Although the new context api and react hooks are great, it is almost certain there will be a need for state management at some point. There are many libraries dealing with this problem:

- based on redux:
    - redux ducks
    - rematch
    - easy-peasy
    - redux saga
- other:
    - mobx

They all have their strengths and weaknesses and comparing one to another is beyond the point of this document. The "dashboard app" will use "redux ducks" for now since is the only one not needing an external library (other than redux and react-redux which the other libraries use as well). It is indeed verbose and opinionated but it also conveys full transparency over how things are done.

Next in line would be rematch since it does away with types and actions keeping the whole redux experience cleaner while at the same time offering structure, code organisation and separation of concerns between views and state.

Redux ducks intro

The application uses as state management a flavour of redux called “redux ducks" https://github.com/erikras/ducks-modular-redux.

If you don’t know much about redux, think about it as a “database" in the browser which only lives until the page is refreshed. Changes to the database are done by functions called actions and reducers. The principles are pretty simple but the whole thing can get very messy very fast.

What redux "ducks" provides:

- a way to organise everything
- scalability
- complete ui decoupling
- api service
- developer sanity

Redux ducks does not come with any external libraries which do something behind the scenes. Its sole purpose is to organise the state management workflow in a way that is it is easy to manage, scale and test.

Redux ducks overview

State management is completely decoupled from the UI and can be found in each of the projects in the “redux folder".

What’s in a duck

In a duck you can find the following files:

The name of the “duck" should reflect the name of the data model. In this case we use mock data so we called it “dummylist".

types.js

```
// types.js
// these are the types of actions that will be dispacthed by our actions f
// the name of the action should be a verb + a description of the action
// in this case:
//   - fetch the dummy data
//   - fetch the dummy data has been completed

const FETCH_DUMMY_LIST = "FETCH_DUMMY_LIST";
const FETCH_DUMMY_LIST_COMPLETED = "FETCH_DUMMY_LIST_COMPLETED";

export default {
    FETCH_DUMMY_LIST,
    FETCH_DUMMY_LIST_COMPLETED,
};
```

actions.js

```
import types from "./types";

// an action is a function which returns a standard object
const fetchDummyList = ( params ) => {
    const { baseUrl, path } = params;

    return {
        type: types.FETCH_DUMMY_LIST, // the type of the action
        async: true, // async set to true if API is involved
        payload: { // payload holds the data that is being passed to the r
            baseUrl,
            path,
        },
    };
};

export default {
    fetchDummyList, // export the action
};
```

reducers.js

```
import types from "./types";

const initialState = [];
// reducers are pure javascript functions which return the an altered stat
// in this case we are dealing with a list represented by an Array
const dummyListReducer = ( state = initialState, action ) => {
    switch ( action.type ) { // using the switch operator we iterate throu

      case types.FETCH_DUMMY_LIST_COMPLETED: {
          // the trick is to return a new state
          // in this case we destructure the data received from the server
          // and return a new array

          return [ ...action.payload ];
      }

      default:
          return state;
      }
};

export default dummyListReducer;
```

operations.js

```
import actions from "./actions";

// the main feature of the operations is to export the actions
// but it can be used to chain multiple actions. We don't have
// an example for that right now but this document will be updated
const { fetchDummyList } = actions;

export default {
    fetchDummyList,
};
```

index.js

```
import reducer from "./reducers";

// export operations so they can be used in the views
export { default as dummylistOperations } from "./operations";

// export the reducer so it can be used in the store
export default reducer;
```

Note: every duck must be exported via the `/redux/ducks/index.js` file:

```
export { default as dummylist } from "./dummylist";
```

Store.js

The main purpose of the store is to combine all the reducers, middlewares and expose the store to the application:

```
import { createStore, applyMiddleware, combineReducers, compose } from "re
import thunkMiddleware from "redux-thunk";
import * as reducers from "./ducks";
import apiService from "./middlewares/apiService";

export default function configureStore( initialState ) {
    // combine reducers
    const rootReducer = combineReducers( reducers );

    // use Redux Inspector in chrome devtools
    const composeEnhancers = devToolsInstalled()
        ? window.__REDUX_DEVTOOLS_EXTENSION_COMPOSE__
        : compose;

    return createStore(
        rootReducer,
        initialState,
        composeEnhancers( applyMiddleware( apiService, thunkMiddleware ) )
    );
}
```

API communication

API communication is handled by the `/redux/middlewares/apiService.js` middleware.

```
const apiService = () => next => action => {
    const result = next( action );

    // all actions will pass through this function and if the async flag
    // is NOT set the action will be passed on to the next() function.
    if ( !action.async ) {
        return result;
    }

    ...
}
```

Important to note here is that based on the outcome of the request, actions will be dispatched in order to be handled by the reducers.

```
// request has returned an error:
function handleErrors( err, action, next ) {
    next( {
        type: `${ action.type }_FAILED`,
        async: true,
        generalFetching: action.generalFetching,
        pageFetching: action.pageFetching,
        payload: err,
    } );

    return Promise.reject( err );
}

// request was successful:
function handleResponse( res, action, next ) {
    next( {
        type: `${ action.type }_COMPLETED`,
        async: true,
        payload: res,
    } );

    return res;
}
```

How to use redux in views

Follow these easy steps:

- import operations file from redux folder and connect function from react-redux

```
import { connect } from "react-redux";
import { dummylistOperations } from "../../redux/ducks/dummylist";
```

- expose the state and the action to the component this way:

```
const mapStateToProps = state => ( {
    dummylist: state.dummylist,
} );

const mapDispatchToProps = {
    fetchDummyList: dummylistOperations.fetchDummyList,
};

export default connect( mapStateToProps, mapDispatchToProps )( Homepag
```