MD Bookmarks Data flow

Status: Final
Author: tsergeant@chromium.org
(This document is public)

Document History

Objective

We aim to create a model and data-binding system for the MD Bookmarks project which effectively separates back-end logic from UI. We should be able to make changes to the bookmark tree structure easily, and have those changes automatically reflected in the UI components. We do not aim to create a general purpose system, although the ideas here could be useful elsewhere.

Background

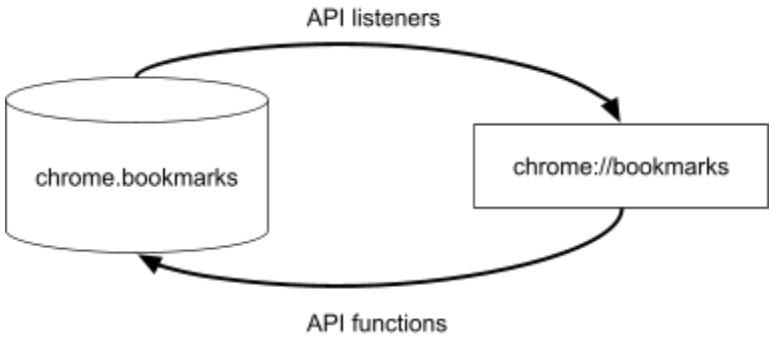
MD Bookmarks project

MD Bookmarks is a rewrite of the Chrome bookmark manager (chrome://bookmarks) following Material Design guidelines and using the Polymer Web Components library. This follows on from existing work: the Downloads (Design Doc), History and Settings pages have all been rewritten in this way.

Bookmarks Extension API

Much of this work is motivated by the specific data structures and APIs that are used by the bookmarks manager. The bookmark manager uses the chrome.bookmarks and chrome.bookmarkManagerPrivate extension APIs to retrieve and modify bookmark state.

These APIs operate on the `BookmarkTreeNode` data structure. Each Node has an *id*, a *title*, a *parentId* pointer, and either a *url* (for items) or an array of *children* (for folders). Modifications are made by calling API functions (eg, `chrome.bookmarks.create`), and then responding to the corresponding listener (`chrome.bookmarks.onCreated`). This listener will be fired regardless of where the modification originated: either on the current page, from the bookmarks bar/menu, or from Chrome sync. This listener will be fired with enough information to update the local page state, which allows the page to stay in sync with the bookmarks backend without performing full data refreshes.



Existing MD Bookmarks data-flow model

By responding to API listeners correctly, it's possible to keep the JavaScript objects representing the current page state in sync with the bookmarks backend. However, automatically reflecting changes to those JS objects back to HTML-based UI requires extra work.

As we were already using Polymer for UI elements, we initially decided to leverage Polymer's databinding system to perform this reflection. The initial design was to create a <bookmarks-store> element, which is a single source of truth for the local copy of the bookmarks tree. The store is responsible for performing *all* modifications made to that tree, and UI components are one-way bound to the tree state: they (almost) never modify state directly.

This design worked well to begin with, but as time went on we started to see some problems. It was difficult for us to maintain the tree state effectively, while working around the limitations of the Polymer data binding system and managing two different types of results (main tree results and search results).

One-way data flow models

One-way data flow is a model which has been recently popularised by React, Flux/Redux and friends. In particular, Redux is a tiny library focused around three principles:

- The page state should have a single source of truth
- UI elements should have read-only access to the page state
- Changes to the page state are made with pure functions

We think that sticking to these principles has the potential to simplify the Bookmarks manager code.

Overview

In this section, we propose a way to rewrite the Bookmarks data-binding system to adhere more closely to the principles of Redux. In the next section, we'll explore some of the more gritty details of implementing individual features in this new model.

There's a WIP CL for this design at https://codereview.chromium.org/2704983002/.

Store

This is a plain-JS singleton (`bookmarks.Store.getInstance()`) which stores the current state tree. It has the following API:

```
bookmarks.Store:
  init(initialData: Object)
  get data(): Object
  addObserver(observer: StoreClient)
  removeObserver(observer: StoreClient)
  handleAction(action: Object)
  handleDeferredAction(action: function(function(Object)))
```

Notably:

- `Store.data` is the state tree for the entire application, including tree nodes, selected items, search terms, etc.
- `data` is publically readable but not writable. The only way to modify it is through `handleAction`.
- `handleAction` is called with an `Action`, which is an object with a name field. `handleAction` will use the `Action` object to produce a new data object with the state of the world after that action. `Store` then notifies observers that the page state has changed. Actions are the only way for page state to be modified.
- `handleDeferredAction` is a used to dispatch Actions asynchronously (see 'Deferred Actions' below).

Handling actions with reducers

Actions are handled using *pure functions* called *Reducers*. A reducer (in the same sense as the functional programming primitive reduce) takes a state and an action and produces a new state.

Reducer functions *must* be pure. They should not mutate existing objects: instead, they create new objects with any required changes implemented. They must not make any API calls, or touch the DOM, or use Math.random, or anything else which could cause the same input to produce a different output.

Reducer functions *may* implement business logic, but they do not have to. It is acceptable to require the action to include extra details to remove complexity from the reducer.

Reducer functions *should* avoid creating new objects for things which do not change. Unnecessary copies waste memory and can cause extra work at the UI layer.

The root reducer in bookmarks is `bookmarks.reduceAction`, which then calls into specialised reducer functions to update individual parts of the state tree.

StoreClient

bookmarks.StoreClient is a Polymer behavior which ties the Polymer UI layer to the data storage layer. Every UI element which needs to pull data from the store will implement this behavior.

```
bookmarks.StoreClient:
  watch(localProperty: String, valueGetter: (Object) => Object)
  dispatch(action: Object)
  dispatchAsync(action: function(function(Object)))
  updateFromStore()
  onStateChanged(newState: Object)
```

`watch` is the mechanism through which Polymer properties are tied into the store. For example,

```
this.watch('item', (state) => state.nodes[this.itemId]);
```

Says "whenever the store changes, copy `store.selectedFolder` into `this.selectedFolder` and update the UI". Importantly, the Polymer property is only updated if the value (or object reference) has changed:

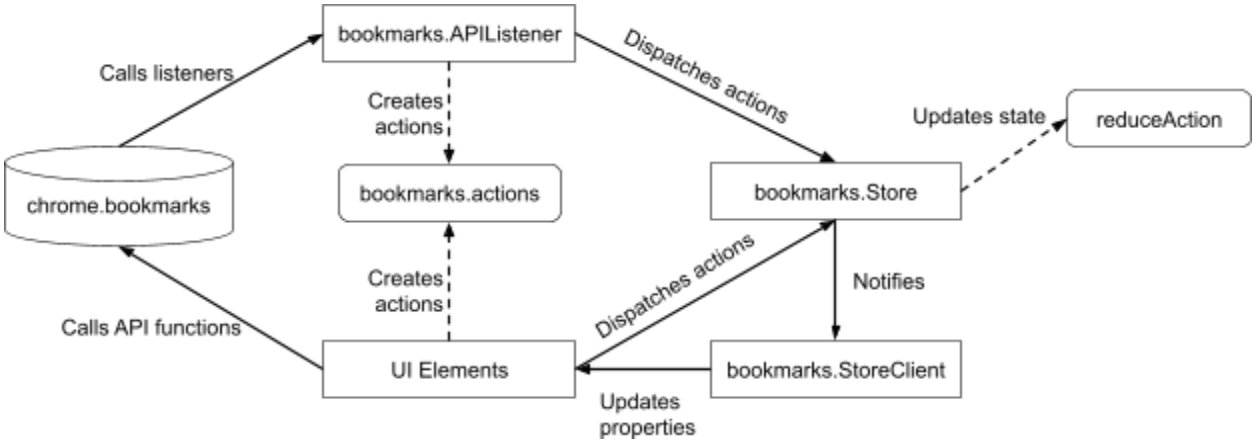
```
var oldValue = this[watch.localProperty];
var newValue = watch.valueGetter(newState);

if (oldValue == newValue)
  return;
this[watch.localProperty] = newValue;
```

Since our reducers only produce new objects when necessary, Polymer should only update when the underlying values have changed.

Detailed Design

Data flowchart



State tree structure

```
MdBookmarksState:
    nodes: Object<String, BookmarkNode>
    selectedFolder: String
    closedFolders: Set<String>
    search:
        term: String
        inProgress: Boolean
        results: Array<String>
    selection:
        items: Set<String>
        anchor: String
```

Note that nodes is flattened out to a Map keyed by Node ID (which we had already done in idToNodeMap). Each node has its children replaced with an Array of IDs . Normalising the data in this way makes it easier to access and modify.

Initialization

The store should be initialized in one go, using Store.init(). This should be called with the entire initial state of the world, including:

- The normalized tree structure from chrome.bookmarks.getTree()
- The selected folder or search term from the URL, if there is one
- The folder open status from localstorage

These initialization tasks can be kicked off as the page is loading. No actions are allowed before init() has been called, so we should hide the UI (in a similar way to MD History) before this.

Writing UI elements

At this stage, it's useful to consider how an existing UI element needs to change to fit into the new system. Previously, <bookmarks-item> needed to be provided with an entire item for it to render. Now it just needs an item ID:

```
<bookmarks-item item-id="[[id]]"></bookmarks-item>
```

Using this ID, it is able to update the item for itself whenever the item changes:

```
attached: function() {
  this.watch('item', (store) => store.nodes[this.itemId]);
  this.watch('isSelectedItem',
      (store) => store.selection.items[this.itemId])
  this.updateFromStore();
},
```

And since the ID can be changed externally, we need to update from the store whenever it changes:

```
properties: {
  itemId: {
    type: String,
    observer: 'updateFromStore',
  },
  ...
},
```

That's it! As before, `item` can be bound to UI and will update whenever the store changes.

Writing actions

Actions should be created by functions which live in a shared `actions.js` file. These can be very simple, or if necessary, they can include logic based on the current state of the store:

```
function selectItems(store, baseId, isAdd, isRange) {
  var toSelect = [];
  // Look up the store to determine the items that need to be
  // selected.
```

```
  return {
    name: 'select-folder',
    items: toSelect,
  };
}
```

Creating actions like this has the useful side-effect of documenting all possible actions and their parameters.

Deferred Actions

We also support 'DeferredActions', which are special in a couple of ways:

- DeferredActions can group multiple action dispatches into a single action creator function.
- DeferredActions allow action creators to perform work asynchronously before dispatching the action (useful for getting results from the chrome.bookmarks APIs).

To write a DeferredAction, return a function which has a single callback parameter from your action creator.

```
function setSearchTerm(term) {
  return function(dispatch) {
    // dispatch can be at any time to dispatch an action to the Store
    dispatch(startSearch(term));

    // dispatch can be called multiple times (even asynchronously).
    chrome.bookmarks.search(term, function(results) {
      dispatch(finishSearch(results));
    });
  }
}
```

These are then dispatched using the special functions `StoreClient.dispatchAsync` and `Store.handleDeferredAction`.

Writing reducers

Reducers live in a shared `reducers.js` file. The root reducer divides work up between the different subtrees:

```
function reduceAction(state, action) {
```

```
  return {
    nodes: bookmarks.NodeState.updateNodes(state.nodes, action),
    search: bookmarks.SearchState.updateSearch(state.search, action),
    ...
  };
}
```

Each subtree should be treated independently. For example, updates to store.nodes are made in the bookmarks.NodeState module. This module should not need to look at other parts of the state tree.

When writing reducers, the Object.assign method is a useful helper for creating a new Object with part of the subtree changed, but leaving everything else untouched:

```
return Object.assign({}, node, {'title': changeInfo.title})
```

Key subsystems

Search

Search is implemented with three actions:

- start-search: sets search.term and search.inProgress
- finish-search: sets search.results and unsets search.inProgress
- clear-search: unsets all search values

These three actions are created by the one function: `bookmarks.actions.setSearchTerm(term)`. Depending on the term, it will create a start or clear action, then asynchronously (using a deferred action) perform a search and generate a `finish-search` action when results are available.

Selection

Selection is implemented as a Set of IDs of all selected items. This is cleared after any action which causes the display to change (select-folder, search-results). The select-items action adds additional items to the selection.

UI elements can check if an ID is selected by listening to `store.selection.items.has(itemID)`, which is a constant-time operation whenever the store updates. Similarly, UI elements can listen to `store.selection.items.size` to determine how many items are selected.

API Listeners

Chrome.bookmarks API listeners are no longer registered in the store directly. Instead, we will have a new module, `bookmarks.ApiListener`, which listens to the API and translates calls to the listeners into actions which are dispatched to the store.

Performance analysis

Initial analysis indicates that the new system performs similarly to the old system. Measurements were taken in a folder with ~500 bookmark items, which is particularly relevant when deleting an item, as it means all other items must be moved.

Test	Old System	New system
Deleting an item	80ms (but could be reduced to <10ms with some refactoring)	70ms
Ctrl-Selecting an item	3-5ms	3-5ms
Editing an item	3-5ms	3-5ms

It's promising that we have similar performance with little effort. However, we should continue to monitor this by:

- Monitoring the number of elements that need to observe the store
- Only performing constant-time operations in the valueGetter() critical path

Code Location

Code location: chrome/bookmarks/resources/md_bookmarks

- store.js: Definition of bookmarks.BookmarksStore
- actions.js: Definitions of bookmarks.actions.*
- store_client.js: Definition of bookmarks.StoreClient. Depends on store.js
- reducers.js: Definition of bookmarks.*State reducer functions
- api_listener.js: Definition of bookmarks.ApiListener. Depends on store.js/actions.js

Test code: chrome/test/data/webui/md_bookmarks

Caveats/Alternative Approaches

We see this model as having several advantages:

- We rely less on Polymer specifics: We no longer need notifyPath or linkPaths.

- Many operations are conceptually simpler, since we can either operate on a list of IDs or on nodes directly.
- Search can be treated the same as displaying a regular list of results, since both are just a list of IDs.
- Page UI state is no longer coupled to BookmarkTreeNodes (eg, we separate out folder open/closed state), which makes bulk-updating the tree easier
- It is easier to test how the page state changes, since reducers are pure functions. There's no more uncertainty about whether our path notifications are *actually* correct.

However, it has significant caveats:

- We are reimplementing part of the data-binding system for ourselves. Our system will undoubtedly run into problems that either Polymer or Redux have already solved.
- Writing correct reducers involves being very careful about copying/mutating data
- We are preventing Polymer from performing certain performance optimisations
    - This particularly affects arrays, where dom-repeat and iron-list rely on receiving splice notifications to optimise rendering. Replacing the array on each update means that we no longer send splices. It is possible to work around this, but there's no particularly nice solution.

We think the tradeoffs make sense in this case, due to the type of data that we have (a database of ID-keyed nodes) and the type of operations that we want to do (modifying/rearranging those nodes). This approach may not make sense for other pages with different data structures/actions (eg: History).

Alternatives considered

Two-way data binding

Two-way data-binding is the well-lit path for Polymer applications. However, due to the nature of the data structure and API, we would not benefit greatly from this: we would still need a centralised place to perform updates in response to API listeners and would run into the same problems with path notifications.

Improve Polymer-based one-way binding

We would be able to substantially improve our current system by careful refactoring. Specific improvements:

- Move tree modifications into helper functions
- Pull item selection into a separate module
- Create a reloadSubtree() function which can be used to update sections of the tree

Test Plan

Unit tests:

- Existing store tests are replaced with unit tests of everything in `actions.js` and `reducers.js`. These are easy to test, since everything in them is a pure function.
- We should migrate existing tests of UI elements to use a TestStore, which would allow a state to be specified and to check that the correct actions are dispatched.

Integration tests:

- We should add an integration test for Store/StoreClient
- And add the UI/backend integration tests which we've been talking about for a while

Document History

- 2017-05-03: Updated with explanation of deferred actions, which we added in this CL, and changes selection and closedFolders to Sets.
- 2017-03-15: Updated with some API changes that we made during code review. At this stage the code has almost entirely landed, so this document can be considered 'final'.
- 2017-02-28: Updates to "near-final draft"
- 2017-02-22: First draft