Story UFC-192: Transaction Creation

The following document will go over the design of XXX, as outlined in Story ticket UFC-XXX.

Acceptance Criteria

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

- I can click a button to choose to create a new transaction that sends me to a form to fill out.
- I can fill out the form to have a transaction created.
- I can autocomplete transactions using the description input.

Design Brainstorming

The account select inputs should be actual `select` inputs on mobile, because they get a better native experience. On desktop, however, we should use our custom `AutocompleteInput`.

For the date input, I'm thinking we just go with the default, native browser date picker. I mean, I know Safari doesn't even *have* one (at least, on Mac), so they'll be left out, but eh. Screw em. Chrome has a better desktop date picker now and the mobile ones are generally pretty good too. So, why not?

Autocomplete Input

So I've been thinking a bit about the autocomplete input. Specifically with how it functions in relation to the store.

Currently, the input takes.. input, dispatches it to a saga, which performs a search against the index in the store, and then the results of the search are stored in the store for the input to then retrieve to display as autocomplete suggestions.

The problem I think this setup has is that the search results have only one place that are stored in the store, but we know in the future that there are going to be multiple sources that perform searches. As such, in order to not have to get into trying to make this one piece of search results state work for multiple sources (and creating multiple pieces of state - one for each source), I figured we needed to take a look at other options for searching.

The one that immediately came to mind was doing what we did for Pagination - using Context with a custom provider and hooks.

In this case I was thinking of something like this:

- Custom provider that connects to the store to perform the searches and store results in its own internal state. It then passes the results down through the Context.
- A state hook that exposes the detach results.
- A dispatch hook that exposes the function for performing a search.

This is the core of the idea.

An addition that could be made on top of this is to have the notion of a 'selected result'. This way the autocomplete input can just mark a result as selected and then the Transactions Form can pick up this selected transaction to populate the form.

So for the context of the Transactions form, the complete flow would look like this:

- Search Provider wrapping the Sidebar Transactions Form (i.e. at the scene level).
- Autocomplete input uses both the state and dispatch hooks.
  - State hook for getting the results to display them to the user.
  - Dispatch hook for executing the search with the users query.
  - Dispatch hook for marking a result as selected.
- Transactions form itself uses just the state hook. It looks up to see if there's any selected transaction and populates the form once there is.
  - Hmm, slight problem here. If the state *hook* is what gives the form the selected transaction (which is just an ID), how will the form perform the store lookup to get the full transaction?
    - Could use a selector hook to perform the lookup.
    - Or could have the Search provider do the lookup and store the complete selected transaction.
      - But.. then how does the Search provider do the lookup? The marking of a selected transaction doesn't go through the store (only internal state), so it too would have to use a selector hook to do the lookup.
    - And the reason this is even worth discussing is because I've mostly been able to avoid using the react-redux hooks, since I prefer decoupling with connect functions than hooks.
      - But I guess, from a reusability standpoint, it doesn't really make a difference? However, it does introduce more complexity since there's now two places that a component can access the store.

        - Wait, wait, I got it! Instead of the search *results* being just a list of IDs, it can be the full list of transactions. So the Search provider performs the search and the gets back the results as full transactions.

    - Hold up, what does it mean for the Search provider to "perform the search and get the results"? In order to perform the search, an action is dispatched. This action is then picked up by a saga, which performs the search. But then how does it 'return' the results to the provider? By a piece of store state? So we're not really solving anything here then? This is just all a waste of time?

        - The alternative, as far as I can come up with, to storing the results in store state is to use a selector to perform the search (instead of a saga). Since, ya know, selectors are the 'views' of our redux 'database'.

        - However, this then brings up the question of "how do we get the query to the selector"?

            - Apparently, according to this: Hooks · React Redux, we can pass arguments dynamic arguments to selectors as a *second* argument in the selector function?? Is this new???

            - With this, we could create a selector that accepts the query as a second 'dynamic' argument. Which means that we *would* be able to implement 'searching a store index' as just a selector instead of a saga that stores results back into the store. Hooray! ?

This is what the naive implementation of a `searchTransactions` selector would look like:

```
const makeSearchTransactionsSelector = () => createSelector(
    [
        (state, query) =>
transactionsIndexSlice.selectors.searchBigrams(query)(state),
        (state, query) =>
transactionsIndexSlice.selectors.searchWords(query)(state),
        getTransactionsById
    ],
    (bigramIds, wordIds, byId) => {
        const bigramTransactions = (
            transactionIds
                .map((id) => idMap(id, transactionsById))
                .filter(({description}) =>
SearchService.stringStartsWithQuery(description, query))
        );

        const wordTransactions = idsMap(wordIds, byId);

        return SearchService.orderTransactions([...bigramTransactions,
...wordTransactions]);
    }
```

```
);

// Can then be called in a component like:

const [query, setQuery] = useState("");
const searchTransactions = useMemo(makeSearchTransactionsSelector,
[]);
const transactions = useSelector((state) => searchTransactions(state,
query));
```

Note a couple things here:

- Instead of creating new 'selectors' inside the list of selectors here, we should just change `searchBigrams` and `searchWords` to accept a second argument for the query.
- We should tidy up the main body of the selector (I basically just copied and pasted stuff from the existing selectors and saga).

But that's the general idea.

So now we don't even have to store the transactions as a piece of component state; it's just a derived value that gets passed down by the Context. The only component state for the provider is the query.

Wait, that's not true. We still need the selected transaction state. Which we can store as just an ID, but pass down through Context as a full transaction. Actually, should we instead store the selected transaction as an *index* of the list of result transactions? That way we can do an O(1) lookup to find the selected transaction in the result transactions, instead of a full search.

- I mean, we could always store it as an ID and then use another selector to look back up the transaction, but there's really no good reason to do that when we already have the full transactions in the component. And we *really* want to limit our surface area on the store.

OK, so to recap, what we get out of this custom context provider is the ability to have independent search results that still leverage the store for the indexed transactions. That's basically what it boils down to.

TabBar w/ Sections

I was initially thinking of making a generic `TabBarWithSections` molecule to abstract the concept of having a tab bar that switches between several sections, but then I realized that it'd be kind of a shitty abstraction. We'd have to pass in an array of the tabs and sections *anyways* so there wouldn't actually be that much abstraction.

As such, instead I'm just going to build a `TransactionDateOptions` molecule directly. Once we have multiple instances of using the `TabBar` with sections, then we might be able to refactor something out.

TextAreaInput

OK, so I recognised that I was gonna need a new `TextAreaInput` for the Notes input (since it was designed as 'bigger'). However, upon realizing that I'd have to port all of the `Inputs` functionality to it as well (or modify the `Input` to support changing the base component, which seemed troublesome considering all the things TypeScript would complain about), I decided that this wasn't worth the effort.

The Note input will just be a regular input. Considering how frequently it'll be used, I'm sure few people would even complain about (or know to complain about it). "Good enough"

Component Breakdown

Atoms

- CloseButton (IconButton + X icon)
- [modification] Sidebar (move CSSTransition into the component)

Molecules

- AutocompleteInput (LabelledInput + options dropdown + `downshift`)
- TransactionDateOptions (TabBar + LabelledInput + ...)
- TransactionTypeOption (OptionCard)
- TransactionTypePicker (TransactionTypeOption + RadioGroup)

Organisms

- TransactionForm (OverlineHeading + CloseButton + LabelledInput + AutocompleteInput + TransactionDateOptions + TransactionTypePicker + CollapsibleSection + Divider + ShadowButton + LinkButton)

Scenes

- SidebarTransactionForm (Sidebar + TransactionForm)

Tasks

- Create the form.
- Connect the form.