Adding Modal Dialog Support

As you may have noticed, this course isn't really meant to result in a fully-working meaningful application. It's primarily intended to give me reasons to show off specific useful React and Redux techniques. That means that there's a number of things that are going to be obvious over-engineering. Like, say, modal dialogs. This app doesn't *need* modal dialogs, but guess what: we're going to add modal dialogs to ***Project Mini-Mek*** anyway :)

The basic concepts of handling modals in React and driving them from Redux have been described many times elsewhere, in excellent detail. My links list has many articles on the topic, but here's a selected reading list:

- Dave Ceddia's post on Modal Dialogs in React demonstrates the basic approach for rendering and control a modal.
- Dan Abramov's Stack Overflow answer on how to control modal dialogs using Redux shows how to write reducers that contain descriptions of a modal, and a central React component to render it.
- David Gilbertson's post Modals in React, which discusses the tradeoffs of different approaches for putting modals on top of the rest of the app
- Mike Vercoelen's article Scalable Modals with React and Redux, which shows Dan's technique for Redux-driven modals in practice
- And finally, my own prior blog post on handling return values from generic "picker" modals

We're going to put these concepts into action, and even expand on them in some ways.

Driving React Modals from Redux

Let's start by putting together the pieces needed to show a single modal dialog. We're going to need a few things:

- A `ModalManager` component that takes a description of what modal component to show, and what props the modal should receive, plus a lookup table of available modal components, and renders the right modal component from that description
- Actions and reducers that store and clear the description for the current modal
- An actual modal component to show

> **Commit a19f234: Add basic handling for a single modal dialog**

features/modals/modalsReducer.js

```
import {
    MODAL_CLOSE,
    MODAL_OPEN
} from "./modalConstants";

import {createReducer} from "common/utils/reducerUtils";

const initialState = null;

export function openModal(state, payload) {
    const {modalType, modalProps} = payload;
    return {modalType, modalProps};
}

export function closeModal(state, payload) {
    return null;
}

export default createReducer(initialState,  {
    [MODAL_OPEN] : openModal,
    [MODAL_CLOSE] : closeModal
});
```

The reducer logic is trivial. We're simply going to store an object that contains the name of the modal type and the props it should receive, and either set the value or clear it out.

features/modals/TestModal.jsx

```
import React, {Component} from "react";
import {connect} from "react-redux";

import {
    Modal,
} from "semantic-ui-react";

import {closeModal} from "features/modals/modalActions";

const actions = {closeModal};

export class TestModal extends Component {
    render() {
        return (
            <Modal
                closeIcon="close"
                open={true}
                onClose={this.props.closeModal}
            >
                <Modal.Header>Modal #1</Modal.Header>
                <Modal.Content image>
                    <Modal.Description>
                        <p>This is a modal dialog.  Pretty neat, huh?</p>
                    </Modal.Description>
                </Modal.Content>
                <Modal.Actions>
                </Modal.Actions>
            </Modal>
        )
    }
}

export default connect(null, actions)(TestModal);
```

Semantic-UI-React conveniently has a `Modal` class already, which allows you to render headers, content, and action buttons. Since the focus of this article is on how to manipulate modals via React and Redux, and not specifically how to build them from scratch, we'll use the SUI-React `Modal` class rather than trying to build our own.

Note that we pass an `open={true}` prop to `<Modal>`. That's because the SUI-React `Modal` can be either opened or closed, and in our case, whenever we show a `<TestModal>`, we want the inner `<Modal>` to be displayed.

features/modals/ModalManager.jsx

```
import React, {Component} from "react";
import {connect} from "react-redux";

import TestModal from "./TestModal";

const modalComponentLookupTable = {
    TestModal
};

const mapState = (state) => ({currentModal : state.modals});

export class ModalManager extends Component {
    render() {
        const {currentModal} = this.props;

        let renderedModal;

        if(currentModal) {
            const {modalType, modalProps = {}} = currentModal;
            const ModalComponent = modalComponentLookupTable[modalType];

            renderedModal = <ModalComponent {...modalProps} />;
        }

        return <span>{renderedModal}</span>
    }
}

export default connect(mapState)(ModalManager);
```

Now we get to the heart of the basic modal setup. Let's take a step back first, though, and consider the theory behind this.

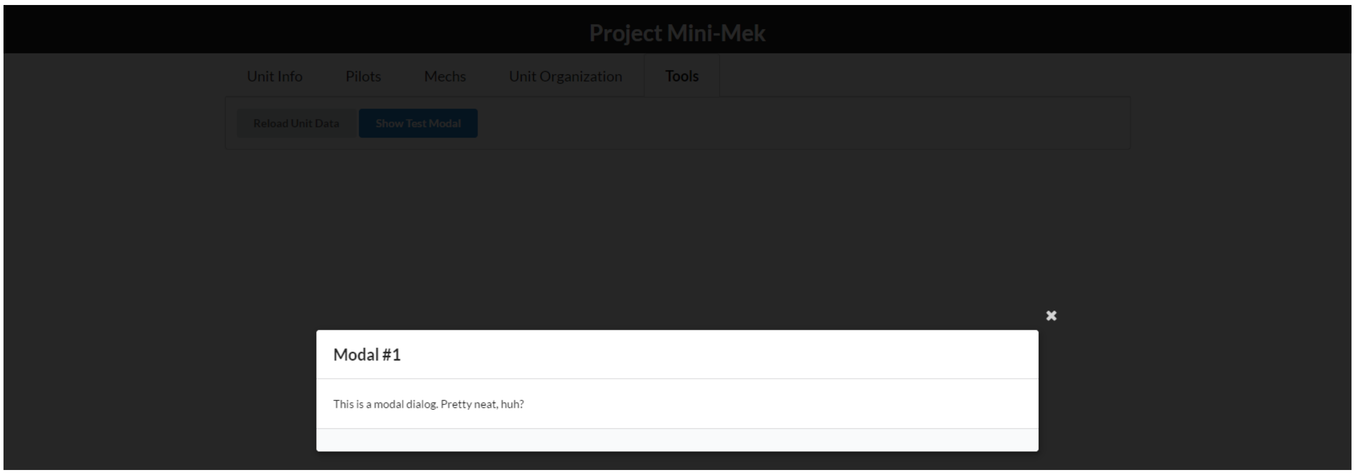
In a typical object-oriented GUI toolkit, you might do something like `const myModal = new MyModal(); myModal.show()`, and the modal class would be responsible for displaying itself somehow. This is also true with things like jQuery-based UI plugins as well. At that point, you have some "implicit state" in your application. Your app is "showing a modal", but there's no real way to track that this is happening beyond the fact that there's an instance of `MyModal` that's been created, or - if this is a web app - the fact that there's extra elements appended to the page. The "are we showing a modal?" state is implicitly `true`, but not actually being tracked anywhere.

With React (and even more so with Redux), you are encouraged to make as much of your state explicit as possible. That's why you frequently see apps that store a value like `requesting : true`, because you can now specifically base UI behavior on the fact that there's an AJAX request in progress (like showing a loading spinner). **In our case, we want to explicitly track the fact that we're showing a modal, and even more than that, a description of what the current modal is.**

Our `<ModalManager>` component will live near the top of the app's component tree. It reads the current modal description from Redux. If there's a description object, we need to actually show a modal.

We create a lookup table where the keys are some string identifier, and the values are modal components. In this case, we're just using the variable name as the key (via ES6 object literal shorthand syntax), so if `modalType` is `"TestModal"`, that's the component class we'll retrieve. Once we have the right modal component class in a variable, we can use normal JSX syntax to render that component. We can also take whatever props object was included in the description, and pass that to the modal.

Here's what the result looks like:



Modal #1

This is a modal dialog. Pretty neat, huh?

A modal! With a backdrop! Isn't this exciting? :)

Code-wise, that's really all there is to this idea. Use some descriptive value to decide when you should show a modal, and if so, which one, then render it and pass in whatever props it's supposed to have. But, we can expand on this idea in a lot of useful ways.