**Advance JS**

**Redux Setup**

**MiddleWare wit WebserviceCall)**

**Step 1:**

**Create new folder (myservice)**

**Step 2:**

**cd myservice**

**Step 3:**

**npm init**

**Step 4: install express (is web pack to provider apis (middle layer))**

**npm install express**

**npm install expressjs**

**Step 5: import module library**

**var express = require(‘express’);**

**Step 6: write some code to expose the service**

**Step 7: run the middleware service**

**node app.js**

**promise api :**

**fetch(“**http://localhost:3010/**”)**

**Advance JS with Redux**

Redux is a predictable state container for JavaScript apps.

npm install --save redux

npm install react-redux

npm install --save-dev redux-devtools (optional)

### Flux

Redux was inspired by several important qualities of Flux. Like Flux, Redux prescribes that you concentrate your model update logic in a certain layer of your application (“stores” in Flux, “reducers” in Redux). Instead of letting the application code directly mutate the data, both tell you to describe every mutation as a plain object called an “action”.

Unlike Flux, **Redux does not have the concept of a Dispatcher**. This is because it relies on pure functions instead of event emitters, and pure functions are easy to compose and don't need an additional entity managing them. Depending on how you view Flux, you may see this as either a deviation or an implementation detail. Flux has often been described as (state, action) => state. In this sense, Redux is true to the Flux architecture, but makes it simpler thanks to pure functions.

Another important difference from Flux is that **Redux assumes you never mutate your data**. You can use plain objects and arrays for your state just fine, but mutating them inside the reducers is strongly discouraged. You should always return a new object, which is easy with the object spread operator proposal, or with a library like Immutable.

**Redux**

**Store (data store)**

**Action (Event)**

**State (current state – just like session )**

**Dispatch (call to / push to invoke to event with data/state )**

**Connect (pull )**

**Example:**

**import { createStore } from 'redux'**

**function counter(state = 0, action) {**

**switch (action.type) {**

**case 'INCREMENT':**

**return state + 1**

**case 'DECREMENT':**

**return state - 1**

**default:**

**return state**

**}**

**}**

**// Create a Redux store holding the state of your app.**

**// Its API is { subscribe, dispatch, getState }.**

**let store = createStore(counter)**

**// You can use subscribe() to update the UI in response to state changes.**

**// Normally you'd use a view binding library (e.g. React Redux) rather than subscribe() directly.**

**// However it can also be handy to persist the current state in the localStorage.**

**store.subscribe(() => console.log(store.getState()))**

**// The only way to mutate the internal state is to dispatch an action.**

**// The actions can be serialized, logged or stored and later replayed.**

**store.dispatch({ type: 'INCREMENT' })**

**// 1**

**store.dispatch({ type: 'INCREMENT' })**

**// 2**

**store.dispatch({ type: 'DECREMENT' })**

**// 1**

**Concepts:**

**{**

**todos: [{**

**text: ‘food',**

**completed: true**

**}, {**

**text: 'Exercise',**

**completed: false**

**}],**

**visibilityFilter: 'SHOW\_COMPLETED'**

**}**

**--**

**{ type: 'ADD\_TODO', text: 'Go to swimming pool' }**

**{ type: 'TOGGLE\_TODO', index: 1 }**

**{ type: 'SET\_VISIBILITY\_FILTER', filter: 'SHOW\_ALL' }**

**--**

**function visibilityFilter(state = 'SHOW\_ALL', action) {**

**if (action.type === 'SET\_VISIBILITY\_FILTER') {**

**return action.filter**

**} else {**

**return state**

**}**

**}**

**function todos(state = [], action) {**

**switch (action.type) {**

**case 'ADD\_TODO':**

**return state.concat([{ text: action.text, completed: false }])**

**case 'TOGGLE\_TODO':**

**return state.map((todo, index) =>**

**action.index === index**

**? { text: todo.text, completed: !todo.completed }**

**: todo**

**)**

**default:**

**return state**

**}**

**}**

**--**

**function todoApp(state = {}, action) {**

**return {**

**todos: todos(state.todos, action),**

**visibilityFilter: visibilityFilter(state.visibilityFilter, action)**

**}**

**}**

**Three Principles**

**The state of your whole application is stored in an object tree within a single store.**

This makes it easy to create universal apps, as the state from your server can be serialized and hydrated into the client with no extra coding effort. A single state tree also makes it easier to debug or inspect an application; it also enables you to persist your app's state in development, for a faster development cycle. Some functionality which has been traditionally difficult to implement - Undo/Redo, for example - can suddenly become trivial to implement, if all of your state is stored in a single tree.

**console.log(store.getState())**

### State is read-only

**The only way to change the state is to emit an action, an object describing what happened.**

**store.dispatch({**

**type: 'COMPLETE\_TODO',**

**index: 1**

**})**

**store.dispatch({**

**type: 'SET\_VISIBILITY\_FILTER',**

**filter: 'SHOW\_COMPLETED'**

**})**

### Changes are made with pure functions

**To specify how the state tree is transformed by actions, you write pure reducers.**

**function visibilityFilter(state = 'SHOW\_ALL', action) {**

**switch (action.type) {**

**case 'SET\_VISIBILITY\_FILTER':**

**return action.filter**

**default:**

**return state**

**}**

**}**

**function todos(state = [], action) {**

**switch (action.type) {**

**case 'ADD\_TODO':**

**return [**

**...state,**

**{**

**text: action.text,**

**completed: false**

**}**

**]**

**case 'COMPLETE\_TODO':**

**return state.map((todo, index) => {**

**if (index === action.index) {**

**return Object.assign({}, todo, {**

**completed: true**

**})**

**}**

**return todo**

**})**

**default:**

**return state**

**}**

**}**

**import { combineReducers, createStore } from 'redux'**

**const reducer = combineReducers({ visibilityFilter, todos })**

**const store = createStore(reducer)**

# Basics

Don't be fooled by all the fancy talk about reducers, middleware, store enhancers—Redux is incredibly simple. If you've ever built a Flux application, you will feel right at home. If you're new to Flux, it's easy too!

* Actions
* Reducers
* Store
* Data Flow
* Usage with React

# Actions