PanelAPI

The `PanelAPI` namespace contains React Hooks and components which allow panel authors to access Lichtblick data and metadata inside their panels. Using these APIs across all panels helps ensure that data appears consistent among panels, and makes it easier for panels to support advanced features (such as multiple simultaneous data sources).

To use PanelAPI, it's recommended that you import the whole namespace, so that all usage sites look consistent, like `PanelAPI.useSomething()`.

```
import * as PanelAPI from "@lichtblick/suite-base/PanelAPI";
```

PanelAPI.useDataSourceInfo()

"Data source info" encapsulates **rarely-changing** metadata about the sources from which Lichtblick is loading data. (A data source might be a local bag file dropped into the browser, or a bag stored on a remote server; see players and dataSources for more details.)

Using this hook inside a panel will cause the panel to re-render automatically when the metadata changes, but this won't happen very often or during playback.

```
 PanelAPI.useDataSourceInfo(): {
  topics: Topic[],
  datatypes: RosDatatypes,
  capabilities: string[],
  startTime?: Time,
  playerId: string,
};
```

PanelAPI.useMessagesByTopic()

`useMessagesByTopic()` is a small wrapper around `PanelAPI.useMessageReducer` (see below). It makes it easy to just request some messages on some topics, without doing any transformations on the messages. This is convenient, but also means that the entire messages are kept in memory, so it's recommended to only use this for a small number of messages at a time (small `historySize`).

Using this hook will cause the panel to re-render when any new messages come in on the requested topics.

```
PanelAPI.useMessagesByTopic(props: {|
  topics: string[],
  historySize: number // Number of messages to keep per topic.
|}): { [topic: string]: Message[] };
```

PanelAPI.useMessageReducer()

`useMessageReducer()` provides panels a way to access messages from topics. `useMessageReducer` is a fairly **low-level API** that many panels will use via `PanelAPI.useMessagesByTopic` (see above). Users can define how to initialize a custom state, and how to update the state based on incoming messages.

Using this hook will cause the panel to re-render when any new messages come in on the requested topics.

```
PanelAPI.useMessageReducer<T>(props: {|
  topics: (string | { topic: string })[],
  restore: (prevState: ?T) => T,
  addMessage: (prevState: T, message: Message) => T,
|}): T;
```

Subscription parameters

- `topics`: set of topics to subscribe to. Changing only the topics will not cause `restore` or `addMessage`/`addMessages` to be called.

Reducer functions

The `useMessageReducer` hook returns a user-defined "state" (`T`). The `restore` and `addMessage`/`addMessages` callbacks specify how to initialize and update the state.

These reducers should be wrapped in `useCallback()`, because the useMessageReducer hook will do extra work when they change, so they should change only when the interpretation of message data is actually changing.

- `restore: (?T) => T`:
    - Called with `undefined` to initialize a new state when the panel first renders, and when the user seeks to a different playback time (at which point Lichtblick automatically clears out state across all panels).
    - Called with the previous state when the `restore` or `addMessage`/`addMessages` reducer functions change. This allows the panel an opportunity to reuse its previous state when a parameter changes, without totally discarding it (as in the case of a seek) and waiting for new messages to come in from the data source.

      For example, a panel that filters some incoming messages can use `restore` to create a filtered value immediately when the filter changes. To implement this, the caller might switch from unfiltered reducers:

      ```
      {
        restore: (x: ?string[]) => (x || []),
      ```

```
  addMessages: (x: string[], msgs: Message[]) => {
    msgs.forEach((m) => x.concat(m.data));
    return x;
  },
}
```

to reducers implementing a filter:

```
{
  restore: (x: ?string[]) => (x ? x.filter(predicate) : []),
  addMessages: (x: string[], msgs: Message[]) => {
    msgs.forEach((m) =>  if (predicate(m.data)) x.concat(m.data));
    return x;
  },
}
```

As soon as the reducers are swapped, the **new** `restore()` will be called with the **previous** data. (If the filter is removed again, the old data that was filtered out can't be magically restored unless it was kept in the state, but hopefully future work to preload data faster than real-time will help us there.)

- `addMessages?: (T, Message[]) => T`: called when any new messages come in on one of the requested topics. Unlike `addMessage`, this callback is provided with every new message since the last call. Optional for back compat with older panels, this is the recommended approach moving forward.
- (DEPRECATED) `addMessage?: (T, Message) => T`: called when any new message comes in on one of the requested topics. The return value from `addMessage` will be the new return value from `useMessageReducer()`. Will not be called if an addMessages callback was provided.
- Note only one of the two optional parameters above must be provided, providng neither or providing both will result in an error.