32 Redux: Combining Reducers

Objectives

- Students will learn to use redux with react
- Students will learn to design redux reducer's for controlling application state
- Students will learn to design action creator functions for working with redux

Recap: Store, Reducer, Actions

- All application state is kept in the **store**
- The store receives messages with **actions**

- The store uses a **reducer** to find out how it should manipulate it's state according to the `.type` of the action.

- Components use `mapStateToProps` to pick data out of the state in the store and attach it as if it were passed as props.

- Components use `mapDispatchToProps` to create props that are functions that can send messages with actions back to the store via `dispatch()`

Recap: Reducer

- A **reducer** is a function that accepts a state and an action and returns a new state
- `(state, action) -> newState`
- Import constants from an actions file to see what types of actions are supported.

```
import {ADD} from '../actions/math-actions';
const initialState = {data: 0};

export default function counterReducer(state, action) {
  if (state === undefined) {
    return initialState;
  }

  let newState = {};
  switch(action.type) {
    case ADD:
      return Object.assign(newState, state, {data: state.data + 1});
    default: return state;
  }
```

Recap: Store

- Import `createStore` from `redux` and pass in a reducer.
- Import the `<Provider>` component, give it the `store` as a prop and wrap your entire `<App>` with the `<Provider>`.

```
import {createStore} from 'redux';
import {Provider} from 'react-redux'

import myReducer from '../reducers/my-reducer';
const store = createStore(myReducer);

render() {
  <Provider store={store}>
    <App />
  </Provider>
}
```

Recap: Actions

- Actions carry messages to the `store`
- Actions are plain-old-javascript-objects ("POJOs") with a `.type` property,
- Actions may optionally have other properties with more data (like the contents of a new TODO item)
- Define action types as string contants.
- Include functions called **action creators** that bundle together action data in an object with the `.type` specified.

counter-actions.js

```
export const DELETE_ALL = 'DELETE_ALL';
export const ADD_TODO = 'ADD_TODO';

export function addTodo(text) {
  return {type: ADD_TODO, text: todo}
}

export function deleteAll() {
  return {type: DELETE_ALL}
}
```

Combining Reducers

- As an application becomes more complex you will create multiple reducers.
- Redux supports automatically combining multiple reducers into one reducer with the `combineReducers` function.

reducers/index.js

```
import {combineReducers} from 'redux';
import counterAppReducer from './counter-app';
import timestampReducer from './timestamp';

export default combineReducers({
  counter: counterAppReducer,
  timestamps: timestampReducer
})
```

Accessing Combined Reducers

- Access the state associated with each reducer according to what property name you associated it with in `combineReducers`.
- Here the component uses `mapStateToProps` and accesses the timestamps in the timestamps reducer with `state.timestamps.timestamps`

reducers/index.js

```
export default combineReducers({
  counter: counterAppReducer,
  timestamps: timestampReducer
})
```

components/some-component.js

```
render() {
    {this.props.timestamps.map(timestamp -> <div>{timestamp.toString}</div>)};
}


const mapStateToProps = state => {
  return {timestamps: state.timestamps.timestamps};
}
```

(reference) Timestamp Reducer

```
import {STAMP} from '../actions/timestamp-actions';

const initialState = {
  timestamps: []
};

export default function counterReducer(state, action) {
  if (state === undefined) {
    return initialState;
  }

  let newState = {};
  switch(action.type) {
    case STAMP:
      return Object.assign(newState, state, {
        timestamps: [...state.timestamps, new Date()]
      });
    default: return state;
  }
}
```

Summary

- Complex applications require multiple reducers
- Multiple reducers allow us to cordon uncommon code to multiple files
- Having multiple reducers across multiple files makes it easier for us to reason about our applications, and makes collaborating with source control easier. (less merge conflicts!)

Practical multiple-reducer examples (if you are facebook):

- A reducer for creating a new post
- A reducer for flipping through photos in an album
- A reducer for loading stories while scrolling through a timeline
- A reducer for tagging friends in a photo
- A reducer for searching for friend's profiles
- A reducer for choosing your top eight friends