonna do the filtering of the transactions?" In my mind, we have a couple of options:

i. Wrap the Transactions scene itself with the Pagination provider and do the filtering at the scene level so that the TransactionsList receives the filtered transactions.

ii. Leave the Pagination provider rendered by the Transactions scene and do the filtering in TransactionsList (with the logic encapsulated by a hook).

Some of the things I thought while trying to come up with a decision on the solution where:

- How will this decision play out once we have to put the TransactionsList in a different scene?
- Who's responsibility should it really be to do the filtering?
- Since we're gonna have a separate TransactionsTable, it'll also need the filtering. How will it get the filtering based on this decision?
- If we do pagination filtering one way or the other, how are we going to handle the other types of filtering (date range, searching, type filtering)?
    - This was the deeper, and more important, question IMO.
    - I think the answer here lies in the fact that these types of filtering are *not* the same:
        - Date Range filtering and Searching will *have* to be hooked into the store. They have no choice in the matter because of the fact that they have store-based indexes.
        - As such, these two types of filtering work differently from type and pagination filtering: they don't take the a list of transactions and filter it down -- they use an index to lookup exactly the transactions they need.
        - As such, their results from the store are already filtered.
        - As such, they don't really fit into this Context based paradigm that we've developed with Pagination.
        - As such, it makes sense to keep Pagination (and eventually, Type) filtering in this localized Context/Provider paradigm.

Given the above, I think it makes more sense to create one common hook (say, `useFilteredTransactions`) that calls into the Context hooks (in this case, Pagination's `usePaginationState`) to filter a list of transactions for consumption by the TransactionsList (and TransactionsTable).

That is, the filtering logic should be *inside* the List/Table for Pagination (and Type) filtering, but *outside* the List/Table (at the store/selector level) for Date Range filtering and Searching.

Let's go over the three (current) use cases for the TransactionsList/Table:

i. Transactions scene:
    - The scene is connected to retrieve the entire list of transactions (in the future, it will be connected to get the date range + search filtered transactions).
    - The scene passes the transactions to the List/Table.

- Since the scene is rendering the Pagination (and Type) providers, the List/Table can call the `useFilteredTransactions` hook to filter the transactions without the List/Table needing to be passed anything else.

ii. Account Details scene:

- The scene is connected to retrieve the account's list of transactions (which, again in the future, will be connected to get the date range + search filtered transactions).
- The scene passes the transactions to the List/Table.
- Since the scene is rendering the Pagination (and Type) providers, the List/Table can call `useFilteredTransactions`.

iii. Recent Transactions list on the Dashboard scene:

- The scene is connected to retrieve the recent transactions.
- The scene passes the transactions to the List/Table.
- Since the scene *is not* rendering the Pagination (and Type) providers, the List/Table can call `useFilteredTransactions` but nothing will actually be done -- the transactions will just pass through unfiltered.

I think this lays out a good system for determining which filtering happens where and why.

Component Breakdown

Atoms

- TransactionTypeIcon (the four variants of the transaction type icon)

Molecules

- PaginationSwitcher (IconButton + TextField)
- PaginationSummary (TextField)

Desktop

- TransactionsTableRow (IconButton + table elements)
- TransactionsTableColumnHeaders (table elements)

Mobile

- TransactionsListItem (a ListItem bound with the content of an transaction and connected)

Organisms

Desktop

- TransactionsTable (TransactionsTableColumnHeader + TransactionsTableRow + …)

Mobile

- TransactionsList (TransactionsListItem)

Scenes

- Transactions (OverlineHeading + TransactionsTable/TransactionsList + PaginationSwitcher + PaginationSummary)

Tasks

- Build mobile TransactionsList.
- Build desktop TransactionsTable.