Story UFC-106/128: Transaction Autocomplete

The following document will go over the design of the Transaction Autocomplete feature, as outlined in Story ticket UFC-106/128.

Acceptance Criteria

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

i. Once I have typed 2 characters into the description box, I start to be shown a list of descriptions that I can choose from.

ii. The list of descriptions must contain results that start with the text I've typed so far.

iii. The list of descriptions should also contain results that are related to what I've typed so far. For example, if I type "phone", then "Paid phone bill this month" could be one of the results.

iv. I can use the keyboard arrows to pick a result from the list.

v. Once I've selected a result, the rest of the form is filled out with the details of the transaction (except the Date field).

vi. I do not want the Date field to be filled out with the information from the selected transaction.

vii. If I have happened to have created multiple transactions with the same description, I only want the latest one's information to be used for auto-filling the form when I select it as a result (i.e. if two transactions share the same description, but have different amounts, I want the latest's amount to be used).

viii. Once I've selected a result, I can then press Enter while inside the input to create the transaction (or just click the 'Create' button, but duh).

Design

At a high level, how this feature should work could be described as follows:

i. The `NewTransactionForm` now has a `MaterialAutocompleteInput` component in place of the former `MaterialInput` used for the `Description` field.

ii. This `MaterialAutocompleteInput` takes as props (among other things) the following:

- The current value of the text in the input.

- An array containing the options that the user can select from.

  Essentially, the `MaterialAutocompleteInput` is just a slightly fancier `MaterialDropdown`.

iii. `NewTransactionForm` has a connected version of the `MaterialAutocompleteInput` known as the `TransactionAutocompleteInput`.

- This version connects to the store to grab the values for the options that the user can select from.

iv. Additionally, it also connects to the store to be able to notify when an option has been selected. When this happens, the `NewTransactionForm` should lookup the details of the selected transaction and fill its inputs with the transaction's information (except Date, of course).

v. In the store, there will be a new slice: `transactionsIndex`.

vi. This new slice will (for now) hold two important pieces of *computed* state:

1. `byBigram`:
   - A map indexed by the bigrams created from the first two letters of every transaction's description. These bigrams then map to the IDs of the corresponding transactions.
   - This index exists to speed up the exact description matching from having to be a linear search by trading off memory for speed.
2. `byWord`:
   - A map indexed by the stemmed words of every transaction's description (of course, with tokenization, normalization, and excluding stop words, etc). These stemmed words then map to the IDs of the corresponding transactions.
   - This index exists to do a more sophisticated search that isn't as naive as just trying to match the exact description letter-for-letter. This will enable information retrieval using things like a Vector Space Model (but at a minimum, something like a boolean model with implicit ANDs).
3. `byWordBigram`:
   - A map indexed by the prefix bigram of all of the words
4. `byId`:
   - A map indexed by transaction ID that then maps to the corresponding bigram and words.

        - This would be used for doing updates/deletions to the index. Need to lookup by ID first to then go delete the ID from the bigram → ID map and word → ID map.

vii. This new slice will have an indexing algorithm whereby it will listen for any actions that affect the `transactions` slice and update the indexes appropriately.

   - e.g. for `create` actions, the indexer will get the new transaction's description, create the bigram, and then add the bigram and ID to the index (although I guess it'll also have to check for existing transactions with the same description to satisfy Acceptance Criteria 7 -- actually, it'd probably be better to do this as part of the search algorithm (i.e. only take the latest transaction with the same description) so that the index can focus on just indexing stuff).
   - This 'indexing algorithm' will probably need to be a series of sagas.
   - The indexing algorithm will have to run on page load after the transactions have been fetched up to create the base indexes.
   - Will probably want to add the `transactionsIndex` slice to `redux-persist`.

viii. Once the indexes have been built, will then need to add actions/sagas to handle searching.

   - The `TransactionsAutocompleteInput` will emit the action to search whenever the user types (probably debounced?) with the query.
   - The saga will then listen for the action, take in the query, and perform a search over the indexes to find IDs, then store those found IDs in another piece of state (in the `forms` slice?).
   - Those IDs can then be mapped to actual transactions using a selector, and then those transactions can be fed back to `TransactionsAutocompleteInput` to fill out the options.

ix. If the user selects one of the options, then another type of action will need to be emitted.

   - This action will need to be listened for by probably a `forms` reducer.
   - The reducer will store the ID as something like `newTransactionFormTransactionId`.
   - The `NewTransactionForm` will then know to lookup the transaction and populate all of the fields with the transaction's data.
   - The `NewTransactionForm` will need to know to do this field population whenever the `newTransactionFormTransactionId` changes -- because the user could select another option.

Implementation Details

- `MaterialAutocompleteInput`
  - New component
- `TransactionsAutocompleteInput`
  - New component
- `NewTransactionsForm`
  - Sub in `TransactionsAutocompleteInput`
  - Modifications to pull the autocompleted transaction and fill out fields
    - This might end up making `NewTransactionForm` very, very similar to `EditTransactionForm`... time to combine them?
- `transactionsIndex.slice`
  - New slice
  - Add it to `redux-persist`
    - Need to figure out the best way to handle calculating the index on page load and reusing the persisted index
- `transactionsIndex.sagas`
  - New sagas
  - `indexOnCreate`, `indexOnSet`, `indexOnUpdate`, `indexOnDelete`
  - `searchTransactions`
    - Takes in query as payload, sends it to the `searchService` along with the indexes, gets results, and emits results to the `newTransactionsFormSearchResults`.
- `forms.slice`
  - Modifications to add the `newTransactionsFormTransactionId`
  - Modifications to add the `newTransactionsFormSearchResults`
  - Maybe time to split `forms.slice` into one for each form?
- `crossSliceSelectors`
  - Add selectors for:

    a. Getting the transaction of `newTransactionsFormTransactionId` -- `getNewTransactionFormTransaction`

    b. Mapping the search result transaction IDs to actual transactions -- `getNewTransactionsFormSearchResults`

- `searchService`
  - New utility service
  - Need to create a `services` folder under `utils` and move the `csvParser` under it