Story UFC-196: Transactions Import

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

Acceptance Criteria

This story's gonna be a little different. Here's all of the acceptance criteria from the old story:

- I can click on a button to start the import process.
- I can see the progress of which step I'm at in the import process.
- I can see that there are 5 steps: picking the account to import to, picking a file, creating/choosing an import profile, categorizing transactions/updating descriptions, and completion.
- During the account picking step, I can choose one of my assets or liabilities to have transactions imported against.
- During the file picking step, I can have a file picker brought up so that I can pick a CSV file to import.
- I am not allowed to continue with the import process if I don't use a CSV file.
- I am warned if the file I picked is not a valid CSV file.
- During the import profile step, I have two choices: I can either choose an existing import profile from my list of import profiles to use, or I can create a new one.
- If I choose an existing import profile, I can see the mappings for the profile so that I can double check that it's the correct profile.
- After choosing an existing import profile, I can then move on to the next step.
- If I choose to create a new import profile, I am presented with a way to map the columns of the CSV to the properties of a transaction in uFincs.
- While creating a new import profile, I can give the new profile a name.
- I can map column indices to the transaction properties of uFincs.
- While mapping the CSV columns, I am also presented with a couple of sample rows from the CSV file to aid me in choosing which columns map to which uFincs transaction properties.
- Once I am done mapping the CSV columns to uFincs transaction properties, I can then confirm the creation of the import profile.

- Once I have confirmed the creation of the import profile, I am moved on to the next step.
- During the transaction categorization step, I can pick the type of the transaction (income/expense/transfer for asset imports, expense/transfer for liability imports), as well as the other account that the transaction affects.
- During the transaction categorization step, I can also update the descriptions, amounts, and dates of each transaction.
- Once I am done categorizing the transactions, I can then move onto the final confirmation step.
- During the confirmation step, I am presented with the full list of transactions that I'll be importing, the name of the account the transactions will be imported into, and some summary statistics (e.g. number of transactions, net change to the account's balance, etc).
- If I am satisfied with the import summary, I can then confirm the import. I am then returned to the Home page.
- If I am not satisfied with the import summary, I can move back to the previous steps to change things as I wish.
- At each step, I have the option to go back to a previous step and modify my choices.
- The state of the transactions import process is cleared if I navigate away to another page.

And here's everything new that we need:

- I want to be able to modify transactions in bulk. That is, I want to be able to select multiple transactions (or all transactions) and then perform an action (e.g. change the description) on all of the selected transactions.

Removed Acceptance Criteria

The following Acceptance Criteria were removed, because I felt like it, I was lazy, or they didn't really add much (more so the former than the latter).

- In the import process step summary, I can see the name of the account that I chose.
- In the import process step summary, I can see the name of the import profile that I chose.

Design Brainstorming

OK, so this is a big ticket. Like, a *big* ticket. This could easily take 2-3 weeks to get done. I know that if I say it'll take more time, then it'll take more time by virtue of Student Syndrome, so it's now gonna take 1 week. Why? Because we're behind schedule, obviously.

So what are the big things we need to design for here? Well, we already have a large portion of the business logic done, what with the `transactionsImport` slice and sagas.

However, there are some new additions that we're making as part of the redesign. The biggest addition is the use of selectable transactions and bulk actions when adjusting the transactions as part of step 4. This will likely require the most non-ui work (as well as the most ui work).

Other than that, we also have a completely revamped interface for picking/creating the Import Profiles. AKA, another table and another list.

Additionally, there are some modifications I've been wanting to make with regards to the `ImportableTransaction` class. Namely, making sure it is properly typed with a string for the amount. On top of that, I've realized that the `cleanTransactions` in the `transactionsImport` state should actually be a map of ID → regular `Transaction`, whereas `transactions` should be a map of ID → `ImportableTransaction`. This makes sense once we have the `ImportableTransaction` properly typed.

ImportableTransaction Type

What we should do is have the `ImportableTransaction` implement the `TransactionData` interface, but omit the `amount` so that we can re-type it as a string.

Then we can in whatever other stuff the `ImportableTransaction` needs (e.g. `includeInImport`, `isDuplicate`, etc).

Editable and Selectable Transactions List/Table

re: editable, see below: Made the executive decision to not have inline editing of transactions.

re: selectable, I've made the decision to move the 'Select all' checkbox into the Bulk Actions instead of integrating it into the Transactions table. Why? Cause I'm lazy and hate this ticket.

Also, I think I've finally come up with a reasonable implementation for selectable/bulk actions.

Basically, we'd have a context hook (hereby known as the "Selectable List" hook, aka `useSelectableList`) that stores one thing: a map of `ID → bool` that indicates which transactions are checked. This state can then be used in the following ways:

- At the `BulkTransactionActions` component level, it can do the following:

    - Receive all of the transactions as a prop, so that the 'Select all' checkbox can hook into the context state so toggle everything/know how many are selected.
    - Pass the IDs of all the selected transactions (along with the selected action) to the Transactions Import slice for modifications (this will require binding a separate version of the component to the slice).
- At the `TransactionsTableRow` component level, it can do the following:
    - Check if the current transaction is selected, and pass that state to the `TransactionTypeIcon` accordingly.
    - Toggle the selected state when clicking on the `TransactionTypeIcon` (or the row itself).

Having now thought about it, if we need to pass a list of the IDs somewhere, then I guess storing a map of `ID → bool` isn't the most efficient. I think it'd be more efficient to just have the mere presence of the ID in the map be the indicator that the ID has been selected (i.e. it can be a map of `ID → any` for all that it matters). Then, if the ID isn't in the map, it isn't selected.

This way, checking if an ID is selected is still O(1), while converting the map to a list is just a simple call of `Object.keys` (as opposed to a call of `Object.keys` with a filter for everything that has a `false` flag).

Build Process

Since this story is so large and I'm in a very procrastinatory mood (what with this being the last ticket of the phase before we can get to the 'fun' stuff), I feel like we should *really* split this story up into several technical tickets.

Like, I'm thinking one ticket each for *just* the layout of each step, then a final ticket for getting the selectable/editable functionality of the transactions list/table working, along with actually integrating everything and making it functional.

Because all of the functionality will come in the last ticket, all of the e2e tests can be done then.

Removing the Animations from TabBarWithSections

As part of the third step, there is a tab bar that is used to switch between choosing an existing ImportProfile or creating a new one.

Since this component was very similar to the one used in the TransactionForm, I decided to make an organism out of it.

However, I hit a snag: the fact that we need to use a fixed height (since the tab sections are absolutely positioned, they are removed from the layout) means that we can't have

dynamically sized sections. However, since we're using tables that can have dynamic numbers of rows, we *can't* specify a fixed height.

As such, I decided to just rip out the animations. Maybe we can revisit adding back the animations in the future, but I'm doubtful.

Here's the version of the component with the animations still in it:

TabBarWithSections.zip

Transactions Table - Inline Editing

I don't want to do it.

Too much work.

Users can have bulk editing and single editing.

Deal with it.

Executive decision made.

Also means we don't have to figure out how to make an inline editable `TransactionTypeIcon` that *also* doubles as a checkbox...

Update: You know what a side effect of this decision is that I didn't initially realize?

We don't need the `ImportableTransaction` to have `amount` as a string. Since we're not doing inline editing where the source of truth is the store, we don't have this constraint anymore. Bulk actions should work fine casting string user input to a number for the store.

Hooray! We can eliminate this horribleness!

Transactions Table/List - Editing

When the user clicks the 'Edit' button, what is it supposed to do in the import process? Since we currently tied the editing action to a URL, how are we going to modify it to support editing transactions that aren't in the main store?

The first thought I had was to have the edit button just redirect to a different (import process specific) URL, then modify the TransactionForm to support editing import process transactions.

That should work.

Target Account

So you know how we have that 'target account' thing? On the `ImportableTransaction`? And as a field for mapping as part of the `ImportProfile`? Well, it's a bitch.

See, now that we're not doing inline editing, that means that we have to support this 'target account' thing as part of the transaction form. It also means that we *do* have to put the ID of the account we're importing to onto the transactions, so that it shows up in the table.

So, I guess we just have to fully commit to the 'From' 'To' method. Like stated above, that means that we need to put the importing account on all of the transactions. Then, whereas before we were handling the 'target account' by putting it as the placeholder for the account input, we'll now probably have to do 2 things:

i. Use the 'target account' as some kind of placeholder in the From/To column (whichever isn't occupied by the importing account).
    - This 'placeholder' should be styled appropriately so that the user understands that it must be changed to be a uFincs account. I'm currently thinking something like lighter text color, maybe an underline and/or maybe an asterisk with a message indicating what's going on.

ii. Use the 'target account' as the placeholder value for the corresponding input field in the `TransactionForm`.
    - *How* we go about doing this is another question. This is really starting to make the case for having a separate `TransactionsImportForm` (or something to that effect), considering we kinda just forced the existing form to just handle both use cases.
    - Currently, the following are what would differentiate having a second version of the form:
        - Which URL the ID is picked up from (handled in `connect`).
        - What the `onSubmit` does (also handled in `connect`).
        - Setting the placeholder of one of the account inputs to use the 'target account'.
        - Being able to limit the options of the non-target account input to just the account being imported to.
    - I think the best way of going about this is probably to just export a copy of the `TransactionForm` as the `PureTransactionForm`, so that we can re-use the complete layout for the `TransactionsImportForm`.
        - Then, we can add props to the pure form for controlling the placeholder text of the account inputs.
        - Finally, we can copy over much of the `connect` logic, while removing and tweaking things that aren't necessary.

Bulk Actions

OK, so I'm at the point where I need to implement the logic for the bulk actions (the layout has already been done for a while).

My theory is that this should be a relatively simple process:

i. Get the IDs of the currently selected transactions from the `useSelectableList` state.

ii. Given the type of change being done (and the user's input), send the IDs, type of change, and new value to the `transactionsImport` slice through a new action, `updateTransactionProperties`.

iii. Update the transactions.

OK, maybe I didn't really need to write this down...

But I think a more interesting discussion is how to *integrate* the `BulkTransactionActions` into this process. What props do we need to make this work?

I suppose something as simple as this:

- `onBulkAction: (ids: Array<Id>, property: string, newValue: string) → void`
  - I think we should probably create a string type/enum like `BulkEditableTransactionProperty` so that we can make it explicit about which properties are bulk editable.
  - Additionally, note that I specifically make `newValue` a string. This means that we can pass through the user's input from the `BulkActionDialog` completely raw. This means that the slice will have to handle things like converting amounts to `Cents` (since, again, we made the decision to remove string amounts from the `ImportableTransaction`).

Something else I realized: in order for the bulk `account` property change to work, all of the transactions have to be of the same type. That means we'll need to pass in the `transactionsById` to the bulk actions so that we can reference the selected transactions against their types.

Every other action is fine to operate on any set of transactions, regardless of type.

Also, the Type input needs a default value that corresponds with... the selected transactions? But what if there are multiple transactions? Take the first one? Nah, just leave the default at Income.

Also, the transactions need to be unselected after a change has been performed.

Multi-Select using "shift"

If, like many other applications, we wanted to offer multiple selection through 'shift' clicking, how would that be implemented?

Well, we already pass in the `index` to each `TransactionsTableRow` (for navigating focus between rows), so that should mean that this is feasible.

I think it should basically work like this:

- When 'shift' clicking a row, check if there are any other selected transactions:
    - If there aren't, just select the row.
    - If there are, ...
        - Hmm, I think this is where we hit the roadblock. Why? Because we don't have the complete list of transactions.
        - What we would do is: given the index of the current row, lookup the index of the selected row that is closest. Then, select all transactions between them.
            - To lookup the index of the closest selected row, we'd have to be able to turn the set of selected IDs into the complete list of sorted transactions.
        - One way to get around not passing in the whole transactions is do like what we did with the focus navigation: cheat and store the data on the DOM nodes. We'd just need to store the IDs as data on the DOM nodes as well.
            - Then we'd be able to get a list of the sorted IDs just by querying the DOM. Hacky, yes, but workable.

Maybe we should just leave this functionality for a new story.

Using AutocompleteInput in the Import Process

So something that I realized pretty early on in the logic implementation process was that the `AutocompleteInputs` were insufficient for our use case in the import process. Why? Because the `onInputValueChange` handler pushed every single change (i.e. user input) to the store. As a result, user input that didn't correspond to a selected suggestion would be left in something like the `accountId` field, which is obviously incorrect. This would be enough for the user to be able to 'move forward' to next step, even though it's obviously incorrect.

As such, I quickly moved all of the `AutocompleteInputs` over to `SelectInputs`, since then users would be forced to pick an option.

However, I still want to offer this improved experience to users (at least, on desktop). As such, I think we'll need to add another callback to the `AutocompleteInput`: `onOptionSelected`.

Now, I know I tried to do this previously and gave up, but here's how I think we can make it work:

- We can leverage `onSelectedItemChange` (a prop of `downshift`), but *in addition* to it, we can also leverage `onInputValueChange`. How? By sending through the selected item whenever the user types out the full name of an option.
- Actually, now I remember why `onSelectedItemChange` didn't work out: because it would fire at the same time as `onInputValueChange`.
- So what we *actually* have to do is just continue ignoring `onSelectedItemChange` and just do the following:
  - Whenever the user types out the full name of an option, send through the full suggestion object (i.e. the one that contains an ID/proper value) through `onOptionSelected`, but just the doubled user input through `onInputValueChange`.
  - Whenever the user *selects* an option, send the full suggestion object through both `onOptionSelected` *and* `onInputValueChange`. This way, we'll preserve the existing functionality but enable new functionality.

Update: OK, so for various reasons, adding the `onOptionSelected` didn't work out and we've fallen back to just checking if the label/value are different before committing an option change.

The reason it didn't really work was because I initially tried implementing `onOptionSelected` uncontrolled; that is, there was no `value` prop passed in to control the selected option. Instead, the input value was controlled internally.

I realized this didn't really jive well with the fact that the store is the source of truth for the selected option; in particular, there wasn't really a way to reset the selected option when, e.g., changing the account type in the first step.

As such, I just gave up and went back.

Update: Having seen the pattern a couple more times, I wonder if we should have a separate handler that kind of piggy backs on `onInputValueChange` to execute the 'option selecting' logic that we're doing at the higher level?

That is, we currently have stuff like this:

```
onInputValueChange={({label, value}) => {
    setAccountName(label);

    if (label === value) {
```

```
        // This is an optimization so that the store function doesn't
get called
        // on every single input value change, but only when it should
be cleared.
        if (accountId !== "") {
            // When the label/value are the same, this means no option
was found
            // and only the input value changed. Clear the store.
            saveAccountId("");
        }
    } else {
        // When the label/value are different, an option was found.
Save it.
        saveAccountId(value);
    }
}}
```

What if we dedicated `onInputValueChange` just for the `label` (so it just does `setAccountName`) and then had a separate `onOptionSelected` for the rest of the logic? Could that work?

Ooh, I had an even better idea. What if we just *wrapped* the `AutocompleteInput` into something like an `AutocompleteSelectInput`? Then *it* could contain the state for the input value, while only exposing a handler for changes in the option.

However, it would still need to accept a `value` prop, except that instead of being the *label* of a `SuggestionOption`, it would be the *value* of a suggestion option. I know, this is all so overly complicated and poorly designed, but whatcha gonna do...

Update: Well, I built out `AutocompleteSelectInput`. Poorly. I built it in such a way that it is kinda 'uncontrolled' (since you don't pass in the selected value to dictate the value of the input), but this ended up being a huge limitation, at least for use with most of the import steps.

For the Choose Account and choose import profile steps, we can't use it because the inputs need to have their value controlled by the store. Those are now just left as regular `select` inputs.

The only place we can *actually* use it is in the "Change Account..." bulk action dialog, since its input value is only sent outwards and never dictated at a higher level.

Clearing State when going Backwards

I noticed that going backwards in steps doesn't clear the state from the later steps.

I don't know if this was intentional during the original implementation of the import process, but it definitely doesn't seem to make much sense. As such, I propose to rectify this.

My original (naive) implementation would have been to just clear any later steps whenever we go back a step, but I think the better implementation is a bit more nuanced than that.

In particular, I think it makes more sense to clear everything on later steps when *changing* anything on a previous step.

- For example, if we've already adjusted our transactions, but then go back and change the import profile, the transactions should be reset and re-parsed based on the new import profile.
- Or if we go back from adjusting transactions to change the account, everything should be cleared.
- etc

This way, users have the ability go back to previous steps to view information, but only have it cleared when they change something.

Component Breakdown

Atoms

- Checkbox
  - Don't forget the 'not-all-selected' state (i.e. where it's a horizontal line instead of a checkmark)
- [modification] TransactionTypeIcon (add selectable state)

Molecules

- BulkTransactionActions
  - DesktopBulkActions (secondary Button + vertical Divider)
  - MobileBulkActions (Input + Dropdown)
  - Don't forget about the sticky header state.
- BulkActionDialog (LabelledInput + Button + LinkButton)
- ProgressStep (TextField)
- StepDescription (TextField + heading)
- StepNav igationButtons (ShadowButton + LinkButton)
- StepNavigationFooter (StepNavigationButtons + Divider)

Organisms

- ProgressStepper

    - DesktopProgressStepper (ProgressStep x5 + Divider x4)
    - MobileProgressStepper (StepNavigationButtons + stuff)
- CsvMappingTable (LabelledInput + a new table)
- CsvMappingList (LabelledInput + stuff)
- [modification] TransactionsList (add selectable transactions, excluded state, etc)
- [modification] TransactionsTable (add selectable transactions, excluded state, etc)

Scenes

- TransactionsImport (OverlineHeading + BackButton + ProgressStepper + StepNavigationFooter)
    - ChooseAccountStep
    - ChooseFileStep
    - MapCsvStep
    - AdjustTransactionsStep
    - SummaryStep

Tasks

- Build the `ChooseAccountStep` layout.
- Build the `ChooseFileStep` layout.
- Build the `MapCsvStep` layout.
- Build the `AdjustTransactionsStep` layout.
- Build the `SummaryStep` layout.
- Make the transactions import process functional.