Redux (The state management Library)

Redux is a state management library. The state mentioned here is same as that of state in a React Component. Instead of maintaining a state in a React component, we lend it to be managed by Redux Library instead.

The React Library is focused about rendering the User Interface and handling interactions. It is not supposed to maintain state.

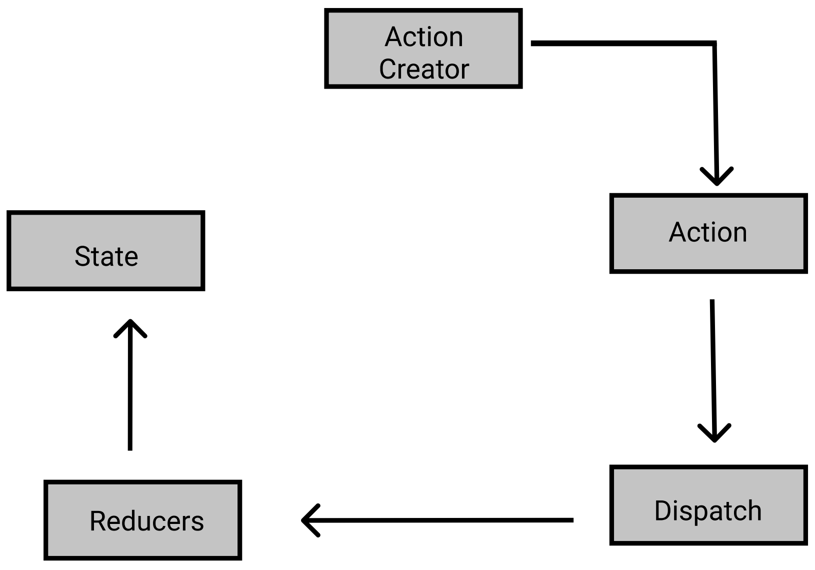
In general, Redux allows creation of complex applications easier.

Redux Library is not explicitly designed for React. It has been ported to other languages. It is important to note that React is not essential for making use of Redux. In other words, Redux is independent of React.

How Redux Works Internally

Redux Cyle

|*Action Creator*| --> |*Action*| --> |*Dispatch*| ---> |*Reducers*| ---> |*State*|



Redux follows a cycle known as Redux Cycle. By cycle what i mean is that the order in which the execution is done remains same and cannot be altered. This cycle starts with an Action Creator that produces an action. This action is ingested by a Dispatcher which creates copies and distributes each copy to a reducer. The reducer is finally responsible for mutation.

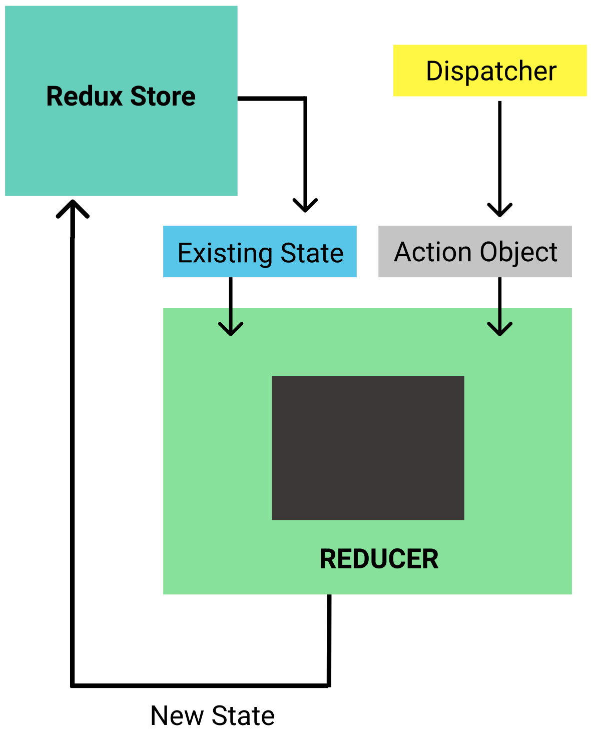
***Action Creator** : an action creator produces an action object.*
A sample example of creating a function that works as an Action creator is as follows:

```
const createPolicy = (name,amount) =>{
  return {
    //Action (form)

    type:"CREATE_POLICY",
    payload:{
      name:name,
      amount:amount
    }
  }
}

const deletePolicy = (name) =>{
  return {
    type:"DELETE_POLICY",
    payload :{
      name
    }
  }
}


const createClaim = (name,amountToCollect) =>{
  return {
    type:"CREATE_CLAIM",
    payload : {
      name,
      amountOfMoneyToCollect:amountToCollect
    }
  }
}
```

***Action:** an action contains some data, which allows the reducers to act on the state and decide wheater to mutate the state or not. And how the mutation should occur.*

***Dispatch** : can be considered as an action copier, it takes an action and creates copy of action object one for each reducer. Then it forwards a copy of action to the reducer.*

Reducer: *takes an action as an input. A reducer takes in two parameters.*
The first argument to a reducer is the previous state that the reducer is concerend about.
The second argument being the action object that is passed to it by the dispatcher.
The reducer based on some logic mutates the data. It is important to note that each reducer must return the state back, in both the cases, wheather the mutatation has generated a new state or the existing old state.



```
//Reducres (Departments)

const claimsHistory = (oldListOfClaims = [] , action) =>{
  if(action.type === "CREATE_CLAIM"){
    //this department cares about this form
    return [...oldListOfClaims,action.payload]
  }

  // we dont care about this form

  return oldListOfClaims;
}


const accounting = (bagOfMoney = 100,action) => {
  if(action.type === "CREATE_CLAIM"){
    return bagOfMoney - action.payload.amountOfMoneyToCollect;
  }

  else if(action.type === "CREATE_POLICY"){
    return bagOfMoney + action.payload.amount;
  }

  return bagOfMoney;
}


const policies = (listOfPolicies = [],action) => {
  if(action.type === "CREATE_POLICY"){
    return [...listOfPolicies,action.payload.name]
  }
  else if(action.type === "DELETE_POLICY"){
    return listOfPolicies.filter(name => name!==action.payload.name);
  }

  return listOfPolicies
}
```

Creating a Store with Redux

```
const {createStore, combineReducers} = Redux;

const ourDepartments = combineReducers({
  accounting:accounting,
  claimsHistory:claimsHistory,
  policies:policies
})

const store = createStore(ourDepartments);

const action = createPolicy("Hyder",200);

store.dispatch(action);

console.log(store.getState())
```

The combine Reducers function combines all the reducers together through an object.
The Keys correspond to state property name created through that reducer.
For Example a console log will give us.

Each dispatch runs entire cycle.

Now we can only modify the state through the dispatch function using action creators through action. There is no other way or direct way to modify it.

Why to use Redux ?

Although Redux has an initial friction , however as time progresses and code base increases, we can easily manange the application since the state can only be mutated through the use of reducers.

In a way the reducers are self documenting the codebase, this helps to keep the codebase clean and help other engineers to understand the logic and application much more easily.

Integerating React with Redux

For integerating Redux with React, we need to install few packages using npm

the command is : npm install –save redux react-redux

***Folder Structure** : The another important aspect of using Redux is to maintain the folder structure. Typically, following structure is followed*

```
/src [Parent Folder]

      [Children]
      /actions  -> files related to action creators
      /components  -> components
      /reducres -> files related to reducers + export of combineReducers
      /index.js   -> sets up both React and Redux of the app
```

Named Vs Default Exports

a named export allows us to export multiple functions from a single file. To use named export we just need to write export keyword before the function we are trying to export out of that file. An example can be shown as

Named Export index.js

```
//Action Creator

export const selectSong = (song) => {
//Return Action

return {
type: "SONG_SELECTED",
payload: song,
  };
};
```

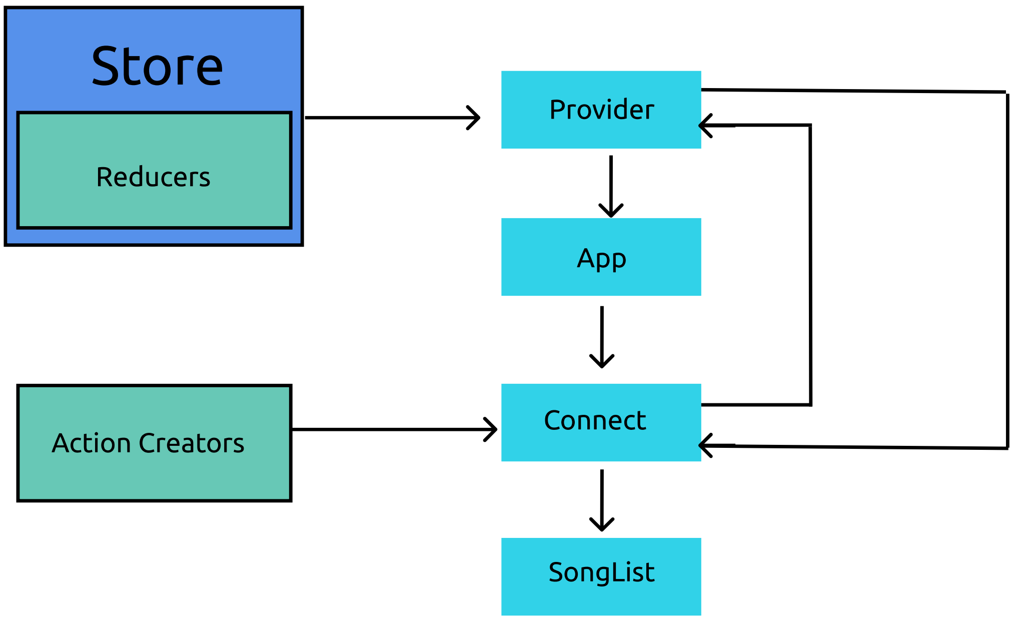
Importing from a Named Export File

```
import { selectSong } from “./index”
```

Thus for named imports we need to use curly braces while incase of Default there is no need of curly braces.

React Redux Architecture:

earlier we installed a 3th party library called react-redux.



Provider *is a component that is provided via react-redux library. It takes a prop called store. This store is the actual data store, that is created by calling the combineReducers method of Redux. Even the App component is placed inside this Provider component.*

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

import App from "./components/App";
import reducers from "./reducers";

ReactDOM.render(
<Provider store={createStore(reducers)}>
<App />
</Provider>,
document.querySelector("#root")
);
```

This code is to written inside main index.js

Connect: *is a funciton / component that communicates with the provider as shown in the diagram above. To remind ourselves Provider holds the state.*
It is available as a named import from the react-redux library
import {connect} from 'react-redux';

*There is typically the existence of a function called **mapStateToProps**. This function takes state as argument.*

```
import React from "react";
import { connect } from "react-redux";

class SongList extends React.Component {
render() {
return <div>SongList</div>;
}
}

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

export default connect(mapStateToProps)(SongList);
```

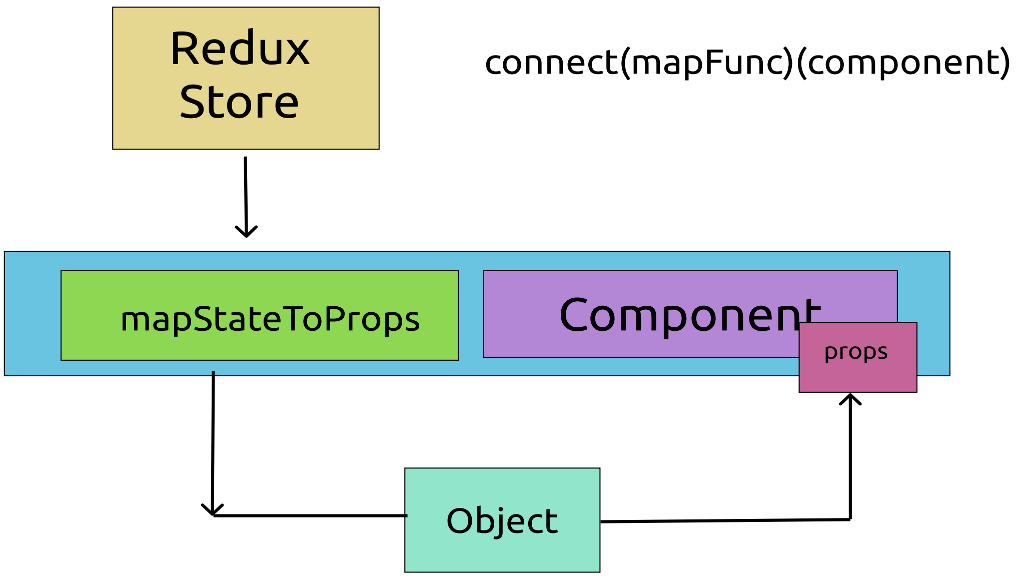
Conenct function in depth

connect function returns another function that is the reason we have this syntax connect()();

*The first parameter it takes is a mapper function typically labelled as **mapStateToProps.** This function is automatically given state by the connect function. This function needs to have a return object. The object defined will dictate the state props available to the second incoming argument Component to the connect function.*

The second paramter to the connect function is the component to which mapStateToProps is attached.

The combined effect of this instruction can be visualized as SongList component having a prop with name songs.



```
const mapStateToProps = (state) => {
return { songs: state.songs };
};

export default connect(mapStateToProps)(SongList);
```

Thus these two lines means that mapStateToProps will return an object with key songs which will be equal to the state property of songs.
Finally this will be send to SongList component as a prop. Thus SongList component will have access to this data via a prop called as songs (same as key of the return object of the map function).

Action Creators and their Usage:

We import the action creator in the component but do not directly use it. However, the action creator is passed to the connect function as the second parameter after mapStateTo Props function.

But Why do we need to pass it to the connect function?
The answer to this question is that Redux is no Magic. It cannot detect an action creator being called if we use it directly. Since if it fails to detect an action creator being called, it will never run the Redux Cycle automatically.
Thus an manually called action creator will never make its way upto dispatch and reducers. Thus we need to register it with redux using that connect function.

```
import { selectSong } from "../actions";

export default connect(
mapStateToProps,

{selectSong}

)(SongList);
```

/ **Describe the creation process of a simple React-Redux Counter Application** */*

Loading External Data into Redux using APIs.

A typical data loading scenario using Redux involves the following steps.

1) Component gets rendered onto the screen.
2) Component's "componentDidMount" LifeCycle method gets called.

Note: Since we are going to use Lifecycle Methods, we will be using class based Component.

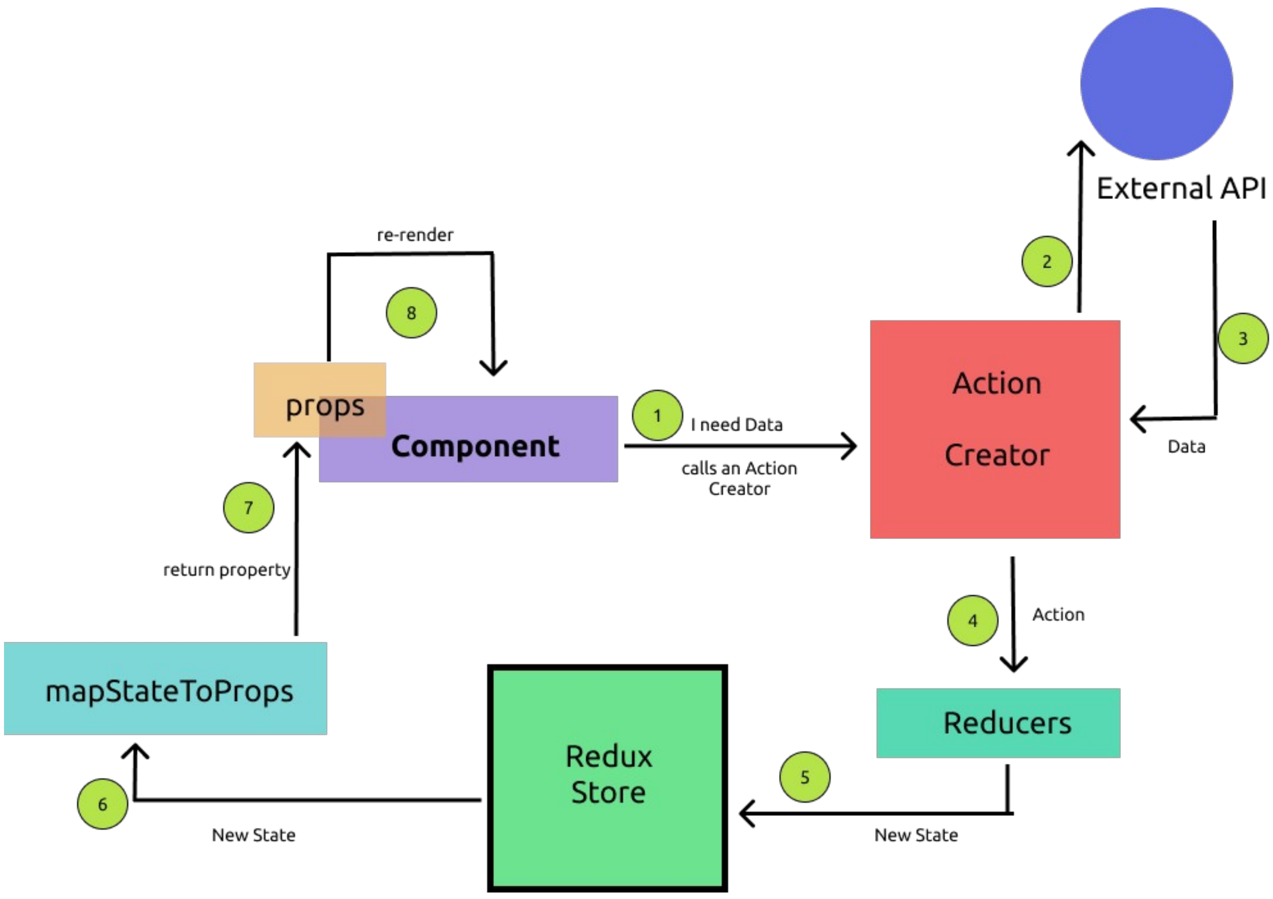
3) We call action creator from "componentDidMount"
4) Acction Creator runs the code to make an API request
5) API responds with data
6) Action creator returns Action with fetched data from API on the payload of action.
7) Dispatch method propogates this Action to all the Reducers.
8) Some pre-configured Reducer sees this action, returns the data off the payload.
9) Because a new state object is generated , react-redux will cause React App to re-render.

Generally

Components are responsible for calling Action Creators if they require external data.

Action Creators are responsible for making API requests [where Redux Thunk comes into picture]

We get fetched data into component by generating new data into our redux store. The data is passed into Component through mapStateToProps



Understanding the API call from Action Creator.

Suppose we have an api configured in apis directory with a baseURL property.Now we want to use this to make a API call inside an action creator.

Following code seems straighforward implementation of the above concept.

Inside "/actions/index.js"

wrong implementation

```
import jsonPlaceholder from "../apis/jsonPlaceholder";

 export const fetchPosts = async () => {
   const response = await jsonPlaceholder.get("/posts");

   return {
     type: "FETCH_POSTS",
     payload: response,
   };
 };
```

1) But the above code wont work. Becasue the above method doesnt return an action but "Request Object" which is not a plain JS object. This problem is caused because of async await syntax we are using.

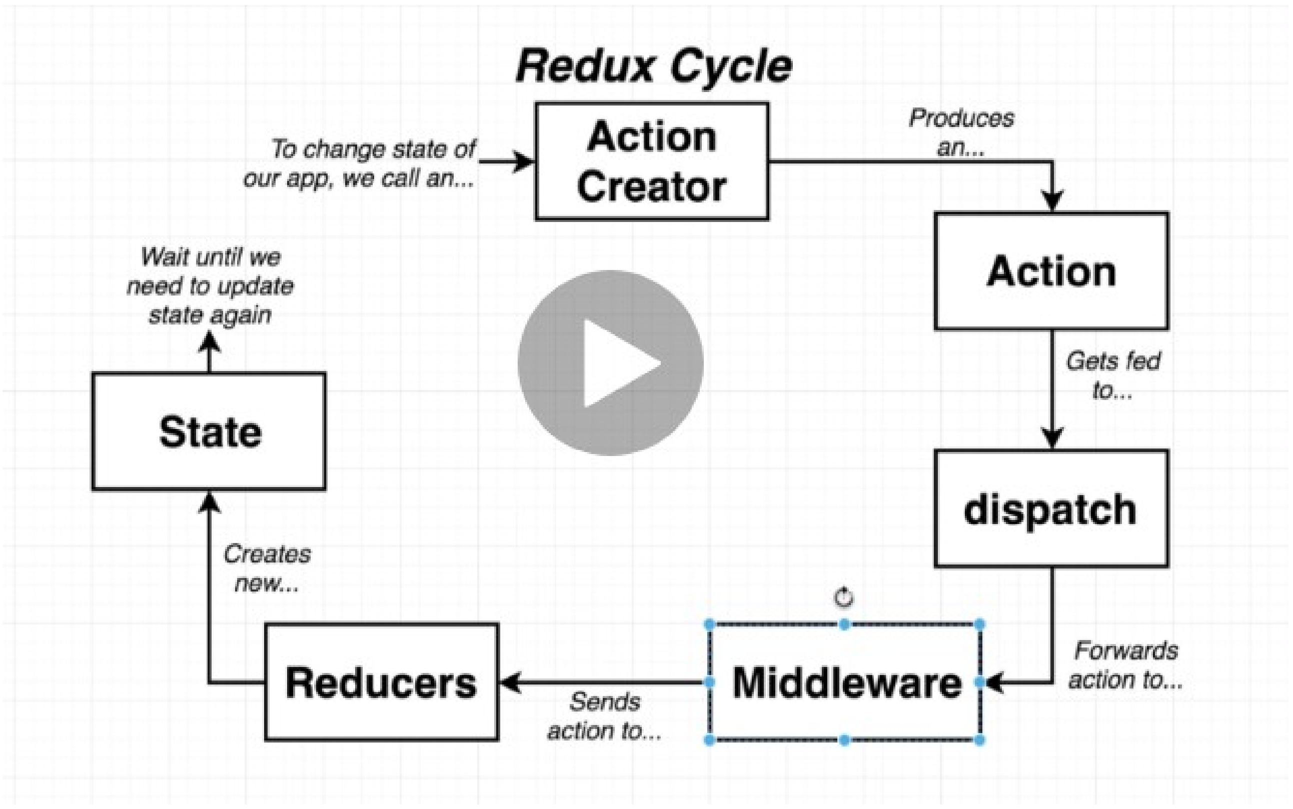
2) If we dont use async await. We will send the Object to reducer without the data.

Using middleware – Redux-Thunk.

There are two types of Action Creators

- ***Synchronous Action Creators:*** *a sync action creator is one, which returns an action with data readily or instantaneously.*
- ***Asynchronous Action Creators:*** *an async action creator is one, which takes some amount of time for collecting the data. For handling asynchronous behaviour these action creators use middlewares. Redux-Thunk is a middleware.*

Middlewares in the Redux Cycle.



Rather than directly passing on the action object to reducers. The action is first passed onto middlewares.

Middleware is a plain JS function that gets called with every action that we dispatch. Middleware has ability to stop, modify, or mess around with actions.

There are alot of open source middlewares. Redux Thunk is a middleware to help with async actions.

Redux Thunk

Redux thunk helps with async actions. It is a popular middleware.

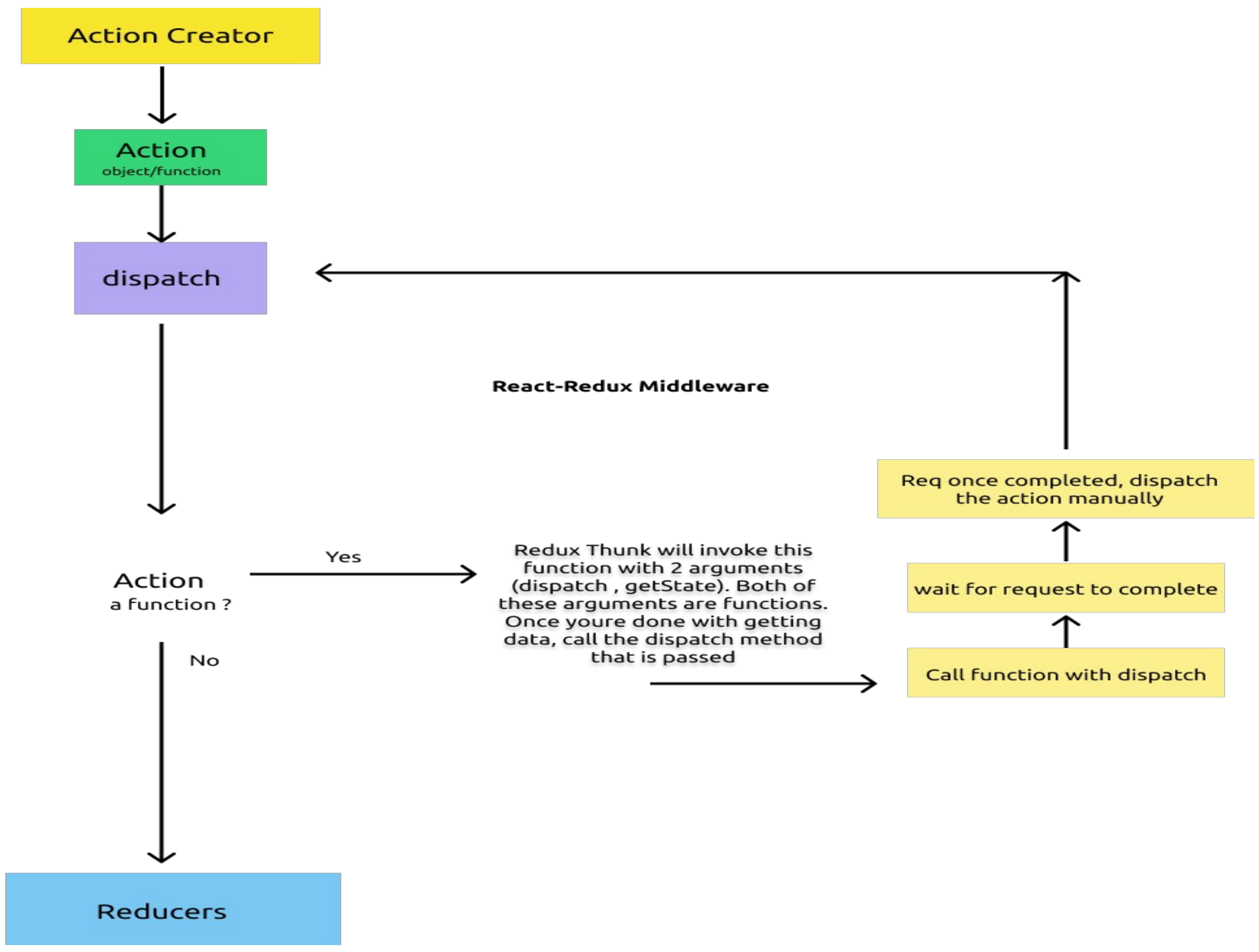
The normal Rules
1. Action Creators must return action objects.
2. Actions must have a type property.
3. Actions can optionally have a payload.

Rules with Redux-Thunk

1. Action creators can return action objects or return functions.

2. If action object is returned, it must have a type. If a function is returned, it executes that fn.

3. If action object is returned, it can optionally have a paylaod prop.


Action Creator
Action
object/function
dispatch
React-Redux Middleware
Action
a function ?
Yes
No
Redux Thunk will invoke this function with 2 arguments (dispatch , getState). Both of these arguments are functions. Once youre done with getting data, call the dispatch method that is passed
Call function with dispatch
wait for request to complete
Req once completed, dispatch the action manually
Reducers

Thus with Redux Thunk we manually dispatch an action through middlewares.

With Redux Thunk instead of just returning a action object , we can return functions. When functions are returned Redux Thunk will call that function with dispatch paramter. Once the function has completed its execution, it will manually call the dispatch.

The working code.

```
export const fetchPosts = () => {
return async function (dispatch, getState) {
const response = await jsonPlaceholder.get("/posts");
dispatch({ type: "FETCH_POSTS", payload: response });
};
};
```

Rules of Reducers:

1) a reducer can return anything expect undefined. For undefined an error occurs. By default JavaScript functions return “undefined”, if the return is not mentioned explicitly.

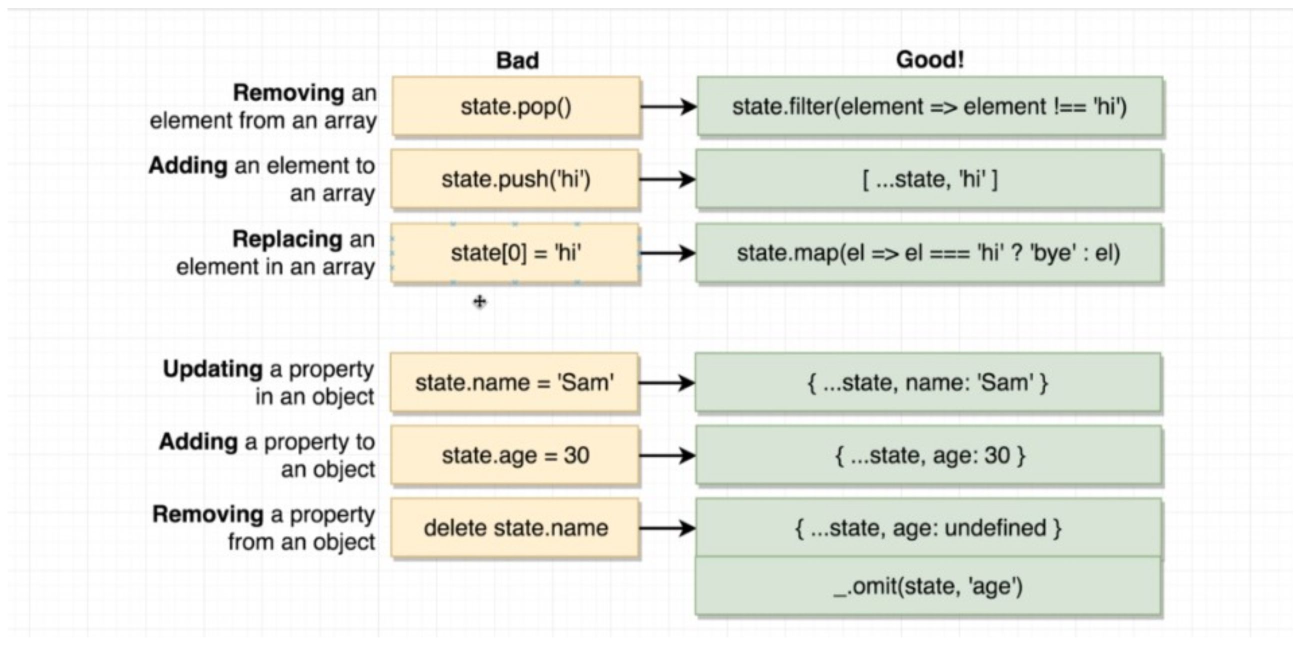
2) a reducer can produce a new state only using consuming the previous state and an action. Thus the initial state cannot be undefined, to handle this we can use default arguments.

3) Reducers are Pure. That is they should not depend upon some other function expect itself.

4) Reducer must not mutate previous state argument.

Dont operate on previous state ie dont return the previous state exactly return a new reference because it compares the memory references for re-rendering

Safe State Updates inside Reducers



	Bad	Good!
Removing an element from an array	state.pop()	state.filter(element => element !== 'hi')
Adding an element to an array	state.push('hi')	[...state, 'hi']
Replacing an element in an array	state[0] = 'hi'	state.map(el => el === 'hi' ? 'bye' : el)
Updating a property in an object	state.name = 'Sam'	{ ...state, name: 'Sam' }
Adding a property to an object	state.age = 30	{ ...state, age: 30 }
Removing a property from an object	delete state.name	{ ...state, age: undefined }
		_.omit(state, 'age')

SWITCH CASE SYNTAX INSIDE REDUCERS

usually a reducer can care for more than once action objects, in such cases, we cannot rely on if else ladder. Hence more appropriately we use Switch Case Syntax.

```
export default (state = [], action) => {
  switch (action.type) {
    case "FETCH_POSTS":
      return action.payload;
    default:
      return state;
  }
```

Extracting Component Logic into MapStateToProps.

In order to make the component more reusable, the logic is extracted to mapStateToProps function. Thus the logic related to redux store and state is extracted to MapStateToProps

MapStateToProps takes 2 arguments
First : State from the store
Second : the props of the component

```
const mapStateToProps = (state, ownProps) => {
  return {
    user: state.users.find((user) => {
      return user.id === ownProps.userId;
    }),
  };

};
```