Story UFC-393: Currency Symbol Preference

The following document will go over the design of the custom currency symbol preference, as outlined in Story ticket UFC-393.

Acceptance Criteria

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

- I want to go into my settings and change the displayed currency symbol from $ to ... anything else.
- I want this change in preference to be persistent between refreshes and logins.
- I want every displayed instance of "$" to change to whatever currency symbol I set the preference to.

Design Brainstorming

OK, so we're gonna need to do several things:

i. Add a new section to the settings to support general user preferences.
   - Will need to design this in Figma.

ii. Add a new database table for user preferences.

iii. Figure out how to store the preferences client-side.
   - Presumably in Redux.

iv. Somehow modify `formatMoney` to pull in the currency preference without having to pass it any extra parameters.
   - That is, how do we add state from redux to a stateless static class method?

Let's take it one by one.

Preferences UI Design

Simple. Just like the Billing setting section except instead of a button, it's select input (and obviously the text is different).

Although... thinking about it more, if we *really* want to have nicely designed preferences, we should have a little indicator next to the select input for when the change is saved. Just a nice little checkmark. Or an 'X' if it fails.

- As far as error messages go, it *could* be put onto the input as just part of its error label (like usual), or we could do error toasts. I think, in this case, error toasts is easier, but error labels is probably a better UX.

So, we need a new 'SuccessIndicator' atom. We definitely then need to somehow integrate it with Input, but I don't know how. Add in the base Input component? Create a separate molecule that combines the two? But that would be stupid cause then we'd have to create versions for each input type.

I think the best solution is to add it to the base Input and then only include it in the HTML output with a flag prop.

Preferences Database Table

I think we should go simple here. Just a single `preferences` table, one row per user, each column is a difference setting. So in this case, the starting schema would look like:

- id
- userId
- currency
  - Note: I think we'll need to set `defaultValue: null` on this because of how the data encryption works. We can't exactly assign a defaultly encrypted value of "$" since... that would require us to generate the encrypted value at like user sign up. That's *wayyyy* too much work, so just default to null for "$" and then update it accordingly if/when a user changes the setting.
  - Although... we're gonna need to create the preferences row at user creation *anyways*, so...
  - And we're gonna need a migration to create the preference rows for all existing users...

And then, as we add new preferences, we just add new columns.

Other options would be having dedicated tables for setting types, one row per setting/per user, JSON blobs, etc.

This basic design aligns with the "Single Row" design described in Storing User Settings in a Relational Database.

Other designs: https://stackoverflow.com/questions/10204902/database-design-for-user-settings

Preferences Redux State

Theoretically, this shouldn't be any more complicated than something like the `user` slice. So... it won't be anything more complicated than that.

Oh, wait a minute, we literally have a `preferences` slice already. It's used for the `showFutureTransactions` state, which is only kept client-side right now. Hmm... I guess there isn't really a problem with just chucking in the other server-side preferences with them.

However, that does mean that we need to add in the sagas for fetching/setting the server-side preferences.

formatMoney Modifications

Oh god. Whatever we do to support this, it's gonna be jank.

OK, so how do we pull state into a currently stateless function?

- We could... change `ValueFormatting` to be a stateful class. It could store a copy of the `currency` preference as a class (i.e. static) variable that gets updated by... manually subscribing to the Redux store and listening for changes in store value? I guess that could work; everything would still be static that way, so we wouldn't have to update anything.
- We could... have a saga that listens for changes to the `currency` preference and *that's* what updates the static value on `ValueFormatting`? Yeah, that could work too. Probably less jank than making `ValueFormatting` care about the existence of Redux by having it subscribe itself, not to mention being much more in line with our, you know, use of sagas.

You know, I like that second solution so much that all my other ideas (somehow tying the state to local storage...) are just stupid.

OK, hard part over. Now to do the UI design.

No no wait, there's still something I haven't considered: putting the currency symbol on the *right* side for certain currencies (or even elsewhere??? idk). Do I really care that much to support it? No. I just wanted to make sure that was recorded somewhere.

Where to get the Currencies From

Yeah, there's the minor problem to figure out where we're gonna get the actual currencies/symbols from.

I guess this is good enough? currency-symbol-map/map.js at master · bengourley/currency-symbol-map

- We can just lift out the whole currency map from this; no need to actually bring in the package. Just gotta make sure to keep the copyright notice.

The one 'downside' to that package is that it only has the 3 letter codes for currencies (rather than full names), but I think that's good enough.

In-Dev Problem: Changing the Symbol Doesn't Cause Renders

Date: December 26, 2021

So, I've just realized why storing app wide state in the static property of a regular class is a *terrible* idea as far as React is concerned: it isn't considered a change of state that's worthy of a re-render.

Which is to say, if we change the currency symbol set in `ValueFormatting`, it won't cause any currently rendered components to update with the new symbol. This is most obviously noticed during a regular page refresh: the components will render in with $ and won't change once `ValueFormatting` has been updated from the sagas.

Wow, it's almost like we use Redux for a reason!

So, what are our options?

- Well, the most obvious one is just moving the currency symbol state into Redux.
    - This means that everytime we want to use the currency symbol outside of `ValueFormatting`, we need to use a selector into Redux (or more likely, a hook that wraps it up for an easier interface). OK, fine, whatever.
    - However, this makes thing more complicated for `ValueFormatting` itself. If the source of truth for the currency symbol is moved into Redux, then that means that either:
        a. `ValueFormatting` now has to subscribe itself to the store to get updates, or
        b. The currency symbol has to be passed into the `formatMoney` call by the calling component.
    - 1 doesn't actually solve anything, since that wouldn't fix the re-render problem. 2 *does*, but it means that every function that calls `formatMoney` has to do a Redux lookup for the currency symbol.
        - Looking into how many times `formatMoney` is actually used, I guess this isn't the *worst* thing in the world. It does, however, mean that a good number of components that previously didn't rely on Redux now will. Hurts re-usability I suppose, but oh well...
    - 2 makes the most sense (from an idiomatic POV) if we choose Redux as the source of truth.
- The second most obvious option is a *huge* hacky workaround: just re-render the whole app whenever we change the currency symbol.

    - OK, but how? Well, we *could* wrap the whole app in a Context that's specifically for the currency and then just update the currency in there whenever we want to re-render everything.
    - Note that while we would be storing the currency in this Context state, it wouldn't actually be *accessed* from there; it would still be stored and accessed from the static property on `ValueFormatting` for ease-of-use.
    - So it'd really just be a giant hacky way of re-rendering everything.
    - Theoretically, there wouldn't really be any performance penalty to doing this — after all, if all we store in this Context is the currency, and we only change it when we change the currency, then re-renders *should* only happen under those conditions (which should be few and far-between).
    - However, it just *feels* like a 'wrong' solution.
        - Although, at this point, do I really care? Not really.
    - Although I guess one hiccup with this is how we go about updating Context state from Redux...
        - I guess we can have a dummy component that just listens for changes in the currency preference and updates the Context accordingly...
    - Oh never mind, this is completely idiotic solution that's actually non-functional because I forgot that Context only re-renders Consumers, not *everything*.
- OK, but we can still take the above concept and apply it like so: have the *root* app listen for currency preference changes and re-render accordingly.
    - But it's still a stupid solution.
    - Let's just go with the Redux solution...

OK, Redux source-of-truth it is!

Adding to Backup Format

Ya know, I was thinking that we should add the preferences to the backup format. Cause, it's obviously user data that is persisted.

But then I was thinking, "does a user *really* need to back that up? nah, I don't need to implement this."

But then I realized that it's not just for backing up core data, it's also about restoring to instances of uFincs. If a user takes a backup of their data, it should restore *everything* just as it was in another instance/account of uFincs.

So, I guess I can't even argue myself out of this...

Component Breakdown

Atoms

- SuccessIndicator
- [modification] Input (add SuccessIndicator)

Molecules

N/A

Organisms

- CurrencyPreferenceForm

Scenes

- MyPreferences section components
- [modification] Settings (add the new section for "My Preferences")

Tasks

- Build the Backend.
- Build the Frontend.