Redux

**Basics :-**

## Actions :-

**Actions** are payloads of information that send data from your application to your store. They are the only source of information for the store. You send them to the store using [store.dispatch()](https://redux.js.org/api/store" \l "dispatchaction).

Actions are plain JavaScript objects. Actions must have a type property that indicates the type of action being performed

The dispatch() function can be accessed directly from the store as [store.dispatch()](https://redux.js.org/api/store" \l "dispatchaction), but more likely you'll access it using a helper like [react-redux](http://github.com/gaearon/react-redux)'s connect(). You can use [bindActionCreators()](https://redux.js.org/api/bindactioncreators) to automatically bind many action creators to a dispatch() function.

Action creators can also be asynchronous and have side-effects. You can read about [async actions](https://redux.js.org/advanced/async-actions) in the [advanced tutorial](https://redux.js.org/advanced/advanced-tutorial) to learn how to handle AJAX responses and compose action creators into async control flow.

## Reducer :-

**Reducers** specify how the application's state changes in response to [actions](https://redux.js.org/basics/actions) sent to the store. Remember that actions only describe what happened, but don't describe how the application's state changes.

The reducer is a pure function that takes the previous state and an action, and returns the next state.

e.g

(previousState, action) => nextState

It's called a reducer because it's the type of function you would pass to [Array.prototype.reduce(reducer, ?initialValue)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/Reduce). It's very important that the reducer stays pure.

* Perform side effects like API calls and routing transitions;
* Call non-pure functions, e.g. Date.now() or Math.random().

# Store :-

# In the previous sections, we defined the [actions](https://redux.js.org/basics/actions) that represent the facts about “what happened” and the [reducers](https://redux.js.org/basics/reducers) that update the state according to those actions.

The **Store** is the object that brings them together. The store has the following responsibilities:

* Holds application state;
* Allows access to state via [getState()](https://redux.js.org/api/store" \l "getState);
* Allows state to be updated via [dispatch(action)](https://redux.js.org/api/store#dispatchaction);
* Registers listeners via [subscribe(listener)](https://redux.js.org/api/store#subscribelistener);
* Handles unregistering of listeners via the function returned by [subscribe(listener)](https://redux.js.org/api/store#subscribelistener).

It's important to note that you'll only have a single store in a Redux application. When you want to split your data handling logic, you'll use [reducer composition](https://redux.js.org/basics/reducers#splitting-reducers) instead of many stores.

# Data Flow :-

# Redux architecture revolves around a ****strict unidirectional data flow****.

# This means that all data in an application follows the same lifecycle pattern, making the logic of your app more predictable and easier to understand.

# The data lifecycle in any Redux app follows these 4 steps:

1. **You call** [store.dispatch(action)](https://redux.js.org/api/store" \l "dispatchaction).

{ type: 'LIKE\_ARTICLE', articleId: 42 }

{ type: 'FETCH\_USER\_SUCCESS', response: { id: 3, name: 'Mary' } }

{ type: 'ADD\_TODO', text: 'Read the Redux docs.' }

# Think of an action as a very brief snippet of news. “Mary liked article 42.” or “'Read the Redux docs.' was added to the list of todos.”

# You can call [store.dispatch(action)](https://redux.js.org/api/store" \l "dispatchaction) from anywhere in your app, including components and XHR callbacks, or even at scheduled intervals.

1. **The Redux store calls the reducer function you gave it.**

# The [store](https://redux.js.org/basics/store) will pass two arguments to the [reducer](https://redux.js.org/basics/reducers): the current state tree and the action. For example, in the todo app, the root reducer might receive something like this:

// The current application state (list of todos and chosen filter)

let previousState = {

visibleTodoFilter: 'SHOW\_ALL',

todos: [

{

text: 'Read the docs.',

complete: false

}

]

}

// The action being performed (adding a todo)

let action = {

type: 'ADD\_TODO',

text: 'Understand the flow.'

}

// Your reducer returns the next application state

let nextState = todoApp(previousState, action)

**Note :-** that a reducer is a pure function. It only computes the next state. It should be completely predictable: calling it with the same inputs many times should produce the same outputs. It shouldn't perform any side effects like API calls or router transitions. These should happen before an action is dispatched.

1. **The root reducer may combine the output of multiple reducers into a single state tree.**

# How you structure the root reducer is completely up to you. Redux ships with a [combineReducers()](https://redux.js.org/api/combinereducers) helper function, useful for “splitting” the root reducer into separate functions that each manage one branch of the state tree.

# Here's how [combineReducers()](https://redux.js.org/api/combinereducers) works. Let's say you have two reducers, one for a list of todos, and another for the currently selected filter setting:

function todos(state = [], action) {

// Somehow calculate it...

return nextState

}

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

// Somehow calculate it...

return nextState

}

let todoApp = combineReducers({

todos,

visibleTodoFilter

})

When you emit an action, todoApp returned by combineReducers will call both reducers:

let nextTodos = todos(state.todos, action)

let nextVisibleTodoFilter = visibleTodoFilter(state.visibleTodoFilter, action)

It will then combine both sets of results into a single state tree:

return {

todos: nextTodos,

visibleTodoFilter: nextVisibleTodoFilter

}

While [combineReducers()](https://redux.js.org/api/combinereducers) is a handy helper utility, you don't have to use it; feel free to write your own root reducer!

1. **The Redux store saves the complete state tree returned by the root reducer.**

This new tree is now the next state of your app! Every listener registered with [store.subscribe(listener)](https://redux.js.org/api/store" \l "subscribelistener) will now be invoked; listeners may call [store.getState()](https://redux.js.org/api/store" \l "getState) to get the current state.

Now, the UI can be updated to reflect the new state. If you use bindings like [React Redux](https://github.com/gaearon/react-redux), this is the point at which component.setState(newState) is called.

# Usage with React

From the very beginning, we need to stress that Redux has no relation to React. You can write Redux apps with React, Angular, Ember, jQuery, or vanilla JavaScript.

That said, Redux works especially well with libraries like [React](http://facebook.github.io/react/) and [Deku](https://github.com/dekujs/deku) because they let you describe UI as a function of state, and Redux emits state updates in response to actions.

We will use React to build our simple todo app, and cover the basics of how to use React with Redux.

## Installing React Redux :- [React bindings](https://github.com/reduxjs/react-redux) are not included in Redux by default. You need to install them explicitly:

npm install --save react-redux

# Why Use React Redux?

Redux itself is a standalone library that can be used with any UI layer or framework, including React, Angular, Vue, Ember, and vanilla JS. Although Redux and React are commonly used together, they are independent of each other.

If you are using Redux with any kind of UI framework, you will normally use a "UI binding" library to tie Redux together with your UI framework, rather than directly interacting with the store from your UI code.

**React Redux is the official Redux UI binding library for React**. If you are using Redux and React together, you should also use React Redux to bind these two libraries.

To understand why you should use React Redux, it may help to understand what a "UI binding library" does.

# 