# One quick look at how to create a Redux-connected React component / feature

2019-10-23 version by JV – **UNDER REFINING – Though possibly never final.**

**/actions/ActionTypes.js**: Defining three new allowed action types / signal types / message types / redux event types. Developers must use these => no spelling mistakes happen.

- _REQ = (AJAX) requested   _OK = success back    _X = failure back.

  These are not any standards, just Juhani's convention. Short, and thus easy to spot in the *Redux Dev Tool*

```
 CATEGORIES_SEARCH_REQ:   'CATEGORIES_SEARCH_REQ',
 CATEGORIES_SEARCH_OK:    'CATEGORIES_SEARCH_OK',
 CATEGORIES_SEARCH_X:     'CATEGORIES_SEARCH_X',
```

**/actions/category.js**: Defining three new allowed ways (only allowed ways) to create action objects / signals, signal objects / messages, message objects

```
// ACTION CREATORS (Action object creator functions)

// CATEGORIES SEARCH BY KEYWORD
export const categoriesSearch_REQ = (keyword) => ({
  type: ActionTypes.CATEGORIES_SEARCH_REQ,
  keyword: keyword,
});
export const categoriesSearch_OK = (categoryList) => ({
  type: ActionTypes.CATEGORIES_SEARCH_OK,
  categoryList: categoryList
});
export const categoriesSearch_X = () => ({
  type: ActionTypes.CATEGORIES_SEARCH_X,
});
```

**/actions/category.js**: Defining one *Redux service function* that redux-connected React components can use to make an intelligent search (from backend DB to the frontend Redux store). In the picture this function is called *Action handler*, good or bad name.

- Exported for other modules (Redux-connected React components) to use
- Asynchronous as React <-> Redux connection is asynchronous
- We signal what we are doing, and do it ( _REQ + Axios **AJAX request** ). In the case of other Redux actions the dispatch both signals and does it ( _OK(payload) / _X ).

```
export function categoriesSearchByKeyword(keyword) {

  return async (dispatch, getState) => {

    dispatch(categoriesSearch_REQ(keyword));

    const ajaxRequest = {
      method: 'get',
      url: API_ROOT + `/category/search/${keyword}`,
    };

    axios(ajaxRequest)
      .then((response) => {
        dispatch(categoriesSearch_OK(response.data));
      })
      .catch((error) => {
        console.error("Error: " + error);
        dispatch(categoriesSearch_X());
      })
      .then(() => {
        return {
          type: null
        }; // 'Empty' action object
      });
  }
};
```

- (((At the end the empty action that is returned will be handled by the reducer, but as the type is null, no action is taken. Without this, we might get error messages to console. When will that empty dummy Action object be dispatched and handled? Well, look later when the React component will dispatch a call to this categoriesSearchByKeyword.

  dispatch( categoriesSearchByKeyword(someKeyword) );

)))

**/reducers/category.js**: We add a new category list for the search results to the redux store initial state. (((Later we could make this more elegant and just use the list of found id:s (if our categoryList has all Categories from database) )))

```
import ActionTypes from '../actions/ActionTypes';

// Define initial state template for the reducer, to create the store state fraction once
export const initialState = {
    isLoading: false,
    categoryList: [],
    categorySearchList: [],    <= just this place/container here was added to Redux store
    categoryIdsFound: null,
    categoryCurrent: null,
};
```

**/reducers/category.js**: We add there new handlers to the **categories** reducer. (The name of the reducer (function) will be the name of the subarea/object in the Redux store state: *state.**categories**.categorySearchList , not e.g. state.members.memberSearchList )*

- These cases tell what the reducer should do when it receives any of these three actions. Like the picture states, reducer gets the old state and new action, and returns the new state
- Redux puts that new state to the store auto-magically (that code not visible to use, we just see in /src/index.js that the store and the reducers are connected).

```
export default function categories(state = initialState, action) {

   switch (action.type) {

       ... other cases...
       case ActionTypes.CATEGORIES_SEARCH_REQ:
        return {
            ...state,
            isLoading: true,
        };
       case ActionTypes.CATEGORIES_SEARCH_OK:
        return {
            ...state,
            categorySearchList: action.categoryList,
            isLoading: false,
        };
       case ActionTypes.CATEGORIES_SEARCH_X:
        return {
            ...state,
            isLoading: false,
        };
        ... other cases...
```

**Questions about reducer(s)**

**When will we call the reducer?**

- Never. We dispatch Actions, Redux environment calls the reducer.

**How can we later update the initialState properties, when the application is running?**

- We can’t. initialState just used to create the initialState to the Redux store. After that we dispatch Actions that match some case in some reducer => new store state (fraction) to the state.

**What happens when the switch case of the reducer code returns a new state object (fraction of the whole store state)? Where do we put it?**

- We won’t get it, the Redux environment uses that object to update the store. Just give it the object it needs!

**Use the Redux Dev tools! Go back in history = select previous Action in the list. Always pay attention to what is shown in the other window (Action object | or State at that point | or Difference to the previous Action)**

**(parts of) /components/categoryComponents/CategoriesIntelligentSearch.jsx**: Simplest possible, yet enlightening example of a redux-connected React component. It's both data-bound (data refreshed automatically from Redux store) and action-bound (events/actions dispatched to the Redux code.

- First the component looks like any other React component. It just strangely relies on props that are not given by the parent component, but appear to fall from the sky.

```
import React, { Component } from 'react';
import CategoryItem from './CategoryItem';

// One of the simplest possible sensible Redux-connected React examples:

class CategoriesIntellingentSearch extends Component {

  componentDidMount() {
    let keyword = "Training";
    this.props.categoriesSearchByKeywordLocal(keyword);
  }

  render() {
    let categorySearchList = this.props.categories.categorySearchList;
    return (
      <div>
        <h2>Found Categories based on keyword</h2>
        <div>
        {
          categorySearchList.map(
            (item, index) => (<CategoryItem item={item} />)
          )
        }
        </div>
      </div>
    );
  }
}
```

- The `CategoryItem` child component is a **so-called presentational component**. It merely presents=shows the data the parent feeds it via its props, here via the prop 'item'
- The parent `CategoriesIntellingentSearch` will be ***later*** called a connected component.

**(the grandparent component) /views/categoryViews/Categories.jsx**: Simply uses the CategoriesIntelligentSearch component, but does **not** provide any props to it! Mystery about those props intensifies!

```
        <h2>My Categories</h2>
        <CategoriesIntellingentSearch />        <= no prop values injected here!
```

**(rest of) /components/categoryComponents/CategoriesIntelligentSearch.jsx**: though has the answer to the mystery. So, how does the `CategoriesIntellingentSearch` get those above underlined prop values? The hints start already from the imports:

```
// connect function from a npm module for connecting react and redux
import { connect } from 'react-redux';

// action handler, the function we created before!
import {categoriesSearchByKeyword} from '../../actions/category';

// Creating a function returning a definition object describing how Redux environment
// should connect the Redux store state values to this component's props!

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

// a function returning a definition object describing how Redux should dispatch the local
// prop function calls to Redux code function calls (async).
const mapDispatchToProps = dispatch => ({
  categoriesSearchByKeywordLocal: (keyword) => {
    dispatch(categoriesSearchByKeyword(keyword));
  },
});

// Those two function above solved the mystery. Here is how we will get those two props!

// Finally, what will be returned below is not the React component, but the React
// component after it has bee connected to the Redux store data, and our Redux code
// functions! (Somebody could call it Redux 'decorated' React component)

export default connect(mapStateToProps, mapDispatchToProps)(CategoriesIntellingentSearch);
```

Kind of what happens on the last line:    (a bit advanced topic)

```
connect(mapStateToProps, mapDispatchToProps) // First step, this executed returns a func

export default foo(CategoriesIntellingentSearch); // Second step, calling that func 'foo'

export default CategoriesIntellingentSearchConnected; // Final step, exporting the object
returned by 'foo'
```