

# Redux





# Redux Overview

- “Predictable state container for JavaScript apps”, not just for use with React
- Created by Dan Abramov
  - works for Facebook on React Native team as of late 2015
- Name is a contraction of “reducers” and “Flux”
- Resources
  - main website - <http://redux.js.org>
  - great, free video series from Dan Abramov - <https://egghead.io/series/getting-started-with-redux>
  - Dan Abramov talk - “Live React: Hot Reloading with Time Travel” from react-europe 2015 <https://www.youtube.com/watch?v=xsSnOQynTHs>



# Variation on Flux

- **Flux**

- “application architecture for client-side web applications” designed by Facebook
- supports using multiple “stores”; Redux only uses one
- uses a “dispatcher” to send “actions” to the stores
  - in Redux, the single store has a **dispatch** method
- main benefit is that it organizes code in a way that is easy for others to understand and follow the flow
- see <https://facebook.github.io/flux/>

**Flux architecture**

component -> event -> action ->  
dispatcher -> stores -> components

- Also incorporates ideas from **Elm**

- functional language inspired by Haskell that is compiled to JavaScript
- focused on building web UIs
- Elm “updaters” were the inspiration for Redux reducer functions
- see <http://elm-lang.org>



# Three Principles

- Represents entire app state in a **single** JS object (**store**)
  - single source of truth
  - typically a deeply nested object
  - includes data and information about the UI state (ex. current sort order, filtering, ...)

idea borrowed from  
**ClosureScript** library  
**Om** by David Nolan
- State can only be changed by **dispatching an action** to the store
  - unidirectional data flow
  - never directly modified from view
  - avoids race conditions where multiple code paths are taking turns modifying parts of the state

Only components that are unique to the app should dispatch events because doing that requires knowledge of the app state management and makes them non-reusable.
- The store uses **reducers** to derive a new state from the old state
  - functions that take current state and an action and return new state
  - composable - reducer functions can call other reducer functions that handle specific parts of the state tree



# Event Handling Functions

- Send zero or more HTTP requests (Ajax)
  - typically to invoke REST services
- Dispatch one or more actions to the store
  - triggers reducer functions that create new state
- Without Redux, event handling is typically performed by component methods that modify state held by the component
- With Redux, state is held in the store rather than in components, so event handling can be performed by plain functions
- Eliminates need to use the **bind** method on them since they don't call **this.setState(new-state)**
- We'll see this in the "gift" and "gift-redux" apps later



# Actions

- Objects that contain **type** and **payload** properties
  - **type** values are typically constants with string values, because those are serializable (unlike ES6 symbols)
  - **payload** can be an any kind of value
- Sent to store with **store.dispatch(action)**
  - see react-redux **connect** and **bindActionCreators** functions feels too magical for another way to dispatch actions
- For a Todo app, examples of actions include
  - add todo, mark todo completed, delete todo, archive completed todos, filter todos
- Can record a session by saving all actions
- Can replay a session by replaying saved actions



# Action Standard

- Recommended standard for action objects
  - <https://github.com/acdlite/flux-standard-action>
- An action object should only have these properties
  - **type** - describes the kind of action that has occurred; required
    - typically a string constant
  - **payload** - primary way of passing data; optional
    - any kind of value
    - often an object containing multiple properties
    - if **error** is **true**, this should be an **Error** object
  - **error** - indicates whether this action describes an error that has occurred; optional
    - boolean
  - **meta** - “extra information that is not part of the payload”; optional
    - any kind of value



# Action Creators

- Pure functions that create and return action objects
- Not needed for simple actions
- “Bound action creators” create an action object, dispatch it to the store, and return nothing
  - an optional approach



# Reducers ...

- Pure functions that take current state and an action, and return new state
  - do not modify arguments
  - do not call functions that have side effects (ex. Ajax and route changes)
  - can use any of the immutability options discuss earlier to simplify creating a new state object from the current one
- Name comes from **Array reduce** method that takes a function with the same signature
  - `(accumulator, value) => accumulator`
- Easy to write tests for reducer functions
  - since they are pure

**Ajax calls** should be made by event handling functions or functions called by those; can pass results to Redux via actions so store can be updated



# ... Reducers

- Reducers often contain a **switch** statement that switches on **action.type** and returns the current state object (from **switch default**) if they don't handle the action type passed to them
  - an alternative to switch statement is **createReducer** function
    - described at bottom of <http://redux.js.org/docs/recipes/ReducingBoilerplate.html>
- By convention, if current state is **undefined** then return initial state



# Reducer Composition

- Top-level reducer (“root reducer”) can call other reducers to evaluate specific parts of the state tree and generate new state for that part

- each needs a switch statement that evaluates `action.type` and acts on those it understands, returning current state otherwise

more than one reducer can make changes for a given action type

- This is a common pattern that is simplified by `combineReducers` rather than hand-writing the top-level reducer

```
import {combineReducers} from 'redux';
const reducer = combineReducers({
  stateKey1: reducerForStateKey1,
  ...
});
```

- Issues with this approach
  - functionality for a given action type can be spread over many reducers
  - to understand what happens for a given action type, need to examine all reducers
  - the order in which the reducers are evaluated may matter and different orders may be needed based on the action type



# Better Approach

see [react-examples/redux-demo](https://react-examples.github.io/redux-demo/)

- Write action-specific reducers (no switch statement) that are called from a hand-written, top-level reducer
- Make these properties of an object
- Use these property names as action type values
- Top-level reducer calls action-specific reducer by name
- Benefits
  - no switch statements required
  - no action constants required
    - action type values are just names of action-specific reducer functions
  - all code related to handling a specific action type is in one place
- Downside
  - if structure of state tree changes, many reducer functions may require changes



# State

- Can include
  - locally created data
  - remotely created data such as from Ajax/REST responses
  - UI state such as current route, selected tabs, expanded accordions, pagination details
- Advice on shape of state tree
  - keep UI state in a different part of tree from other data
  - store other data in a normalized fashion (like a relational database)
    - separate collections of objects by type rather than nesting (ex. employees, teams, projects)
    - key by an id to support referencing between objects
- Undo/Redo
  - easily implemented by saving all state trees in a list
  - can go to previous or next UI state (if there is one) by setting current state to another state in the list and re-rendering UI
  - but this doesn't address undoing server-side changes like database updates



# Subscribing to State Changes

- View can subscribe to state changes

```
const unsubscribe = store.subscribe(() => {  
  const state = store.getState();  
  // Render a component with new state data.  
});
```

- typically done in `componentDidMount` lifecycle method of top component or in app startup code outside any component
- To unsubscribe, call function returned by `store.subscribe`
  - typically done in `componentWillUnmount` lifecycle method



# Installing

- Mandatory
  - `npm install --save redux`
  - some module bundler like webpack, Rollup, or jspm
- Optional
  - `npm install --save react-redux`
    - official React bindings for Redux
    - “makes the Redux store available to `connect` calls in the component hierarchy”
    - perhaps overkill
  - `npm install --save-dev redux-devtools`
    - “a live-editing time travel environment for Redux”



# Simple Example ...

redux-simple

```
import React from 'react';
import ReactDOM from 'react-dom';
import { createStore } from 'redux';

const Counter = ({number, onIncrement}) =>
  <div>
    {number} <button onClick={onIncrement}>+</button>
  </div>;

const {func, number} = React.PropTypes;
Counter.propTypes = {
  number: number.isRequired,
  onIncrement: func.isRequired
};

function increment() {
  store.dispatch({type: 'increment'});
}
```

If this function made an asynchronous call,  
it might dispatch four actions:

- 1) indicate asynchronous call in progress
- 2) success (promise resolves)
- 3) error (promise rejects)
- 4) indicate asynchronous call completed



# ... Simple Example

```
function render() {
  ReactDOM.render(
    <Counter
      number={store.getState().number}
      onIncrement={increment}/>,
    document.getElementById('content'));
}

function reducer(state = {number: 0}, action) {
  switch (action.type) {
    case 'increment':
      return Object.assign(
        {}, state, {number: state.number + 1});
    default:
      return state;
  }
}

const store = createStore(reducer);
store.subscribe(render);
render();
```



# Without Redux ...

redux-like

```
import React from 'react';
import ReactDOM from 'react-dom';

class Counter extends React.Component {
  constructor() {
    super();
    // Top component holds state instead of a Redux store.
    this.state = {number: 0};
  }

  render() {
    return <div>
      {this.state.number} <button onClick={increment}>+</button>
    </div>;
  }
}

const topComponent = ReactDOM.render(
  <Counter/>,
  document.getElementById('content'));
```

based on feedback from Dan Abramov at  
<https://github.com/rackt/redux/issues/1367>

still uses a single state tree  
(owned by top component)  
and reducer functions



# ... Without Redux

```
function increment() {  
  dispatch({type: 'increment'});  
}  
  
function dispatch(action) {  
  topComponent.setState(reducer(topComponent.state, action));  
}  
  
function reducer(state, action) {  
  switch (action.type) {  
    case 'increment':  
      return Object.assign(  
        {}, state, {number: state.number + 1});  
    default:  
      return state;  
  }  
}
```

call to `setState` will trigger a re-render



# Hyperlink Example

- Hyperlink that dispatches an action when clicked

```
function handleClick(e) {  
  e.preventDefault();  
  store.dispatch(createMyAction(e));  
}
```

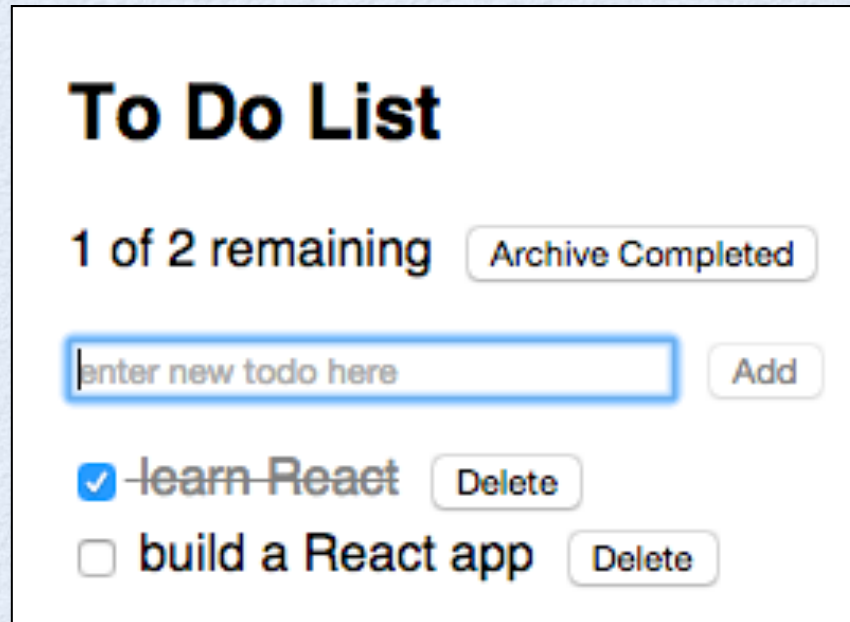
```
<a href="" onClick={handleClick}>Do Something</a>
```



# Todo App

todo-redux-rest

- This version of the Todo app uses
  - React for DOM rendering
  - Redux for state management
  - Immutable for efficient state modification
  - axios for Ajax
  - Node.js and Express for REST server
  - MongoDB for persistence



The screenshot shows a web application titled "To Do List". Below the title, it says "1 of 2 remaining". To the right of this text is a button labeled "Archive Completed". Below this is a text input field with the placeholder text "enter new todo here" and an "Add" button to its right. Below the input field, there are two list items. The first item is "learn React" with a checked checkbox to its left and a "Delete" button to its right. The second item is "build a React app" with an unchecked checkbox to its left and a "Delete" button to its right.

```
To run:  
npm install  
mongod  
npm run build  
npm run rest-server  
npm start  
browse localhost:8080
```

# index.html and todo.css

```
<!DOCTYPE html>
<html>
  <head>
    <title>React Todo App</title>
  </head>
  <body>
    <div id="content"></div>
    <script src="build/bundle.js"></script>
  </body>
</html>
```

```
body {
  font-family: sans-serif;
  padding-left: 10px;
}

button {
  margin-left: 10px;
}

li {
  margin-top: 5px;
}

ul.unstyled {
  list-style: none;
  margin-left: 0;
  padding-left: 0;
}

.done-true {
  color: gray;
  text-decoration: line-through;
}

.error {
  color: red;
  font-size: 18px;
  font-weight: bold;
}
```



# todo-header.js ...

```
import Immutable from 'immutable';
import React from 'react';

class TodoHeader extends React.Component {
  getUncompletedCount(iTodos) {
    return iTodos.reduce(
      (count, iTodo) => iTodo.get('done') ? count : count + 1,
      0);
  }

  shouldComponentUpdate(nextProps) {
    // This test is easy because iTodos is an Immutable object!
    return this.props.iTodos !== nextProps.iTodos;
  }

  render() {
    const {iTodos, onArchiveCompleted} = this.props;
    return (
      <div>
        <h2>To Do List</h2>
        <div>
          {this.getUncompletedCount(iTodos)} of {iTodos.size} remaining
          <button onClick={onArchiveCompleted}>
            Archive Completed
          </button>
        </div>
      </div>
    );
  }
} // end of TodoHeader class
```

These files use the convention that variables referring to immutable objects begin with "i".

# ... todo-header.js

```
const {func, instanceOf} = React.PropTypes;
TodoHeader.propTypes = {
  iTodos: instanceOf(Immutable.OrderedMap).isRequired,
  onArchiveCompleted: func.isRequired
};

export default TodoHeader;
```



# todo.js ...

```
import Immutable from 'immutable';
import React from 'react';

class Todo extends React.Component {
  shouldComponentUpdate(nextProps) {
    // This test is easy because iTodo is an Immutable object!
    return this.props.iTodo !== nextProps.iTodo;
  }

  render() {
    const {iTodo, onDeleteTodo, onToggleDone} = this.props;
    const done = iTodo.get('done');
    const text = iTodo.get('text');
    return (
      <li>
        <input type="checkbox"
          checked={done}
          onChange={onToggleDone}/>
        <span className={'done-' + done}>{text}</span>
        <button onClick={onDeleteTodo}>Delete</button>
      </li>
    );
  }
} // end of Todo class
```

# ... todo.js

```
const {func, instanceOf} = React.PropTypes;
Todo.propTypes = {
  iTodo: instanceOf(Immutable.Map).isRequired,
  onDeleteTodo: func.isRequired,
  onToggleDone: func.isRequired
};

export default Todo;
```



# todo-list.js ...

```
import Immutable from 'immutable';
import React from 'react';
import Todo from './todo';

class TodoList extends React.Component {
  shouldComponentUpdate(nextProps) {
    // This test is easy because iTodos is an Immutable object!
    return this.props.iTodos !== nextProps.iTodos;
  }

  render() {
    const {iTodos, onDeleteTodo, onToggleDone} = this.props;
    return (
      <ul className="unstyled">
        {
          iTodos.map(iTodo => {
            const _id = iTodo.get('_id');
            return (
              <Todo key={_id} iTodo={iTodo}
                onDeleteTodo={() => onDeleteTodo(_id)}
                onToggleDone={() => onToggleDone(iTodo)} />
            );
          }).valueSeq()
        }
      </ul>
    );
  }
} // end of TodoList class
```

MongoDB uses `_id` for the id property of its records.

**Without call to `valueSeq`, will get this in console:**

Warning: Using Maps as children is not yet fully supported. It is an experimental feature that might be removed. Convert it to a sequence / iterable of keyed ReactElements instead.

# ... todo-list.js

```
const {func, instanceOf} = React.PropTypes;
TodoList.propTypes = {
  iTodos: instanceOf(Immutable.OrderedMap).isRequired,
  onDeleteTodo: func.isRequired,
  onToggleDone: func.isRequired
};

export default TodoList;
```



# todo-app.js ...

```
const axios = require('axios');
import React from 'react';
import ReactDOM from 'react-dom';
import rootReducer from './reducer';
import {createStore} from 'redux';
import TodoHeader from './todo-header';
import TodoList from './todo-list';
import './todo.css';

function handleError(msg, res) {
  store.dispatch({
    type: 'error',
    payload: msg + ': ' + res.data
  });
}

class TodoApp extends React.Component {
```

# ... todo-app.js ...

```
onAddTodo(event) {  
  // Prevent form submission which refreshes page.  
  event.preventDefault();  
  
  const text = store.getState().get('text');  
  
  // Update database.  
  axios.post('/todos', text, {headers: {'Content-Type': 'text/plain'}}).  
    then(res => {  
      const resourceUrl = res.data;  
      // Get the assigned id from the response.  
      const index = resourceUrl.lastIndexOf('/');  
      const _id = resourceUrl.substring(index + 1);  
  
      // Update client-side model.  
      store.dispatch({type: 'addTodo', payload: {_id, text}});  
    }).  
    catch(res => handleError('Error adding todo', res));  
}
```

store is set on slide 34



# ... todo-app.js ...

```
onArchiveCompleted() {  
  // Update database.  
  axios.post('/todos/archive').  
    then(() => {  
    // Update client-side model.  
    store.dispatch({type: 'archiveCompleted'});  
  }).  
  catch(res => handleError('Error archiving todos', res));  
}  
  
onDeleteTodo(todoId) {  
  // Update database.  
  axios.delete('/todos/' + todoId).  
    then(() => {  
    // Update client-side model.  
    store.dispatch({type: 'deleteTodo', payload: {_id: todoId}});  
  }).  
  catch(res => handleError('Error deleting todo', res));  
}
```

# ... todo-app.js ...

```
onTextChanged(event) {
  // Update client-side model.
  store.dispatch({
    type: 'textChange',
    payload: {text: event.target.value}
  });
}

onToggleDone(iTodo) {
  const _id = iTodo.get('_id');
  const done = iTodo.get('done') || false; // gets current value

  // Update database.
  axios.patch('/todos/' + _id, {done: !done}).
    then(() => {
      // Update client-side model.
      store.dispatch({type: 'toggleDone', payload: {_id}});
    }).
    catch(res => handleError('Error toggling todo done', res));
}
```



# ... todo-app.js ...

```
// This component needs to be rendered for every change,  
// so no need for shouldComponentUpdate method.  
render() {  
  const iState = store.getState();  
  const iTodos = iState.get('todos');  
  
  return (  
    <div>  
      <TodoHeader iTodos={iTodos}  
        onArchiveCompleted={this.onArchiveCompleted}/>  
      <div className="error">{iState.get('error')}</div>  
      <br/>  
      <form>  
        <input type="text" size="30" autoFocus  
          placeholder="enter new todo here"  
          value={iState.get('text')}  
          onChange={this.onTextChange}/>  
        <button disabled={!iState.get('text')}  
          onClick={this.onAddTodo}>  
          Add  
        </button>  
      </form>  
  
      <TodoList iTodos={iTodos}  
        onDeleteTodo={this.onDeleteTodo}  
        onToggleDone={this.onToggleDone}/>  
    </div>  
  );  
}  
} // end of TodoApp class
```

don't need **bind** of  
event handling  
methods because  
they don't use **this**

# ... todo-app.js

```
function render() {  
  ReactDOM.render(<TodoApp/>, document.getElementById('content'));  
}  
  
const store = createStore(rootReducer);  
store.subscribe(render);  
  
// Initial hydration of store  
axios.get('todos').  
  then(res => {  
    const todos = res.data;  
    store.dispatch({type: 'setTodos', payload: todos});  
    // This will trigger an event to which the store is subscribed  
    // which will call the render function.  
  }).  
  catch(res => handleError('Error getting todos', res));
```

rootReducer is  
defined on slide 39



# reducer.js ...

This is approach to using Redux is explained on slide 12.

```
import Immutable from 'immutable';

const reducers = {
  addToDo(iState, action) {
    const todo = action.payload;

    // The following commented out lines do the same
    // as the call to withMutations after them,
    // but that is more efficient.
    /*
    return iState.
      set('text', '').
      delete('error').
      setIn(['todos', todo._id], Immutable.fromJS(todo));
    */
    return iState.withMutations(state =>
      state.set('text', ''). // clears input
      delete('error').
      setIn(['todos', todo._id], Immutable.fromJS(todo)));
  },
}
```

an object where  
keys are action types and  
values are action-specific  
reducer functions

todos is an Immutable.js Map  
where keys are todo ids and  
values are todo objects

# ... reducer.js ...

```
archiveCompleted(iState) {  
  const iTodos = iState.get('todos');  
  return iState.  
    delete('error').  
    set('todos', iTodos.filter(iTodo => !iTodo.get('done')));  
},  
  
deleteTodo(iState, action) {  
  return iState.  
    delete('error').  
    deleteIn(['todos', action.payload._id]);  
},  
  
error(iState, action) {  
  return iState.set('error', action.payload);  
},
```



# ... reducer.js ...

```
setTodos(iState, action) {
  const todoArray = action.payload;

  // todos is an array of todo objects, but we need a map
  // where keys are ids and values are immutable todo objects
  const todoMap = todoArray.reduce(
    (map, todo) => {
      map[todo._id] = Immutable.Map(todo);
      return map;
    },
    {}); // todoMap starts as an empty object

  // Unlike Immutable Map, OrderedMap iterates over values in
  // the order they were inserted. This is important for maintaining
  // the order of todos after some are deleted or archived.
  return iState.
    delete('error').
    set('todos', Immutable.OrderedMap(todoMap));
},
```

reduces an array  
to a single object

# ... reducer.js ...

```
textChange(iState, action) {  
  return iState.  
    delete('error').  
    set('text', action.payload.text);  
},  
  
toggleDone(iState, action) {  
  return iState.  
    delete('error').  
    updateIn(  
      ['todos', action.payload._id, 'done'],  
      done => !done);  
}  
}; // end of reducers object
```



# ... reducer.js

```
// Need fromJS instead of Map here because the JS object
// passed contains a property with an object value.
const iInitialState = Immutable.fromJS(
  {text: '', todos: {}});

function rootReducer(iState = iInitialState, action) {
  const type = action.type;
  const reducer = reducers[type];
  if (!reducer && type !== '@@redux/INIT') {
    throw new Error(
      'Redux reducer got unsupported action type "' + type + '"');
  }
  return reducer ? reducer(iState, action) : iState;
}

export default rootReducer;
```

Redux sends @@redux/INIT  
action type to get initial state  
when createStore is called.

# server.js

- Out of scope for this class, but check out the source if interested
  - only 120 lines
- Run with Node.js
- Uses Express library for handling REST calls
- Uses MongoDB to persist and retrieve data



# More Sources

- Awesome Redux - <https://github.com/xgrommx/awesome-redux>
  - contains links to official documentation, presentations, articles, tutorials, starting boilerplates, middlewares, tools, frameworks, examples, similar libraries, Chrome extensions, and other resources



# Other Options

- **Relay** from Facebook
  - “ties GraphQL to React”
  - uses GraphQL queries to fetch data from server
  - Facebook plans to move from Flux to this
- **Falcor** from Netflix
  - fetches data and caches in browser
- Both of these
  - allow clients to specify the data they wish to retrieve in a way that is more flexible than REST where queries are somewhat predefined
  - reduce network traffic compared to REST by allowing clients to retrieve only the data they need and nothing more
  - provide caching to make repeated queries more efficient
  - are WAY MORE COMPLEX than Redux/REST



# Lab

- The `gift-redux` directory contains a version of the `gift` app that uses Redux
- Build and run this app following the steps in `README.txt`
- See notes on `main.js` and `reducer.js` on following slides

# Lab: main.js

- Imports **setupStore** function from `./redux-util`
- Imports **reducer** function from `./reducer`
- Imports event-handling functions from `./gift-event-handlers` and `./name-event-handlers`
- Creates **handlers** object that defines all the event handling methods the app needs
  - each calls the **dispatch** function defined in `./redux-util`
- Defines a **render** function that renders the top-most component, **GiftApp**
  - passes the **handlers** and **store** objects to it
- Creates the Redux **store** using **setupStore**
- Makes initial call to **render**

These event handling methods only call **dispatch**, but some apps would do more. For example, dispatch an action to set state that will cause a spinner to be rendered, make a REST call, dispatch another action if it is successful, a different action if there is an error, and another action to clear the spinner.

```
dispatch('spinner', true);
axios.get(restUrl).
  then(res => {
    dispatch('setThing', res.data);
    dispatch('spinner', false);
  }).
  catch(res => {
    dispatch('error', res.data);
    dispatch('spinner', false);
  });
```



# Lab: `reducer.js`

- Imports the Immutable library
- Imports reducers from two categories, names and gifts
- Combines these into a single **`reducers`** object that contains all the reducer functions the app needs
  - allows them to be looked up by name
  - each takes the current state object
  - some take an additional **`payload`** parameter
  - each creates a new version of the state and returns it
- Defines the initial state of the app which is an Immutable **`Map`**
- Defines and exports the top reducer function
  - selects a reducer function based on value of **`action.type`**
  - extracts payload from **`action`** object, if any
  - calls reducer function and returns its result which is the new state



# Lab Steps

- Add a “Purge” button at bottom of UI that deletes all names that have no gifts
- Modify the following files

- **name-reducers.js**

- add a **purge** reducer function that takes `state`, gets the `gifts Map` from `state`, iterates over the names and `Lists` of gifts for each name, checks the size of the gifts `List`, if zero, changes `state` to the result of calling `reducers.deleteName(state, name)`, and returns the final `state` value

**Tip** to iterate over the keys and values of the `gifts` Immutable `Map`:

```
for (const [name, giftsForName] of gifts.entries()) {
```

- **name-event-handlers.js**

- add an **onPurge** method that dispatches a “purge” action

- **gift-app.js**

- add a `Button` at the bottom of the **render** method with an `onClick` handler set to `handlers.onPurge`

- Test the new “Purge” button