**1.1 What is state?**

When something is changed with time is called state. When an action is needed with any changes then we use state management.

**State management with raw JavaScript:** old school way, code redundancy, increase code complexity

If we could have more counter we need more dom element selection and each time need update selection element

**Cons: below code snap**

code redundancy, more code, hard to maintain for large application, etc problem occurs.

// select DOM element

const increment = document.querySelector("#increment");

const decrement = document.querySelector("#decrement");

const counterEl = document.querySelector("#counter");

// initial state

let count = 0;

// event listeners

increment.addEventListener("click", () => {

  count++;

  counterEl.innerText = count;

});

decrement.addEventListener("click", () => {

  count--;

  counterEl.innerText = count;

});

**State management beauty: or modern UI Technology like React**

* Code reusability
* Less code and no redundancy
* Efficient, React itself manage the state

**1.2 Reactive Ui Reusability & React State Management issues**

**Project code inside:** Reactive Reusability folder

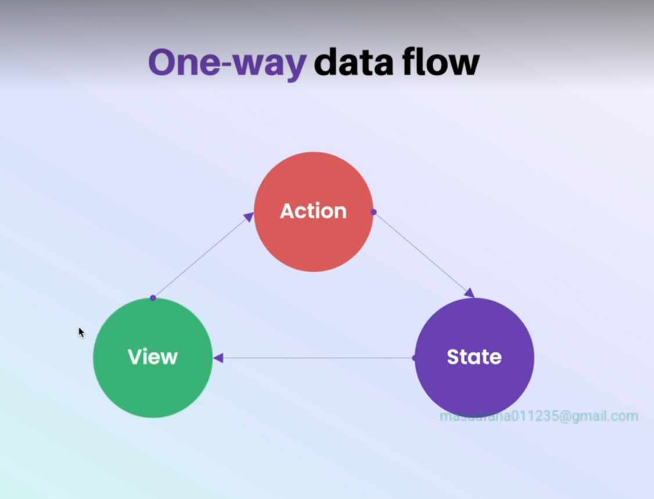
We have a **App parent component and two child component Counter, Count of this App.**

We need to pass the two counter summation into **Count** component. In general we have handler into **Counter component,** but **Count** component inside parent **App** component. We cannot pass from Counter component to App and then Count component.

That is why we **shifting up** the handler into parent component and prop drilling and passing to the counter component. Then parent to **Count** component we pass the total count.

But if our application is large then prop drilling or unnecessary prop pass to such component whose is actually not need. This is really so bad things.

**How State, handler and View work? React data flow**

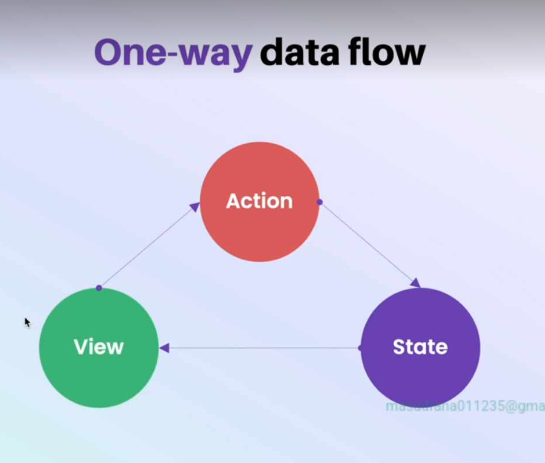
****

*Figure: one-way data flow*

**1.4 Introduction to Redux**

**Redux:** Redux is a flexible state container for JavaScript apps that manages our application state separately

What if we have a central store where necessary data is stored and pass whom is need. No need prop drilling. This approach is awesome.



**Real life concept:** suppose we have subscribe a newsletter, when a new news a published we get this news. Or we have subscribed a youtube channel when a new video is published we will get this video.

Redux: flexible state container. Created by using JavaScript. Any JavaScript Library or frameworks support redux event NodeJS support redux.

**1.5 How redux store works? Different redux component**

When user take an action, This command go to the store. **This is called action.** Means action dispatch

To define object we need object. Which has must type property

When action is done, now some function will work or business login. This is called reducer. Mainly control state

Reducer is a function it takes **state**, and **action** from user as parameter and return **newState** . newState is current state. All the subscribed component will got this newState mainly updated value

**1.6 Simple counter application with redux**

**Initial State:** pass to the reducer

// initial state

const initialState = {

  value: 0,

};

**Make a reducer:** it work based on action dispatch type

// create reducer function: bussiness logic

function counterReducer(state = initialState, action) {

  if (action.type === "increment") {

    return {

      ...state,

      value: state.value + 1,

    };

  } else if (action.type === "decrement") {

    return {

      ...state,

      value: state.value - 1,

    };

  } else {

    return state;

  }

}

**Create store(pass reducer to it) and store.subscribe() method** with a render callback function that will give new state by **getState()** after store is updating**:** for changing UI when action is done or state is changed

// create store

const store = Redux.createStore(counterReducer);

const render = () => {

  const state = store.getState();

  counterEl.innerText = state.value;

};

// update UI initially

render();

// when store is updated, render callback function will be called

store.subscribe(render);

**action dispatch with action type:**

// button click listener

incrementEl.addEventListener("click", () => {

  store.dispatch({

    type: "increment",

  });

});

DecrementEl.addEventListener("click", () => {

  store.dispatch({

    type: "decrement",

  });

});

**1.7 Action payloads and creators**

We can pass payload or additional information to the reducer using action dispatch. We can pass a single property or an object to as payload. Since action is an object, It has type mandatory property and we can pass addition property to this action dispatch. Previous application we have use payloads to increment and decrement dynamically pass to the action object.

// create reducer function: bussiness logic

function counterReducer(state = initialState, action) {

  if (action.type === "increment") {

    return {

      ...state,

      value: state.value + action.payload,

    };

  } else if (action.type === "decrement") {

    return {

      ...state,

      value: state.value - action.payload,

    };

  } else {

    return state;

  }

}

Dynamically pass the increment decrement value:

incrementEl.addEventListener("click", () => {

  store.dispatch({

    type: "increment",

    payload: 5,

  });

});

Common problem solved:

Use action identifiers as constant variable so that when we change this variable then all place will be automatically changed. Action identifiers or action type write as variable and variable name use all place so that single change will change the all place

// action indentifies

const INCREMETN = "increment";

const DECREMENT = "decrement";

// create reducer function: bussiness logic

function counterReducer(state = initialState, action) {

  if (action.type === INCREMETN) {

    return {

      ...state,

      value: state.value + action.payload,

    };

  } else if (action.type === DECREMENT) {

    return {

      ...state,

      value: state.value - action.payload,

    };

  } else {

    return state;

  }

}

NOW we will change the variable value like increment to ‘add’. All place will get ‘add’ string. This problem is solved but if we have many increment and decrement. We need to write more times same task like store.dispatch() function with same object. Now we will solve this issue.

**Now we will create action creator**: means action is done seperately

// action creators

// increment action creators

const increment = (value) => {

  return {

    type: INCREMETN,

    payload: value,

  };

};

**1.8 Understanding reducers**

It always takes two parameter **state and action** .it will not change outside data. It works based on the passed data only. It always predicable means truth answer. When we pass 4 and 5 to add. It always gives us 9. Means we know what will be the output. It can have local variable. It will not work with outside variable. It works with it’s local variable and passed variable value. And it will not change anything

Pass state will not directly changed by the reducer. It return new object. Because state is immutable.

Reducer is pure function. Pure function means for same input, it always give us same answer.

**1.9 immutability**

Characteristic of Reducer:

* Pure function ( work based on only input )
* Immutability (do not change main state )

**Primitive data types:** number, single string,

**Non primitive types:** array, object ( they point same location if we copy one to another variable )

If one of them is changed then changed go through both of them. Because of the same reference. Mainly when we copy one to another. There is not value copy. There mainly copy the same reference

**Immutably copy nested object:**

// nexted object

const state = {

  value: 10,

  properties: {

    a: 20,

    b: 30,

  },

};

// copy this object without change the main object without reference

const copiedObject = {

  ...state,

  properties: {

    ...properties,

    a: state.properties.b + 10,

  },

};

Note: above way we have copy the value not reference. So second object and first object are not same location. We have just copied the value.

Note: each level of nested object we need to copy the object and nested object. Inner object if we do not destructure it mainly hold the **state** object reference. So we should destructure nested object also.

**1.10 why reducers is called reducers?**

Reduce means reduce something. Array reducer example below:

const arr = [1, 12, 3, 4, 5, 6];

arr.reduce((previousValue, currentValue) => previousValue + currentValue, 0);

for each element **reduce** called. It takes a function as parameter and second parameter as previous value. This function called each time for each array element. This function take previous answer and current element and calculate something. At last it return the result

**why reducer called reducer ?**

it return updated state or single valu e after reducing actions.

**1.11 Redux Data flow**