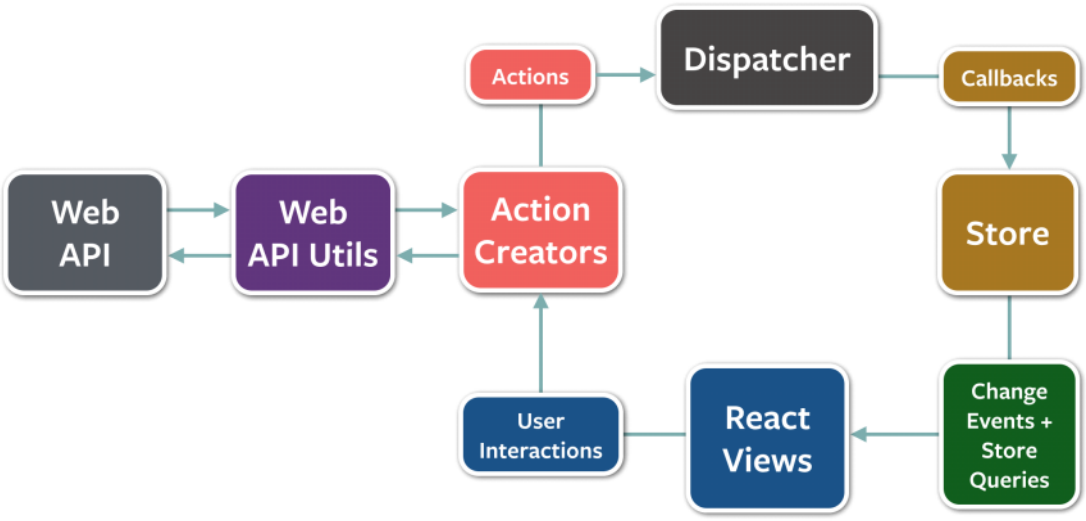
Redux

Friday, August 28, 2015 1:59 PM

Flux diagram (Redux is based on the principles of Flux):



Actions should always contains the 'type' property (~~although this is a convention, not a requirement~~ **this is now a requirement in Redux 3.0**). It can also have any other arguments/information.

Many of the notes below came from this tutorial here. The official docs are also great.

An ActionCreator is a pure function (no side effects) that creates an Action.

```
var actionCreator = function() {
   return {type: 'INCREMENT', amount: 5}
};
```

There's an explanation here about why you would want ActionCreators over just plain objects.

Note that this does not actually *dispatch* the action (obviously), it just creates it. At some point, store.dispatch(someAction) needs to be called, which will fire your reducers to produce a new state.

All data in your application is stored in a state by Redux (Redux is advertised as a "predictable state container").

You handle data modification with reducers. Reducers are just subscribers to actions; they get the current state and the action, and they return a new, "reduced" state.

To propagate modifications to all parts of your application, you use subscribers to the state's modifications.

So Redux is really just:

- A container for your state
- A mechanism to subscribe to state updates
- A mechanism to publish (or "dispatch") actions to reducers, which in turn produce a new state

The three core principles of Redux are:

- Single source of truth: the state of your application is stored in an object tree inside a single store.
    - This allows the state from the server to be serialized and hydrated into the client with no extra coding effort.
    - Also, on this note: don't duplicate data in Redux that can be computed. For example, saving users and then saving them again sorted by ID. Just sort them in the UI when there's a change in Redux.
- State is read-only: the only way to mutate the state is to emit an action
    - This ensures that the views or the network callbacks never write directly to the state, rather they express the intent to mutate. Because all mutations are centralized and happen in a strict order, there are no race conditions.
- Mutations are written as pure functions
    - Reducers are *pure* (as are ActionCreators, since actions themselves are just plain objects), meaning there can be no side-effects. If there were side effects, then you may violate the "state is read-only" principle.

You create an instance of Redux by doing

var store = createStore(reducer); // reducer is a function that takes in a state and an action and produces a new state

Note that the diagram above shows "Store". That's because in Flux, a store holds your data. In Redux, a reducer is provided any data it needs and reduces it to a new state. Thus, in Redux, reducers are stateless.

Also, they're called "reducers" because it's the type of function you would pass to Array.prototype.reduce(reducer, ?initialValue).

Reducers should be pure, so you should never mutate its arguments or perform side effects like API calls or routing transitions.

You can always get the current state of a Redux store by calling "getState()".

Your store starts with an initial state of "undefined", so thanks to ES6's default parameters, your basic reducer will look like this:

```
var reducer = function(state = {}, action) {
    return state;
};
var store = createStore(reducer);
store.getState(); // it is now {}
```

Most reducers will be structured with switch/case statements. Don't forget your default statement of returning the state!

To combine reducers, just import and use "combineReducers" from redux:

```
import {createStore, combineReducers} from 'redux'
var userReducer = function(state = {}, action) {
    // switch/case
};
var itemsReducer = function(state = [], action) {
    // switch/case
};
var combinedReducer = combineReducers({
    user: userReducer,
    items: itemsReducer
});
```

Now if you call createStore(combinedReducer), all reducers will be dispatched to.

The important thing to note here is that each reducer only needs to know about a slice of the application state, giving us a state of this when we create our store:

```
{
    users: {}, // {} is the slice returned by userReducer
    items: [] // [] is the slice returned by our itemsReducer
}
```

We should get in the habit of dispatching using an ActionCreator:

```
var chatMessageActionCreator = function(name, msg) {
    return {
        type: 'CHAT_MESSAGE',
        name, // ES6 shorthand for "name: name"
        msg
    };
};
store.dispatch(chatMessageActionCreator('Adam', 'hello')); // I'm just invoking the function and passing the result to dispatch().

// The above demonstrates ActionCreator --> Action --> dispatcher --> reducer
```

The above dispatch was synchronous. If we want to asynchronously dispatch, we need to write an ActionCreator that returns a function. HOWEVER, Actions themselves MUST be plain objects, which is why we'll need middleware.

```
var asyncChatMessageActionCreator = function(name, msg) {
    return function(dispatch) {
        setTimeout(function() {
            dispatch({
                type: 'CHAT_MESSAGE',
                name, // ES6 shorthand for "name: name"
                msg
            });
        }, 2000); // dispatches after 2 seconds
    };
};

// Remember: you can't just dispatch that above without reading about middleware below.
```

Just like in Node, any number of middleware can be added:

action --> dispatcher --> middleware 1 --> middleware 2 --> … --> middleware N --> reducers

Middleware has this signature:

```
var anyMiddleware = function ({ dispatch, getState }) {
    return function(next) {
        return function (action) {
            // your middleware-specific code goes here
        }
    }
}
```

You're given a 'getState' *function* instead of just a *state* because this is asynchronous, so you need

to be able to *get* the state when you want it.

There's a "function(next)" so that we can call into the next middleware when we're done.

Finally, the innermost function lets us return a value for use in either the next middleware or finally the reducer.

Wanting your ActionCreator to return a function-wrapped expression (a "thunk") is so common that gaeron wrote redux-thunk, a 6-line piece of middleware:

```
export default function thunkMiddleware({ dispatch, getState }) {
  return next => action =>
    typeof action === 'function' ?
      action(dispatch, getState) :
      next(action);
}
```

All this really does is:

- Follows the middleware signature
- If your ActionCreator returns a function, this will call it. Otherwise, it will pass your action (which we now know is a plain JS object) to the next piece of middleware.

To use this:

- npm install --save redux-thunk
- var thunkMiddleware = require('redux-thunk');
- import {createStore, combineReducers, applyMiddleware } from 'redux';
- const finalCreateStore = applyMiddleware(thunkMiddleware)(createStore);
- const store_0 = finalCreateStore(/*your combined reducers*/)

There's probably tons of middleware online, but here's one possibly-useful example of logging middleware

```
function logMiddleware ({ dispatch, getState }) {
  return function(next) {
    return function (action) {
      console.log('logMiddleware action received:', action)
      return next(action)
    }
  }
}
```

All of the above is a cool way to manage state with no way to react to it (no UI, no events, etc.). That's where "subscribe" comes in. It's as simple as this:

```
store.subscribe(function() {
    console.log('store has been updated. Latest state: ', store.getState());
});
```

Just note that explicitly calling subscribe() is not really how you end up *changing state* in Redux. Instead, reducers are where the magic happens. You don't subscribe to 'increment', instead you write a reducer which listens for 'increment' to be dispatched and then produces a new state when it does. However, when you want your application to do something with the newly incremented value (e.g. send it to a server), that's where you would use "subscribe".

9/13/15 - I thought about this some more, and I think I originally thought redux should handle EVERYTHING. I think redux is just for your UI state that is spread across multiple components. So,

for example, I had a ChatInput (an <input> and a "Send" <button>), and I wanted to make it so that when you pressed the "Send" button, it put your chat message into the log and then sent it to the server. So I thought I would dispatch an "addChatMessage" action and then respond to its completion. However, I think the correct way to do it is that the "Send" button has an "onSend" handler which does both of those things.

```
onChatWrapper(text) {
  const actions = bindActionCreators(tempActions, this.props.dispatch); // this should probably be done in the constructor though
  actions.addChatMessage(text);
  this.props.onChat(text);
}
```

Then, the "onChat" that you pass in can be:

```
onChatMessageSent(text) {
  this.socket.emit(constants.PLAYER_CHAT_MESSAGE, text);
}
```

The only problem with this is you can't react to state changes made through the UI, but my discovery (which may not be right) is that you shouldn't be doing that anyway. Only UI should react to UI, and that's the entire point of React. If you want to do something when a chat message happens, then you need to go a layer above where you may think.

It may be tempting to use ES6 symbols to represent action types, but that only makes it harder to record/replay them, so just stick to const strings.

6/18/2016
You shouldn't store classes or functions with prototypes in Redux according to this post.

React-redux

React-redux is a library that will expose your redux store to React components in such a way that it looks like their props. If you ever wanted something like this without having to use React, then the paradigm would just be sharing the redux store directly with whatever classes should have access to it.

If you call a bound action from the constructor, it will correctly update the store immediately, but it won't update your react-redux'd state until after the component has mounted. Reference.

Getting data from a "connect()"ed child

Normally, you can do this in React:

```
class TheChild extends React.Component {
  myFunc() {
    return 'hello';
  }
}
class TheParent extends React.Component {
  render() {
    return ( < TheChild ref = 'foo' / > );
  }

  componentDidMount() {
    var x = this.refs.foo.myFunc();
    // x is now 'hello'
  }
```

```
}
```

However, if you try doing that to a component that you called "connect" on, then it will be wrapped, and you won't be able to get a reference to the component unless you do two things:

1. Export your connected component like this:
   export default connect(mapStateToProps, undefined, undefined, {withRef:true})
   (BlocklyComponent);
2. Refer to the component like **this**:
   const wrappedBlocklyComponent = scriptContent.getBlockly();

   const blocklyComponent = wrappedBlocklyComponent.**getWrappedInstance**();

Redux Devtools (reference)

This is a great plug-in for Chrome that lets you inspect your Redux state from any page.

Here's how I configure my store to use this:

```
import { createStore, applyMiddleware, compose } from 'redux';
import thunk from 'redux-thunk';
import rootReducer from '../reducers';

export default function configureStore(initialState) {
    const connectToReduxDevTools = (process.env.NODE_ENV !==
'production'
        && typeof window === 'object'
        && window.__REDUX_DEVTOOLS_EXTENSION_COMPOSE__
    );

    const composeEnhancers = connectToReduxDevTools
        ? window.__REDUX_DEVTOOLS_EXTENSION_COMPOSE__
        : compose;

    const enhancer = composeEnhancers(
        applyMiddleware(thunk)
        // other store enhancers if any
    );

    const store = createStore(rootReducer, initialState, enhancer);

    return store;
}
```

Note: if you want a remote version of this for checking state of Redux on your server, use this.