Making the ColorPicker Dialog Reusable

Using the Dialog Result Value

Unfortunately, now we have a problem. We can forward the current color value to the `ColorPickerDialog` as part of the "description" that we're storing in state, and use that as the initial color value in the dialog. **However, we need some way to not only retrieve the final color value when the user clicks the "Select" button, but also actually use it to update the right field in the unit info reducer.**

The obvious solution is to simply pass a callback function as another prop to the dialog, but that means we'd be storing the callback function in the Redux store. Per the Redux FAQ, putting non-serializable values in the store should be avoided. Now, at the technical level, doing this *would* work, but it would likely cause issues with time-travel debugging. **It's also not the "right" way to do things with Redux.** So, what can we do instead?

Earlier, I linked a previous post I'd written on handling return values from generic "picker" modals. The basic idea is to **have the code that requested the modal also include a pre-built action object as a prop for the dialog. When the dialog succeeds, it dispatches that pre-built action, with its "return value" attached.** It *is* a level of indirection, but it allows us to continue following the Redux principles.

> **Commit 9b2b7ef: Allow ColorPicker to dispatch pre-built actions after selection**

common/components/ColorPicker/colorPickerActions.js

```
import _ from "lodash";

import { openModal } from "features/modals/modalActions";
```

```
import { openModal } from "features/modals/modalActions";

export function showColorPicker(initialColor, onColorPickedAction) {
    // Define props that we want to "pass" to the ColorPicker dialog,
    // including the body of the action that should be dispatched when
    // the dialog is actually used to select a color.
    const colorPickerProps = {
        color : initialColor,
        onColorPicked : onColorPickedAction
    };
    return openModal("ColorPickerDialog", colorPickerProps);
}

export function colorSelected(color, actionToDispatch) {
    return (dispatch) => {
        if(actionToDispatch) {
            const newAction = _.cloneDeep(actionToDispatch);
            newAction.payload.color = color;

            dispatch(newAction);
        }
    }
}
```

We update the `showColorPicker()` action creator to take a second argument - the action that the caller wants dispatched upon success. We also add a thunk that the dialog can call to handle dispatching that arbitrary action. In the process, we also clone the entire action object just to be really sure that we're not accidentally mutating whatever `payload` is. (It's probably not necessary given how this is likely to be called, but it won't hurt to clone the action here.)

common/components/ColorPicker/ColorPickerDialog.jsx

```
import {closeModal} from "features/modals/modalActions";
-import {noop} from "common/utils/clientUtils";
+import {colorSelected} from "./colorPickerActions";

-const actions = {closeModal};
+const actions = {closeModal, colorSelected};

export class ColorPickerDialog extends Component {

// skip ahead
    onSelectClicked = () => {
-       this.props.colorSelected(this.state.color);
```

```
+        this.props.colorSelected(this.state.color, this.props.onColorPicke
d);

         this.props.closeModal();
     }
```

We update the click handler to call `props.colorSelected()` with both the new color value and the previously-supplied action object.

Now, all we need to do is have the `UserInfo` component pass along an appropriate action when we click the color button, and have a reducer case to handle that action.

> **Commit 1ca6fd8: Implement logic to set the unit color from a ColorPicker**

features/unitInfo/unitInfoActions.js

```
-import {UNIT_INFO_UPDATE} from "./unitInfoConstants";
+import {
+   UNIT_INFO_UPDATE,
+   UNIT_INFO_SET_COLOR,
+} from "./unitInfoConstants";

export function updateUnitInfo(values) {
    return {
        type : UNIT_INFO_UPDATE,
        payload : values,
    };
}

+export function setUnitColor(color) {
+   return {
+       type : UNIT_INFO_SET_COLOR,
+       payload : {color}
+   };
+}
```

I've mostly skipped showing action creators to save space in the posts, but we'll show the `setUnitColor()` action creator just to emphasize that it's a simple action creator, nothing special at all.

features/unitInfo/unitInfoReducer.js

features/unitInfo/unitInfoReducer.js

```
-function setUnitColor(state, payload) {
+   const {color} = payload;
+
+   return {
+       ...state,
+       color
+   };
+}

export default createReducer(initialState, {
    [DATA_LOADED] : dataLoaded,
    [UNIT_INFO_UPDATE] : updateUnitInfo,
+   [UNIT_INFO_SET_COLOR] : setUnitColor,
});
```

The new case reducer is also very simple - we extract the new `color` variable and apply that to our unit info state.

features/unitInfo/UnitInfo/UnitInfo.jsx

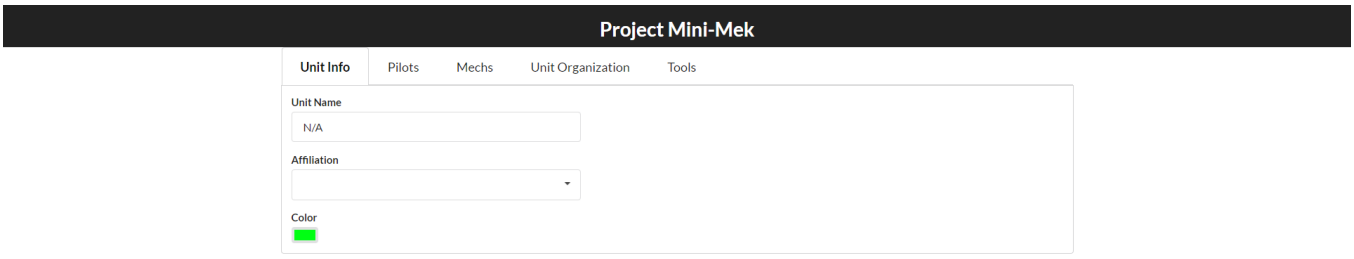
```
import {selectUnitInfo} from "../unitInfoSelectors";
-import {updateUnitInfo} from "../unitInfoActions";
+import {updateUnitInfo, setUnitColor} from "../unitInfoActions";
import {showColorPicker} from "common/components/ColorPicker/colorPickerAc
tions";
import {getValueFromEvent} from "common/utils/clientUtils";

// skip ahead

    onColorClicked = () => {
-       this.props.showColorPicker(this.props.unitInfo.color);
+       const onColorPickedAction = setUnitColor();
+
+       this.props.showColorPicker(this.props.unitInfo.color, onColorPicke
dAction);
    }
```

And finally, we update the click handler for the color button. We create our "pre-built" action using the `setUnitColor()` action creator, but instead of dispatching it, we pass it to `showColorPicker()` so it can be included as a prop to the `ColorPickerDialog`.

Let's try it out. The color field in our sample data is `"blue"`, while the default color prop to the `ColorPickerDialog` is `"red"`. Let's try changing the color to something green, and confirm that it shows up in the color picker button in the unit info form:



Success! We were able to have the `UnitInfo` component request that the dialog be shown, pass along its own pre-built action, have the `ColorPickerDialog` dispatch, and update the unit info state with the result value from the dialog.

An Alternative Approach to Dialog Results

The Redux ecosystem includes libraries for almost every use case, and that includes modals. There's a variety of existing libraries for Redux-connected modals available. O**ne particularly interesting library is redux-promising-modals**. I haven't yet used it myself, but reading the docs, **it appears to offer another valid solution to the question of handling return values from dialogs without breaking Redux principles**.

`redux-promising-modals` offers actions and a reducer for tracking open modals, very similar to what we just implemented (but does not include any components). However, it also includes a middleware. Whenever you dispatch a `PUSH_MODAL_WINDOW` action, the middleware returns a promise, and tracks what modal that promise belongs to. You can then use that promise in the code that called `dispatch()`, and chain off of it. When that modal is closed, the middleware extracts results from the action, and resolves the original promise, thus supplying the "return values" from the dialog.

Here's an (untested) example of what using `redux-promising-modals` might look like:

```
import {pushModal} from "redux-promising-modals"

function showColorPickerForUnitInfo(initialColor) {
```

```
    return (dispatch) => {}
        dispatch(pushModal("ColorPickerDialog", {color : initialColor}))
            .then( (resultColor) => {
                dispatch(setUnitInfoColor(resultColor));
            });
    }
}
```

To me, this looks like an excellent use of the Redux middleware pipeline for intercepting actions, and is a nicely pre-built solution to the problem of tracking open modals and handling modal return values.