Story UFC-202: Recurring Transactions

The following document will go over the design details related to implementing recurring transactions, as outlined in Story ticket UFC-XXX.

Acceptance Criteria

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

- I want to a way for transactions to be automatically created on a given schedule.
- I want to manage (i.e. CRUD) these recurring transaction schedules.
- When changing the date range into the future, I want all my charts to display these future scheduled transactions (differently).
- When changing the date range into the future, I want all my tables to display these future scheduled transactions (differently).

Design Brainstorming

- Based on this this comment on Hacker News, I was intrigued to learn if there was a better way of implementing recurring transactions.
- Low and behold, turns out the `rrule` and `rruleset` standards might actually be useful, since jakubroztocil/rrule exists.
    - tl;dr it's a library that lets us setup the rules for a recurrence pattern (e.g. 'weekly, for 5 months, starting tomrrow, etc') and then query that rule to get dates of occurrence (e.g. between date X and Y, on which dates Z does the transaction happen?).
    - Seems very powerful and a library that, although somewhat large (rrule@2.6.4 | BundlePhobia), would save me an inordinate amount of time implementing these kinds of things by hand.
    - Now the bulk of the work will be figuring out all the cases that the above Hacker News comment says is hard (e.g. updating/deleting all recurring events from a master record, or just a single instance, or ...) in the context of our architecture.
- Refer to Transactions Form for the original brainstorming of this idea. I think the most important part is the following, describing how to present the recurrence schedule to the user:

    - `This transaction happens every [X] [days|weeks|months|years] on [N/A|weekday|day|month, day].It should start on [start date] and finish [Never|On date X|After X occurrences].`

- Instead of putting editing for recurring transactions as a top tab in the Transaction Form (as Transactions Form and the Figma designs do), I instead want to have a separate tab on the Transactions *page*, called "Recurring Transactions", that would come after the regular Transactions tab but before the Summary tab.
    - This view would show the user *all* of their recurring transactions. As in, the 'templates', but the realized transactions themselves (those would be shown on the Transactions tab).
    - Here, users can choose to edit/delete the 'templates', altering all transactions in the future (but not existing/past transactions).
- When computing a date range, I want the following to happen:
    - All recurring transactions are checked if any transactions would happen during the date range.
    - If yes, 'realize' the transaction and include it as part of the date range.
- There are several consequences to doing this:
    1. The transactions table must now show *future* transactions.
        - These should be highlighted to be in the future somehow (maybe an orange outline or something?)
    2. Graphs must now show future trends.
        - For line charts (e.g. net worth over time), this could mean something like showing a vertical line at 'today' (or maybe even just a dot at 'today', just something to denote the separate physically) and then coloring the line past 'today' a different color (again, maybe orange?).
        - For bar charts (e.g. income/expenses over time), it's a bit more tricky. **I'll need to think about this**. The obvious solution is just... change the color, but it's not obvious *how* to change the color (maybe lighten or darken the green/red?).
- Deleting a transaction that has *actually* been realized by the recurrence series (i.e. the date has passed where the transaction should have happened) should not affect past nor future transactions in the series.
    - Ditto with editing.

- We should probably create a dedicated `service` to handle dealing with the recurrence series. Or I guess that would just fall under the purview of the new `TransactionSeries` model.
- Speaking of which, what *is* the data model for these 'recurring transactions'? We kinda never finished that part when making the initial domain model...
- Well, here's the data we need:
  - The template for the transaction
    - So all data currently in a transaction, minus the date
  - The schedule for the recurrence series
    - Using `rrule`'s terminology, this is the following:
      - `interval`
        - This is how often the series repeats (i.e. a number)
      - `freq`
        - This is the 'size' of the interval (i.e. day, week, month, year)
        - ("happens every `[interval] [freq]`")
      - `dtstart`
        - This is the starting date
        - I don't really like using `dt` at the start, so we'll probably either call it `start` or `startDate`
      - `until` [optional]
        - The *date* the series ends
        - This is one of the two ways the series can end
        - To match `startDate`, I think this would better be called `endDate`
      - `count` [optional]
        - How many transactions are generated by the series
        - The second of the two ways the series can end

        - If neither `endDate` nor `count` are specified, then a series is said to not end (i.e. the user chose 'Never').

- OK, so we have a transaction 'template' and a transaction 'schedule' model/table, effectively. Technically, there's nothing *stopping* us from putting it all together on one model/table (I don't think...), but does it make more sense to have them separate?
    - No, I think it's easier to keep them together.
- So let's just have a single `RecurringTransaction` model/table with the following data:
    - `id`
    - `userId`
    - `creditAccountId`
    - `debitAccountId`
    - `amount`
    - `description`
    - `notes`
    - `type`
    - `interval`
    - `freq`
    - `startDate`
    - `endDate`
    - `count`
- And since we've stated that a realized transaction cannot affect the series, there's no need to have a relation between the realized transaction and the `RecurringTransaction`.
    - Theoretically this could come to bite us in the butt if we ever change our mind on this (aka the users want it), so maybe it'd be good to have just in case...
    - Probably not the worst idea to just include it. Make it null by default and we should be fine.
- Also, we probably want to have the `userId` directly on the `RecurringTransaction` so that we can easily look them up for the user, instead of stupidly trying to look them up by... account IDs??

- OK, now that we've determined that we're gonna have a `RecurringTransaction` model, what business logic makes sense to put on it as opposed to a separate service?
  - Well, unless we're going to be having recurring... anything else, I don't think there makes sense to abstract the `rrule` package with a service. As such, the logic behind, e.g., the determination of which dates exist within a given recurrence series (i.e. the reason `rrule` exists) can be placed directly in the model.
  - Now, how we go about integrating the virtual realization of recurring transactions with the date ranges, *that* might require it's own service. It'd basically be a service that integrates the `RecurringTransaction` model with the `DateIndex` structure.
    - Although, thinking about it more, the logic for 'virtual' realization of recurring transactions is along the lines of:
      i. Loop over all recurring transactions
      ii. If start date (and end date, if applicable) is within date range, then:
        1. Use `rrule` to determine all the dates within the range the transaction occurs
        2. Create transactions for each of the dates
      iii. Return all created transactions
      iv. Combine all 'realized' transactions with actual transactions from the date range
    - As such, there isn't really any interaction between the `DateIndex` as a structure and the `RecurringTransaction` as a model.
  - So yeah, I don't think we'll really need a completely separate service to handle it.
  - I mean, I suppose we could have a separate *selector* that handles the virtual realization, so that the `selectTransactionsBetweenDates` selector just combines its existing results with the virtual transactions to get a total result. At that point, almost no changes need to be made to any other part of the infrastructure, except at the display level to denote transactions that are in the future (which, strictly speaking, isn't strictly related to recurring transactions -- users can, of course, make single future transactions; they just aren't displayed any differently right now).

- Now, as for the *actual* realization of recurring transactions, that would seem to best be handled by a process that runs at app boot to check for any recurring transactions that should be realized today. Then just actually create the individual transactions.
- To summarize so far, here are the new 'things' that need to be created/changed:
  - Create the `RecurringTransaction` model
  - Modify the charts to display future transactions differently
  - Modify the `TransactionsTable` to display future transactions differently
  - Build the UI for the `Recurring` section of the `TransactionForm`
  - Add the new 'Recurring Transactions' tab to the `Transactions` scene
  - Build the `RecurringTransactionsTable` (that just builds off the existing `TransactionsTable`)
  - Create a new slice for storing the recurring transactions
  - Build the saga that handles actual realization of recurring transactions at app boot
    - i.e. a `recurringTransactions.sagas`
    - Calls a `RecurringTransaction` model function for the core logic
  - Create a new selector for virtually realizing recurring transactions within a date period
  - Modify the `selectTransactionsBetweenDates` selector to include the results from ^
  - Create the sagas to handle CRUD of recurring transactions (including fetching at app boot)
  - Modify the `TransactionForm` to support editing recurring transactions (the template)
    - This basically just means removing the `One-Off` option so only the `Recurring` portion of the date section is present
    - This also means we'll probably need a new URL (`/recurring-transaction/:id`)
  - Remove cross slice selectors that are no longer being used
    - e.g. `selectCurrentMonthTransactions`, `selectTransactionsGroupedByDay`,

`selectTransactionsSortedByDateDesc`, and their corresponding chart selectors

- All the backend stuff needed to accommodate these changes (i.e. the new table, new migrations, new seeders, etc)
- Put a tooltip over the 'Delete' button of the `RecurringTransactionsTable` to indicate that past transactions will not be deleted
    - Yeah, this is kinda a cop out...
- Add the recurring transaction model to the encryption schema
- We'll probably need some sort of caching system in place for the virtual realization of transactions, so that they don't slow down the date range calculations too much

Post-Started-Implementing-Already Design

So, for the `RecurringTransactionsTable` design, we kinda have a small problem. We *technically* have more information that could be displayed (i.e. the schedule information). But I kinda just want to take the lazy route and just re-purpose the `TransactionsTable` in its entirety, except for maybe changing the `Date` column label to `Start Date`.

And if we want to take the laziest route possible, then, in terms of data, we'd have to take the recurring transactions and map them into `TransactionData`, except we use the `startDate` as the `date` and just kinda discard all of the scheduling info.

That'd be the easiest way to go about it. Just add like a prop to re-map the column names or something (or just a boolean prop to toggle the date column label), and then custom handlers for editing/deleting, a custom selector, and we should be good to go.

Also, can't forget about making a `RecurringTransactionsList`...

...

Also, I guess we need to add the recurring transactions to the `transactionsIndex`? Shit...

OK, well I just tried adding them, and it's not so easy. Firstly, we can't `set` on the `transactionsIndex` with recurring transactions because it ends up being two `set` operations (one for transactions and one for recurring). As such, the regular transactions just get removed.

Secondly, I didn't actually get this far, but I have a hunch that the search/autocomplete lookups wouldn't work without modifications to include the recurring transactions in the source data set.

As such, I'm just deciding to not add the recurring transactions to the index. This should be fine since once the transactions have been realized once, they get added to the index anyways.

...

I have made a grand mistake. A mistake of several large, epic proportions.

You know how I thought that we could just add the 'virtually' realized transactions to the `selectTransactionsBetweenDates` and be gucci? Yeah, that's not really gonna work... Why? Because although we have the full objects at the selector level, once they get passed into the transactions table/list, the data has to be selected back from the store. And since the transactions are virtually realized, they don't actually exist in the store anywhere...

Shit.

So what our options?

Well, if we're gonna maintain the status quo of 'the list items select directly from the store', then I guess we're gonna need to find a way to make these 'virtual' transactions slightly less virtual (i.e. store them somewhere).

I guess this is kinda where our requirement for a 'cache system' could come into play...

Basically, every time we 'realize' a recurring transaction into a set of 'virtual' transactions, we store them in some data structure with a cache key that relates them back to the recurring transaction in such a way that we don't have to re-realize them on future computations, but can easily look them up by ID for the purposes of the list/table.

Now to design such a data structure. This is basically gonna look like `DateIndex` 2: Electric Boogaloo.

This probably also means that the `RecurringTransaction.realize` function will be moved out of the model and into the data structure, since the function will rely heavily on the cache system.

Realized Transaction Index

I think there's gonna need to be at least two parts: one index for the transaction IDs, and another index that allows for quick lookups by recurring transaction.

This second index will probably be a bit more complicated than just lookup by the recurring transaction ID, since we need to account for its actual properties (i.e. schedule) when looking up the transactions.

As such, the actual key might look like `${id}-${interval}-${freq}-etc`.

However, thinking about it more, this is just a way to get around cache invalidation.

So what other ways do we have to deal with cache invalidation? Well, we could just listen for recurring transaction updates/deletes and flush the realized transactions from the cache for that particular recurring transaction. This way, the key could just be the ID of the recurring transaction.

So my current idea for the data structure looks like this:

```
{
    byId: {
        [id]: {...transaction}
    },
    byRecurringTransactionId: {
        [recurringTransaction.id]: {
            [date]: id
        }
    }
}
```

Note: By having the internal `date` index only map onto a single transaction ID, we're assuming that a transaction can't recur multiple times on one day. Which I think is a fair assumption.

Now, the reason to use a `date` index inside the recurring transaction index is so that we can run the `rrule.between` function, get the dates, and then immediately check in the index if the transactions have already been realized. If they haven't, then they can be created and stored in the index.

I think this will work. Let us call this structure `VirtualTransactions`.

A Big Snag - Can't cache from a selector

We have hit a *big* snag. One where one particular design decision is finally coming back to haunt us.

You know how all the date range stuff is implemented as a context provider? HUGE MISTAKE.

Why? Because we finally have a case where we need to respond to (i.e. listen for, i.e. redux-saga `take`) on date range change operations. And we can't. Because it's all stored in context.

What is this case we need to worry about? Realizing virtual transactions.

Actually, now that I think about it more, that isn't the core problem.

The core problem is that the cache for virtual transactions differs from something like the date index cache in one key way: while the date index is a *pre-computation* cache, the virtual transactions are a *post-computation* cache.

That is, we insert into the date index before we query it. However, we don't 'query' for virtual transactions -- we compute their existence and *then* cache them. This is why the current `add` operation is so weird: it simultaneously creates/returns the list of realized transactions while also updating the `virtualTransactions` cache.

And *because* the virtual transactions is a post-computation cache, we do not follow the standard redux flow of "action → selector". We can't have an action that computes the transactions and then returns the transactions (because actions don't return things) and

we can't have a selector that computes the transactions and caches them (because selectors can't modify state).

So... what do we do?

It's almost like we need two *separate* caches... cause the only reason we even need the cache in redux is so that we can query it for the transactions table/list. So we could have one cache that is just kinda 'in-memory' that is used at computation time, then a second cache that is storing the virtual transactions in the store for querying by the tables/list.

But that just *feels* wrong.

Another thought I had was to make a single hook that handles kinda does something like "listen for date range changes, select the whole `virtualTransactions` state, run `VirtualTransactions.add` to get an updated state + list of transactions, issue an action dispatch to update the store, then combine the returned list of transactions with the results from `selectTransactionsBetweenDates`". Basically, it'd be a hook that abstracts over the `selectTransactionsBetweenDates` logic.

- But the fact that that logic is now hook level means that a whole bunch of other things now have to be changed (e.g. other selectors that depend on `selectTransactionsBetweenDates`). Although... there doesn't seem to be many of those.

Hmmmmmm...

The hook approach is probably the *easiest* stop gap, until we decide to refactor everything to make the date range state a redux slice. I mean, there's basically *zero* reason for it *not* to be at this point; the original reason was so that we could have independent date range 'zones', but that's no longer a thing.

Another Snag - Toggling between Future/Current Only

I've just come to realize that the lists/tables + charts aren't the only things that rely on date range calculations for transactions.

There's also things like... account balances, transactions summary, account/transaction by type aggregates, etc.

As such, I almost think we need to have some sort of way to show the user that 'future' transactions are being shown. Otherwise, even something like the default "Monthly" view could be confusing. Like, if it shows future transactions all the time, then how would a user be able to see their current position if everything is always calculating the future? For example, the transaction type summary/type aggregates.

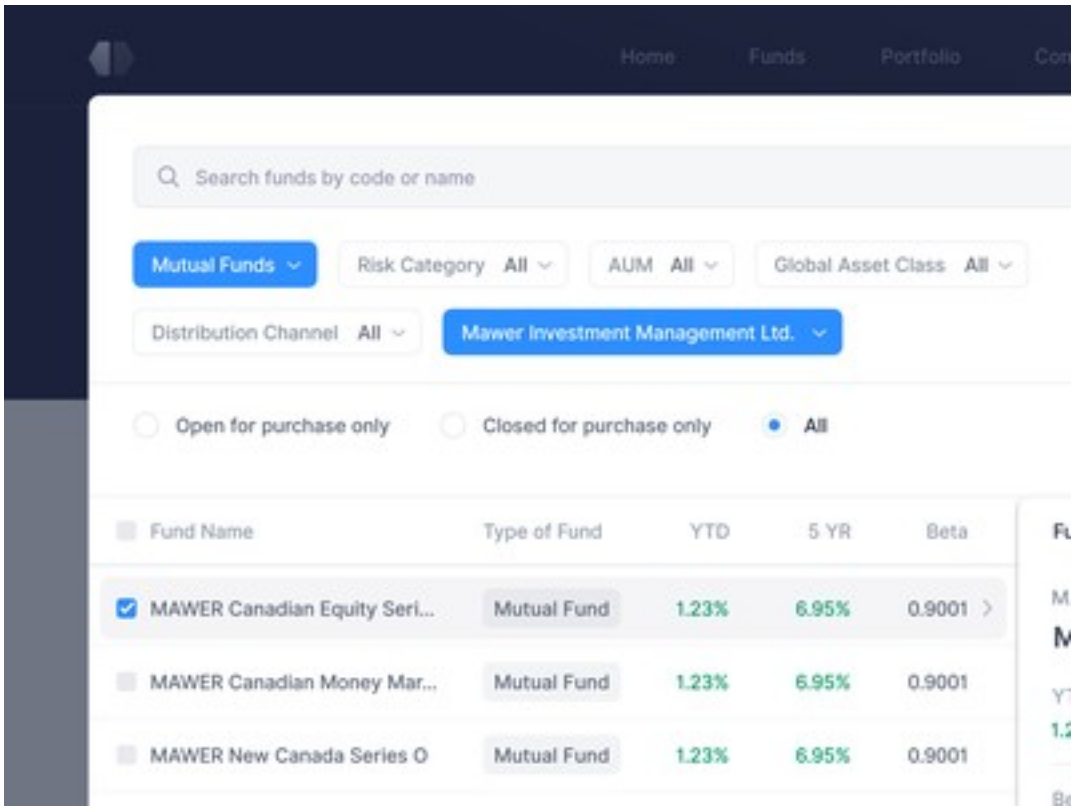
So, there's two ways I think we can go about this: either have a second 'future' amount next to every value, or have some sort of "Include future transactions" toggle so that the user can now whether or not values are including future transactions.

Personally, I'm leaning towards the latter.

Now, where can we *put* this toggle? Well, I think it makes most sense to keep it close to the date range stuff. Maybe to the right of the switcher? That way, on mobile, we can include it under an overflow menu?

- Or maybe it can be displayed beneath the switcher? I feel like that would look better aligned horizontally speaking, but maybe less so vertically. It's a tradeoff.
    - We'd almost certainly have to use such a layout as a responsive layout, for smaller than full-size desktop displays, but larger than phone displays.
- Oh, perhaps we could include it in the range size picker, with another horizontal bar to separate it? Then it could just be a toggle button like they already are.
    - If that's the case, then maybe it would make sense to remove the existing separating bar, and only have the bar between the range size options and the "Include future" option. That way, we more semantically group the range size options apart from this new option.
- btw, I generally hate the idea of putting toggle settings in a dropdown/overflow menu, but I mean... what else are we gonna use?
    - Maybe we could include the button in line with the search input? Before/after it? And shorten the label to just "Future"?
    - No wait, that doesn't work, because the search input is only the transactions page...
    - Maybe we just give up on incorporating it inline and move it into the "Settings" page. I mean, it *is* the settings page...
    - Or maybe we put an overflow menu above the date range switcher? On the same line as the header? Then it'd only be awkward on the Account Details page, since there are three edit and delete buttons up there.
    - Or maybe we just stick the button up in the header, without an overflow menu? It seems like it'll fit with the account detail buttons, so it should be fine...
        - Worst comes to worst, we can cut the label from "Show future" to just "Future" on *really* small screens...
- I definitely don't want to hide this setting on something like a completely separate "Preferences" page in the User Settings, cause 1. it would be stupid to have a page for a single setting and 2. it would be far too disconnected/hidden from the controlled view, aka bad UX.
- In terms of the design of the 'checkbox' itself, we have a couple options:
    1. Regular old checkbox. Aka, the current `Checkbox` component.

2. A more modern 'switch-style' checkbox. You know, the one with a giant circle in the middle that toggles back and forth.
3. A kind of 'toggle button. See as an example:
    a. I kinda like this solution a lot. It'd be much more in line with our current aesthetic (boxy buttons).



OK, now how do we want to store this state? Because it's probably to the user's benefit that this act as a 'preference' or 'setting' that persists between app use.

- Well, since it's a preference, probably makes the most sense to store it on the user's table. Or maybe have a separate preferences table? Maybe a separate preferences table...
    - It's easier to just store it on the users table, but maybe it's more scalable to store them in a separate table? Idk, I don't anticipate having a *ton* of preferences anytime soon.
    - Or maybe, we just go simple and not even store it in the database. Just have it default to "on", store it in Redux, and call it a day.

        - The state will still be remembered through page refreshes, just not between logins/devices. Which I think is acceptable enough for now (until someone complains).

- Then, from the frontend, it probably makes more sense to store this value under a 'preferences' slice rather than using a context provider like the current date range stuff does.

Now, how does this toggle actually interact with all the date range calculations? Well, since we still haven't figured out how we're gonna combine the virtual + regular transactions together, it's hard to say.

- But if we take the approach of having a single hook (say, `useDateRangeTransactions`) that combines regular + virtual, then that would be place that also handles looking up the setting and determining which set of transactions to return.

- That's why it's so important that we have a single place that handles combining the transactions. Otherwise, we'll be checking this setting everywhere.

- Which is also why it's *such* a pain that we can't make a single selector out of all this. Because of the post-computation cache problem, we either have to move the date range state into Redux so that we can listen for those actions, or we just use a hook as the single source of truth.

Actually, going back to the post-computation cache problem... I had an idea.

- Basically, what if we had a component that was 'connected' to the date range context and issued actions to the store for whenever the date range values changed, essentially acting as a bridge for the context to the store.

- Then, the `virtualTransactions` sagas could listen for these date range actions and compute the realized transactions.

- Finally, there'd be a selector that runs the *current* `VirtualTransactions.add` operation (that would normally modify the store state) and just returns the transactions. That way, we don't have to change the state shape.

- Now technically, this is *not* most optimal, in terms of performance. Why? Because we're technically doing a double computation: first, by the selector, and second, by the sagas. We're 'realizing' the transactions in two places. It's just that every n+1 date range change can make use of the already realized transactions from the n'th change. So... likely some performance improvement, just not optimal.

- But at least it ensures ...

    – Nope, this doesn't work.

    – Why? Because we don't maintain ID integrity.

    - If the sagas realize one set of transactions (and store them), but the selector realizes a second set, then they'll have different IDs. That means, for a short period of time, the selector will have output a set of transactions with IDs that don't map to anything in the store. Yes, it'll re-run once the store state has been updated with the saga realized transactions, but that doesn't negate the fact that we're putting the app into an invalid state.

- So... what can we do?
    - I think the best thing we can do is change the `VirtualTransactions` structure to include a separate `findBetween` operation that only realizes the *dates* for each recurring transaction, and then looks up the dates in the state to find the transactions. This way, the selector can use this function to only do lookups. The result is that, instead of creating 'invalid' transactions (with unknown IDs), we instead return *nothing* for one call, but then get them in a second call once the sagas have updated the state.
- Man, this is all *way* more complicated than it probably has to be.
- But at least, by using this context-bridge component, we can keep the source of 'transactions between dates' as a selector instead of a hook.
    - Worth trying.
    - Will also make adding the 'include future transactions' toggle easier, if it's all redux-level stuff.
- Don't know what the performance implications of this 'context-bridge' component will be, but I'm sure it'll be fine.

Identifying Future Transactions

So there's kinda a dilemma with how we want to treat 'future' transactions, specifically wrt the transactions table/list.

Originally, I had wanted to do 2 things to future transactions:

i. Make them visually different (something like an orange outline or background)

ii. Disable the edit/delete actions

The mechanism for checking for a 'future' transaction was just to check the date (if date > today, apply above).

However, I realized that this scheme doesn't fully cover our needs. Take user created future transactions: I guess we want to color them differently, but we definitely don't want to disable the action buttons (cause they are concrete transactions, not virtual).

As such, and taking into account the idea floated above of having a 'future transactions' toggle, I propose the following:

- All future-dated transactions (concrete or virtual) should be visually different
- Only virtual transactions should have their actions disabled
- The 'future transactions' toggle should include both concrete and virtual future-dated transactions

Now, I must acknowledge that there could be a point of confusion with this scheme: namely that users could have a mix of concrete and virtual future-dated transactions, thus having future-dated transactions that sometimes have disabled actions and sometimes don't. As such, they might be confused about *why* some of their future transactions have actions and some don't, and in fact might also get confused about why they even have these future transactions at all.

Now, I would suspect that the number of users who would manually create concrete future-dated transactions would be quite small, especially with the introduction of recurring transactions. So it is unlikely that this would confuse a large number of users.

Nevertheless, it's probably a good idea to put something like a tooltip over the empty actions of the virtual transactions to explain why these transactions don't have any actions.

So, determining a future-dated transaction is easy enough, but how do we determine a concrete vs virtual transaction?

Easy, a virtual transaction is a future-dated transaction that *also* has a `recurringTransactionId`.

- Concrete realized transactions would also have a `recurringTransactionId`, but, by definition, their date can only be today or in the past.
- So this definition should be fine.

Future Transactions - Chart Design

Originally, I was just gonna color the line differently at the junction of future transactions on the charts, but that doesn't accommodate for color blindness.

As such, not only should we change the color (btw, I'm kinda liking pink as the 'future' color), but we should also change the line style to be dashed instead of solid.

Don't know if this is even possible, but I'm sure we'll figure it out.

HUGE UI DESIGN CHANGE

I have now decided to put a border on the `OptionCard` in its inactive state (i.e. what was normally just a white background, now has the same border around it as all the other inputs).

Why? Because, while implementing the `ShowFutureToggle` button, it just seemed weird for it to have no border and look inconsistent with the `DateSwitcher` that's right next to it. As such, I thought that maybe the `OptionCard` should have a border as well, to consistently 'group it' as an 'input'.

Personally, it looks fine, I'm just concerned of whether or not this will confuse users. In theory, it should confuse users *less*, but our use of these 'toggle buttons' isn't really consistent with anything else.

Virtual Transactions: Redux-Persist or Not?

After stumbling into the `store.ts` file and realizing that we haven't been adding the latest slices to `redux-persist`, I hastily threw everything new in, including the virtual transactions slice.

But then, I started to think... *should* the virtual transactions be persisted?

Pros:

- Minorly speeds up date range changes since things only need to be cached once

Cons:

- Can vastly increase the size of the persisted state, with no (obvious) way for the user to clear the cache
    - Users could browse 1000 years into the future once and then be stuck with all those virtual transactions forever
    - Increasing the size this way has detrimental effects in terms of serializing/unserializing
- Can slow down the initial realization since all those virtual transactions have to then be persisted

So... I'm leaning towards *no*.

Component Breakdown

Atoms

- N/A

Molecules

- RecurringDateForm
- [modification] TransactionDateOptions (add ^)

Organisms

- RecurringTransactionsTable

Scenes

- [modification] Transactions (add the RecurringTransactionsTable as a tab)

Tasks

- Build the backend.
- Build the frontend.