Redux

What are we going to cover

What is Redux?

What are the different building blocks with Redux?

How can we use Redux with React?

How can we use Redux with React-Router?

What is Redux?

Redux is a **predictable state container** for JavaScript apps

- Combines the Flux and Elm architecture

Can be used with **any UI library**

- Not tied to React

Has a rich **ecosystem** of utilities

- Redux DevTools with optional Chrome plugin
- Many others

Redux building blocks

Store

Actions

Reducers

Middleware

Store

Holds the application **state**

- Not all state from the database, just what is needed in the user interface

Used to **dispatch** actions

- No separate dispatcher as in Flux

A Redux application normally has **one single store**

- Holds all state required by the application

User interface components select that part of state they are interested in

- For example using the react-redux connect() function

Creating a Redux store

```
import { createStore, combineReducers }
  from "redux";

const store = createStore(
  combineReducers({
    movies: moviesReducer,
    directors: directorsReducer
  })
);
```

Actions

Actions are **objects** that describe store modifications

Are used to **trigger changes** to the state in the store

- Using store.dispatch()

Actions are **simple data structures**

- The **type** property is use to determine what action it is
- The **payload** is data associated with the action
- There can be **error** information associated with an action

Actions are often the result of a user doing something

- But can also be triggered programmatically
  - setTimeout() / setInterval()
  - An AJAX call finishing

Action Creators

Redux actions are normally created using **action creators**

Action creators are **pure functions** that create action objects

- They **do not dispatch** the action created

Creating action objects

```
import * as actionTypes from "./actionTypes";

export const addMovie = movie => ({
  type: actionTypes.ADD_MOVIE,
  payload: movie
});

export const rateMovie = (movieId, rating) => ({
  type: actionTypes.RATE_MOVIE,
  payload: {
    movieId,
    rating
  }
});
```

Reducers

Reducers **transform the current state** based on an incoming action

- Old State + Action => New State

Create **different reducers** for different parts of the state

- For example a to-do list reducer and a single to-do reducer
- Combine there together

Combine different reducers into one new reducer function

- Use the Redux **combineReducers()** function

Reducers should be **pure functions**

- It is recommended not to mutate state but always create new state
- Based on functional immutable concepts
- Makes for more predictable and better testable applications

Reducing the dispatched actions

```
import * as actionTypes from "./actionTypes";
import movieReducer from "./movieReducer";

export default (state = [], action) => {
  switch (action.type) {
    case actionTypes.ADD_MOVIE:
      return [...state, action.payload];
    case actionTypes.RATE_MOVIE:
      return state.map(movie => {
        if (movie.id === action.payload.movieId) {
          return movieReducer(movie, action)
        } else {
          return movie;
        }
      });
  }
  return state;
};
```

Middleware

Middleware sits between dispatching an action and reducing the state with it

- Is optional but often used

Middleware can do all sorts of things with **side effects**

- Log actions
- Do AJAX requests
- Error reporting

Typically involves some side effect we want to keep **outside a reducer**

- Add as many middleware components as required

Adding middleware

```
import {
  createStore, combineReducers, applyMiddleware
} from "redux";
import thunk from 'redux-thunk’;

const store = createStore(
  combineReducers({
    movies: moviesReducer,
    directors: directorsReducer
  }),
  applyMiddleware(thunk)
);
```

A thunk action creator

```
export const moviesLoaded = movies => ({
  type: actionTypes.MOVIES_LOADED,
  payload: movies
});

export const loadMovies = () => dispatch => {
  fetch("/api/movies")
    .then(rsp => rsp.json())
    .then(movies =>
      dispatch(moviesLoaded(movies)))
};
```

Redux with React

Redux is a stand alone library

- But very often combined with React
- These days the defacto standard Flux implementation

React-Redux bindings in a separate NPM package

<Provider store={store}/>

The **Provider** component makes the store available to the React components

- Passes the store using the React context

Normally use as the **root component** passed to ReactDOM.render()

The Provider component takes **one child**

- The React application you developed
- Can be a React-Router component

Providing the store

```
class AppContainer extends Component {
  render() {
    return (
      <Provider store={store}>
          <App />
      </Provider>
    );
  }
}
```

connect()

The **connect()** function connects a React component to the store

- The component should only work with props

Takes 4 optional **parameters**

- **mapStateToProps** function
- **mapDispatchToProps** function or object
- mergeProps function
- Options object

connect() parameters

The **mapStateToProps** function is almost always used

- Function that **extracts data from the store** and provides it to the component as props
- The stores current state is passed as a parameter

The **mapDispatchToProps** is also used very often

- Used to create functions that **dispatch** actions to the store and provided as props
- The **dispatch** function is passed as a parameter
- Has a **convenience** form when an object is passed

connect() parameters

The **mergeProps** function is rarely used

- Determines how mapStateToProps(), mapDispatchToProps(), and parent props are merged
- Defaults to a **sensible** Object.assign({}, ownProps, stateProps, dispatchProps)

The **options** object is also rarely used

- Object with some customizations
  - ***pure*** to create a component with pure behavior
    - Defaults to true
  - ***withRef*** Makes the wrapped component available via the getWrappedInstance() function
    - Defaults to false
  - Various **functions to compare state and props** when pure = true

Connecting a component to the store

```
class MoviesList extends Component {
  render() { // Todo... }
}

const mapStateToProps = state => ({
  movies: state.movies
});

const mapDispatchToProps = dispatch => ({
  loadMovies: () => dispatch(loadMovies())
});

export default connect(
    mapStateToProps,
    mapDispatchToProps
)(MoviesList);
```

Redux with React-Router

React-Redux and **React-Router** will work together just fine

- But route information is not part of the Redux state

The **React-Router-Redux** is often used to keep then in **sync**

- Useful with time travel debugging utilities like Redux DevTools

Immutability with Immer

Work with **immutable state** in a more convenient way

- Based on the **copy-on-write** mechanism.

Write code that just modifies the **draft** object

- Immer will track all modifications and return a new copy with the changes
- Any unchanged objects will still be the original object

A reducer using Immer

```
import produce from 'immer’;

export default (state = [], action) =>
  produce(state, draft => {
    switch (action.type) {
      case 'ADD_MOVIE':
        draft.push(action.payload);
        break;
      case 'RATE_MOVIE':
        const movie = draft.find(
            m => m.id === action.payload.movieId);
        movieReducer(movie, action);
        break;
      default:
    }
  });
```

Conclusion

Redux is the most popular Flux implementation

- Inspired by the Elm architecture

It isn’t tied to React but often used in combination

Has a nice ecosystem of middleware and other utilities