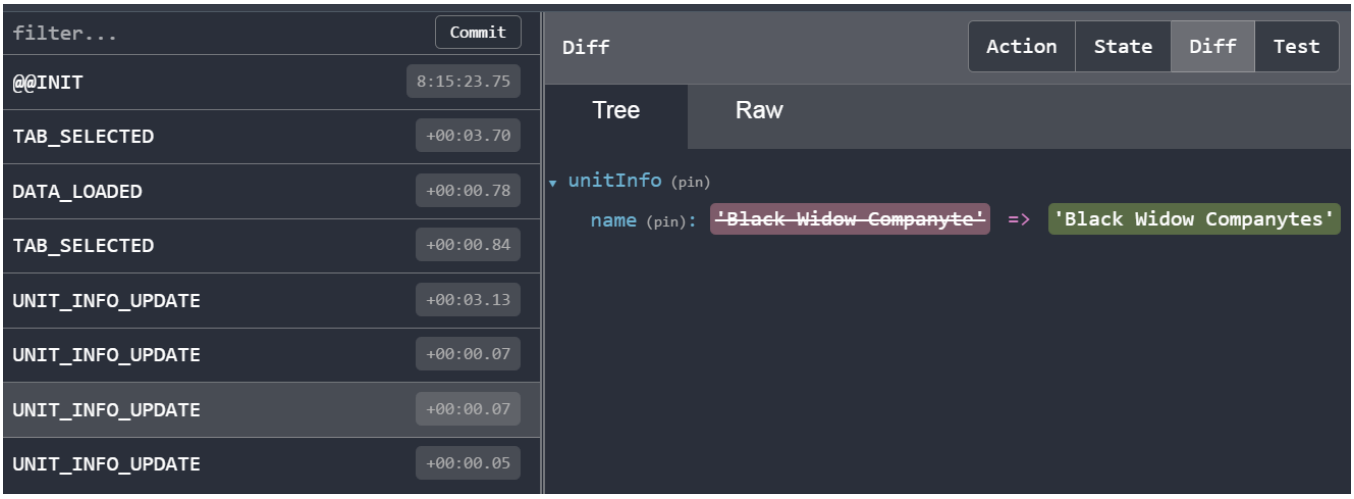
Buffering Form Changes and Dispatches

Investigating Form Actions

With that cleanup done, back to work on our input forms. In the previous section, I mentioned that **“there’s one problem with our text input that we need to address”**, and promised we’d come back to that later. Later is now :)

Let’s type the letters “test” into the end of the “Unit Name” field, and look at the DevTools actions log:



We can see four different `UNIT_INFO_UPDATE` actions in that list from typing “test” into the name input. If we look at the next-to-last action, we can see that after typing the ‘s’, the dispatched name was “Black Widow Companytes”.

Right now, **every single time we type a keystroke into the name input, we dispatch another Redux action**. Every time we dispatch an action, *every* connected component gets notified, and has its `mapState` function run. As an example, that means that for every keystroke, all the `<PilotsListRow>` components are busy looking up their Pilot entries. But... all we really wanted to do was update the name input! None of the Pilots data is going to change from us typing here, so all that is wasted effort. The dispatched actions are also kind of cluttering up the actions log.

As mentioned previously, ***Project Mini-Mek*** really isn't a performance-critical application, and especially not at this stage of development. Still, **it would be nice if we could cut down on the number of actual Redux actions dispatched**.

A Reusable Form Buffering Solution

I noted some of the issues with dispatching actions from form inputs while working on early prototypes for my own application. I needed a solution that could do several things for me:

- Handle `onChange` events from input components
- Respond to those change events by immediately passing the updated values back down to the input, for fast UI response
- If several changes happened close together, only dispatch a single Redux action after they were all done, with the final values
- Be generic enough that I could reuse it in a variety of different places.

I started playing around with some ideas, and eventually came up with a component I dubbed the "FormEditWrapper". (As you may have noticed, nothing is more permanent in programming than a temporary name. We'll call it `FEW` for short.)

My `FormEditWrapper` is a reusable, generic component that can buffer input changes in its own component state, combine those changes with incoming props, and debounce changes to minimize the number of dispatched Redux actions.. Let's look at the source, and I'll explain how it works.

> **Commit eb38ac3: Add FormEditWrapper component**

common/components/FormEditWrapper.jsx

```
import React, {Component, PropTypes} from "react";
import {noop, debounce, defaults, values} from "lodash";

import {getValueFromEvent} from "common/utils/clientUtils";

class FormEditWrapper extends Component {
```

```
    static propTypes = {
        value : PropTypes.object.isRequired,

        isEditing : PropTypes.bool,
        onChange : PropTypes.func,
        valuePropName : PropTypes.string,
        onChangePropName : PropTypes.string,
        singleValue : PropTypes.bool,
        passIsEditing : PropTypes.bool,
        dispatchDelay : PropTypes.number,
    }

    static defaultProps = {
        isEditing : true,
        onChange : noop,
        valuePropName : "value",
        onChangePropName : "onChange",
        singleValue : false,
        passIsEditing : true,
        dispatchDelay : 250,
    }

    constructor(props) {
        super(props);
        const boundDispatchAttributes = this.dispatchAttributeChange.bind
(this);
        this.dispatchAttributeChange = debounce(boundDispatchAttributes, p
rops.dispatchDelay);

        this.state = {
            value : {},
        };
    }

    componentWillReceiveProps() {
        // Reset any component-local changes  Note that the incoming props
        // SHOULD match the changes we just had in local state.
        this.setState({value : {}});
    }

    onChildChange = (e) => {
        const {isEditing} = this.props;

        if(isEditing) {
            const newValues = getValueFromEvent(e);

            if(newValues) {
```

```
                const change = {
                    ...this.state.value,

                    ...newValues
                };

                // Update our component-local state with these changes, s
o that the child components
                // will re-render with the new values right away
                this.setState({value : change});

                // Because this is debounced, we will only call the passed
-in props.onChange
                // once there is a pause in changes (like letting go o
f a held-down key)
                this.dispatchAttributeChange(change);
            }
        }
    }

    dispatchAttributeChange(change) {
        this.props.onChange(change);
    }

    render() {
        const {value : propsValue, children} = this.props;
        const {isEditing, passIsEditing, valuePropName, onChangePropNam
e, singleValue} = this.props;
        const {value : stateValue = {}} = this.state;

        // Use incoming values from props IF there is no corresponding val
ue
        // in local component state.  This allows local changes to win ou
t.
        const currentValues = defaults({}, stateValue, propsValue);

        let valueToPassDown = currentValues;

        if(singleValue) {
            valueToPassDown = values(currentValues)[0];
        }

        const editingValue = passIsEditing ? {isEditing} : {};

        // Force the child form to re-render itself with these values
        const child = React.Children.only(children);
```

```
        const updatedChild = React.cloneElement(child, {
            [valuePropName] : valueToPassDown,

            [onChangePropName] : this.onChildChange,
            ...editingValue
        });

        return updatedChild;
    }
}

export default FormEditWrapper;
```

There's a lot of stuff going on in there. Let's break it down:

Passing Down Props to Children

First, `FEW` expects a prop named `value`, which should be an object. It also uses the `React.Children` API to ensure that it only has a single child. The most basic usage would look like:

```
<FormEditWrapper value={someObject}>
    <SomeChildComponent />
</FormEditWrapper>
```

`FEW` also uses another of React's top-level APIs: `React.cloneElement()`. Since the output of a React render function is just an object, and React exposes the render output of child components as `props.children`, **it's possible for a component to return different children than what it was given**. In this case, `FEW` is going to make a copy of the child render output it was given, and pass down a couple new or different props. One prop is the data value object, and the other prop is an `onChange` callback. So, in that minimal example, it's as if we had written `SomeChildComponent value={someObject} onChange={someOnChangeCallback} />`.

The second thing to note is that `FEW` supplies its own `onChildChanged` method as the `onChange` prop for its child. Whenever `onChildChanged` is called, `FEW` will update its internal state with the values passed up by the child, merging them with whatever values are already buffered in the state. Calling `this.setState()` always forces a re-render, and that leads into the next really interesting part.

Let's use the `<PilotDetails>` form as an example. That form has two main text

Let's use the `<PilotDetails>` form as an example. That form has two main text inputs (name and age). Assume we have this setup:

```
const name = "Original Name";
const age = "42";

<FormEditWrapper value={ {name, age} }>
    <PilotDetails />
</FormEditWrapper>
```

If we type "newname" at the end of the `name` field, `FEW`'s internal state would look like `{ value : { name : "Original Namenewname"}}`. Meanwhile it's still being given the original `value={ {name, age} }` prop. We need to combine these two so that the name value we just typed overrides the name value coming in as a prop. The key is this line here:

```
// Use incoming values from props IF there is no corresponding value
// in local component state.  This allows local changes to win out.
const currentValues = defaults({}, stateValue, propsValue);
```

If we were to inspect `currentValues` after that line, we would see `{name : "Original Namenewname", age : 42}`. Since we didn't have an `age` prop in `FEW`'s state, we copy that over. Since we *do* have a `name` field in state, we *don't* copy over the `name` field we were given as a prop. This means that any values from the child component will override the equivalent values from props, and the resulting object is passed back down to the child component, allowing it to re-render itself with the correct data.

Finally, when `FEW` receives new props, it clears out its internal state buffer. The assumption is that the new props will now be a "saved" version of the changes it's keeping inside, so it can clear those out and go back to just passing down the data it's getting as a prop.

Flexibility and Customization

Perhaps the child input or form component is different than most others. Maybe it needs its value as a prop named `data` (or in the case of `<PilotDetails>`, as a prop named `pilot`). Maybe it needs the callback prop to be named something other than `onChange`. `FEW` accepts some props that tell it to change its default behavior: `valuePropName` and `onChangePropName`. Going back to our earlier example, we could do:

```
<FormEditWrapper value={ {name, age} }  valuePropName="pilot">
    <PilotDetails />
</FormEditWrapper>
```

And now `<PilotDetails>` will be given `props.pilot` instead of `props.value`.

Another customization issue is that actual `<input>` components only want a specific value like a string, instead of an object. In that case, passing the `singleValue={true}` flag to `FEW` will tell it to assume there's only one field it needs to pass down, and to pass that down directly instead of the entire value object.

In my own application, a number of components also need to be given an `isEditing` flag. Originally I passed that through all the time, but when I added the `singleValue` option, I realized that inputs wouldn't need that prop, so I added a `passIsEditing` prop to control whether that gets passed down or not.

Buffering Dispatches

`FEW`'s `onChildChanged` method will immediately call `this.setState()` with the new values from the child, and that queues up a re-render. That method also calls `this.dispatchAttributeChanged()`, which in turn calls `this.props.onChange()`. However, in the constructor, we create a debounced version of `dispatchAttributeChanged`, which will only actually run if enough time has elapsed since the last time it was called (by default, 250ms). That means that if the user quickly types several characters into a text input, `dispatchAttributeChanged` will be called repeatedly, but won't actually run for real until after the user is done typing. As a result, when it *does* run, it will call `this.props.onChange()` with the final value. So, instead of seeing `"n"`, `"ne"`, `"new"`, and so on, you'd see a single `"newname"` value passed back up. The `onChange` prop doesn't have to be a Redux action creator, but it usually will be. That means that **most of the time, only a single Redux action will be dispatched for a series of keystrokes**.

`FormEditWrapper` in Action

With all that in mind, let's use `FEW` to buffer the "name" input in our `<UnitInfo>` component.

> **Commit b33f6d5: Use FormEditWrapper with UnitInfo name input**

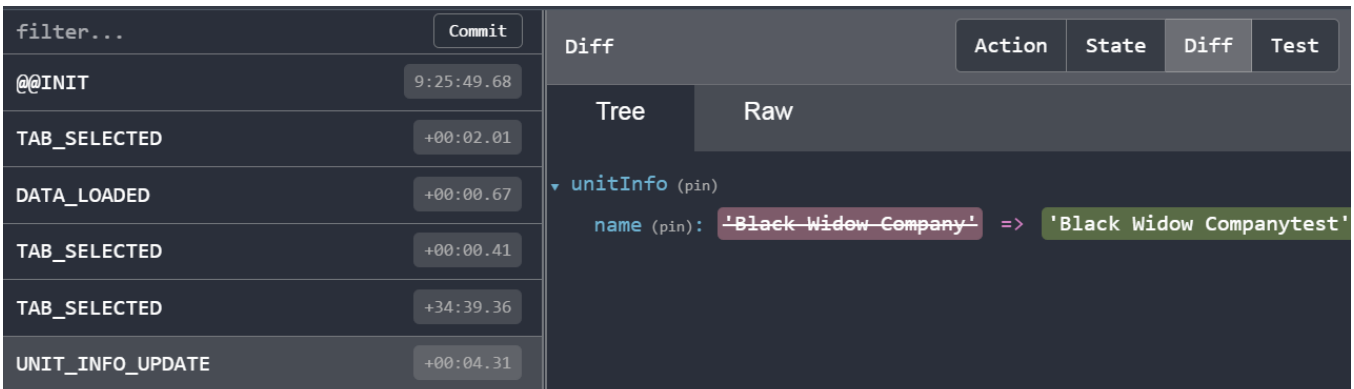
features/unitInfo/UnitInfo.jsx

```
+import FormEditWrapper from "common/components/FormEditWrapper";

// Omit irrelevant code
    render() {
-       const {unitInfo} = this.props;
+       const {unitInfo, updateUnitInfo} = this.props;
        const {name, affiliation} = unitInfo;

// Omit rendering code
                    <Form.Field name="name" width={6}>
                        <label>Unit Name</label>
+                       <FormEditWrapper
+                           singleValue={true}
+                           value={ {name} }
+                           onChange={updateUnitInfo}
+                           passIsEditing={false}
+                       >
                            <input
                                placeholder="Name"
                                name="name"
-                           value={name}
-                           onChange={this.onNameChanged}
                            />
+                       </FormEditWrapper>
                    <Form.Field name="affiliation" width={6}>
```

So now, if we type “test” at the end of the name input and look in the DevTools, we should see this:



Success! Four keystrokes, the input updated immediately as we typed, and only a single Redux action was dispatched.