FE State Management

Design

There are different state types, which are consequentially also managed differently.

For this design to properly work, the API has to be “properly” designed (for example, not mix computed evaluation and database-stored user-data).

Slice state

State shared between widgets is stored in `state/slices`. This means state is available offline.

There are the following flavors:

- Configuration: Needs only be loaded once, and never changes (as hard-coded in the backend). Placed in `configuration`.
- Versioned Entities: User-data, which is versioned as per the requirements. Placed in `versionedEntities`. As these entities are versioned, they can be used in the `synchronizationSlice`, which supports offline mode, error handling, …
- “The rest”: Consists out of state only relevant for the UI (e.g. widgets / spaces config) and state which does not need to be versioned (users, settings).

When a UI requires data from multiple entities in the same view (e.g. the project overview requires data from both projects and project evaluations), the aggregation is created in the most appropriate slice. A wrapper (i.e. `{project: ProjectDto, evaluation: ProjectEvaluationDto }` instead of manually mapping individual properties (i.e. `{projectEnd: QuarterYearDto, projectStart: CalculatedQuarterYear}`) saves effort.

API GET requests are typically debounced (multiple widgets requesting the same GETs in a short timeframe will lead to only a single API GET actually sent out).

Evaluation hooks

Not available offline are the so-called evaluations; placed in `state/hooks`.

There are two types of evaluations:

- Calculated based on data stored in the server-side DB: These evaluations are fetched once when the widget is opened.
- Calculated based on data POSTed to the server. These evaluations are fetched whenever the reactivity system detects changes. To keep this efficient, the slices should notably only publish changes if the affected entity has indeed changed content.

These is no caching at the moment for the evaluation hooks. This could of course be improved by tracking data the evaluation depends on and only loading evaluations anew if the dependencies change.

As they are hooks, the evaluation result only exists within a widget; hence each widget also leads to its own API request. If this becomes a bottleneck, a slice which caches evaluations needs to track the dependencies of the evaluation, to avoid stale data shown to the user.

Technical foundation

`versionedEntityAdapter`

Inspired by createEntityAdapter | Redux Toolkit, the `versionedEntityAdapter` provides selectors and reducers for CRUD operations on a particular type of versioned entity. While it largely follows the architecture and naming of Redux' `entityAdapter`, the `versionedEntityAdapter` integrates and abstracts the `versionId` natively; which impacts both the selectors and reducers.

Most selectors and reducers are shared between all usages of the `versionedEntityAdapter`. Usually customized are the `load` reducers which load entities into the store after requesting them from the API: These reducers need to cleanup unused entities (i.e. entities which were removed server-side), relative to the API request (i.e. if entities were only retrieved relative to a specific object, then only entities relative to that specific object should be cleaned up). The `versionedEntityAdapter` provides appropriate reducers for root entities and entities belonging to a project or object.

Common to all the `load` reducers is that they must not clean up entities with non-synchronized changes. The `versionedEntityAdapter` considers this in its interface.

`synchronizationSlice`

The `synchronizationSlice` handles storing changes to the server by providing `persist` and `remove` methods to other slices. It guarantees dependent entities are synchronized in order. Further, it captures errors and, depending on the error, reattempts the action or asks the user how to proceed. It is designed as a separate slice because it has state (the queue of pending changes) and to keep the slices relaying on synchronization simple.

Participating slices need to register the `SynchronizedEntityConfiguration` at the `synchronizationSlice` which declares:

- `entityType` to relate `persist` and `remove` calls with the appropriate config
- `SynchronizedEntityClient` with `put` and `delete` actions to do the requests
- selectors and reducers (e.g. implemented by the `versionedEntityAdapter`) to construct the payload to send to the server, respectively store the server response
- the root entity and dependencies towards other root entities to detect dependencies during synchronization processing
- how to display the impacted entity to the user

The synchronization itself works like this:

- The participating slice (e.g. `objectSlice`) calls `persist` or `remove`. With this, the `synchronizationSlice` queues the selected action.
- Actions are processed in order. If the action succeeds, it is removed from the queue. If the action fails, it is kept in queue, and actions later in the queue are only processed if there are no dependencies to actions earlier.

When an action fails, the failure cause is determined (e.g. network errors, server fault, conflict). Depending on the failure, the user is informed with various levels of emergency:

- A network error shows up as a small symbol top-right. The user can continue working, and the queue will simply reattempt later.
- A server error shows up as a big red button. While the user can continue working, this should incentivize the user to investigate. When the user clicks the button, the synchronization view opens which shows all pending synchronizations and their status. The user is asked to reattempt and/or contact support, as this is always a programming error (in the frontend or the backend).
- A conflict shows up as a overlay modal (i.e. the user cannot continue working). The user is asked to decide whether to override the server version, or discard the local changes.

`debounceThunk`

Wraps a thunk to ensure it is only executed once, with a default TTL of 10s (… after which it will be executed again once requested). To the consumer, the wrapped thunk has exactly the same call signature (i.e. is called the same as the original thunk would be called).

This wrapper is used for idempotent thunks which likely always have the same result (e.g. GET to the list of objects and update the slice).

This is particularly useful with our widget concept, which are (and must be) unaware of other widgets, and therefore also what kind of data these load. The debounce thunks ensure that data is not loaded multiple times, when multiple widgets open with the same dependencies.

`resolvingSelector`

Wraps a selector to declare (thunk) dependencies. This simplifies the calling code, while making it impossible to forget loading dependencies (and hence eradicate these type of bugs).

Thunks are automatically dispatched when the selector is constructed, and the widget is put in loading mode while the thunks resolve. The consuming code for the resolving selector `makeSelectStratusObject` typically looks like this:

```
const selectStratusObject = useMemo(() => makeSelectStratusObject(stratusObjectId), [stratusObjectId])
const stratusObject = useResolvingSelector(spaceId, widgetId, selectStratusObject, stratusObjectId)
```

Note how the resolving selector is recreated when the `stratusObjectId` changes.

If the `stratusObjectId` might not be defined, you can reuse the same resolving selector using the utils `useLaxMemoFactory` (to construct the selector) and `useLaxResolvingSelector` (to consume the selector):

```
const selectStratusObject = useLaxMemoFactory(makeSelectStratusObject, stratusObjectId)
const stratusObject = useLaxResolvingSelector(spaceId, widgetId, selectStratusObject, stratusObjectId)
```