Advance React

React Refs

- Refs is the shorthand used for **references** in React.
- It is similar to **keys** in React.
- It is an attribute which makes it possible to store a reference to particular DOM nodes or React elements.
- It provides a way to access React DOM nodes or React elements and how to interact with it.
- It is used when we want to change the value of a child component, without making the use of props.

- Its use should be avoided for anything that can be done **declaratively**.
- For example, instead of using **open()** and **close()** methods on a Dialog component, you need to pass an **isOpen** prop to it.
- You should have to avoid overuse of the Refs.
- Think more critically about where it should be use in component hierarchy.

When to use Refs:

- Managing focus, text selection, or media playback.
- Triggering imperative animations.
- Integrating with third-party DOM libraries.

How to create Refs?

- In React, Refs can be created by using **React.createRef()**.
- It can be assigned to React elements via the **ref** attribute.
- It is commonly assigned to an instance property when a component is created, and then can be referenced throughout the component.

How to access?

- In React, when a ref is passed to an element inside render method, a reference to the node can be accessed via the current attribute of the ref.

How to create?

- The ref value differs depending on the type of the node:
- When the ref attribute is used in HTML element, the ref created with **React.createRef()** receives the underlying DOM element as its **current** property.
- If the ref attribute is used on a custom class component, then ref object receives component as its current property.
- The ref attribute cannot be used on **function components** because they don't have instances.

```
• class MyComponent extends React.Component {
•   constructor(props) {
•     super(props);
•     this.callRef = React.createRef();
•   }
•   render() {
•     return <div ref={this.callRef} />;
•   }
• }
```

How to access Refs

- when a ref is passed to an element inside render method, a reference to the node can be accessed via the current attribute of the ref.
- **const** node = **this**.callRef.current;

Refs current Properties

- The ref value differs depending on the type of the node:
- When the ref attribute is used in HTML element, the ref created with **React.createRef()** receives the underlying DOM element as its **current** property.
- If the ref attribute is used on a custom class component, then ref object receives the **mounted** instance of the component as its current property.
- The ref attribute cannot be used on **function components** because they don't have instances.

```
import React from "react"
export default class App extends React.Component
{
    constructor()
    {
        super();
        this.userRef=React.createRef();
         }
editVal()
{
    console.warn(this.userRef)
    //this.userRef.current.value="1000"
    this.userRef.current.focus();
}
    render()
    {
        return(
        <div>
            <input type="text" name="user" ref={this.userRef}/>
            <input type="button" value="Click Me" onClick={()=>this.editVal()} />
        </div>
   ) }
}
```

- Don’t use Ref in your code frequently.
- In React js we are not updating DOM directly.

Callback Refs

- In react, there is another way to use refs that is called "**callback refs**" and it gives more control when the refs are **set** and **unset**.
- Instead of creating refs by createRef() method, React allows a way to create refs by passing a callback function to the ref attribute of a component
- <input type="text" ref={element => **this**.callRefInput = element} />
- The callback function is used to store a reference to the DOM node in an instance property and can be accessed elsewhere.
- **this**.callRefInput.value

- **import** React, { Component } from 'react';
- **import** { render } from 'react-dom';
- **class** App **extends** React.Component {
-     constructor(props) {
-     **super**(props);
-       **this**.callRefInput = **null**;
-
-     **this**.setInputRef = element => {
-       **this**.callRefInput = element;
-     };
-       **this**.focusRefInput = () => {
-       //Focus the input using the raw DOM API
-       **if** (**this**.callRefInput) **this**.callRefInput.focus();
-     };
-   }

```
componentDidMount() {      //autofocus of the input on mount
    this.focusRefInput();
  }
    render() {
    return (
      <div>
    <h2>Callback Refs Example</h2>
        <input  type="text"  ref={this.setInputRef}  />
        <input  type="button"  value="Focus input text"  onClick={this.focusRefInput}   />
      </div>
    );
  }
}
export default App;
```

Forwarding Ref from one component to another component

- Ref forwarding is a technique that is used for passing a **ref** through a component to one of its child components.
- It can be performed by making use of **React.forwardRef()** method.
- This technique is particularly useful with **higher-order components** and specially used in reusable component libraries.

```
import React, { Component } from 'react';
import { render } from 'react-dom';

const TextInput = React.forwardRef((props, ref) => (
  <input type="text" placeholder="Hello World" ref={ref} />
));

const inputRef = React.createRef();

class CustomTextInput extends React.Component {
  handleSubmit = e => {
    e.preventDefault();
    console.log(inputRef.current.value);
  };
• render() {
•     return (
•       <div>
•         <form onSubmit={e => this.handleSubmit(e)}>
•           <TextInput ref={inputRef} />
•           <button>Submit</button>
•         </form>
•       </div>
•     );
•   }
• }
• export default App;
```

React Hooks

- Hooks are the new feature introduced in the React 16.8 version.
- It allows you to use state and other React features without writing a class.
- Hooks are the functions which "hook into" React state and lifecycle features from function components.
- It does not work inside classes.
- React hooks let you use state and side effect in a functional component something that was not possible before.
- React Hooks let you manage the state and react to state changes in functional component

When to use Hooks?

- If you write a function component, and then you want to add some state to it.
- Now you can do it by using a Hook inside the existing function component.

Rules of Hook

- Hooks are similar to JavaScript functions, but you need to follow these two rules when using them.
- Hooks rule ensures that all the stateful logic in a component is visible in its source code.
- These rules are

1. Only call Hooks at the top level

- Do not call Hooks inside loops, conditions, or nested functions.
- Hooks should always be used at the top level of the React functions.
- This rule ensures that Hooks are called in the same order each time a components renders.

2. Only call Hooks from React functions

- You cannot call Hooks from regular JavaScript functions.
- Instead, you can call Hooks from React function components.
- Hooks can also be called from custom Hooks

Pre-requisites for React Hooks

1. Node version 6 or above
2. NPM version 5.2 or above
3. Create-react-app tool for running the React App

React Hooks Installation

- To use React Hooks, we need to run the following commands:
- $ npm install react@16.8.0-alpha.1 --save
- $ npm install react-dom@16.8.0-alpha.1 --save
- It will install the latest React and React-DOM alpha versions which support React Hooks.
- Make sure the **package.json** file lists the folowing React and React-DOM dependencies
- "react": "^16.8.0-alpha.1",
- "react-dom": "^16.8.0-alpha.1",

Hooks State

- Hook uses useState() functional component for setting and retrieving state.

```
import React, { useState } from 'react';
 class CountApp extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
  }
  render() {
    return (
      <div>
        <p><b>You clicked {this.state.count} times</b></p>
        <button onClick={() => this.setState({ count: this.state.count + 1 })}>
          Click me
        </button>
      </div>
    );
  }
}
export default CountApp;
```

```
import React, { useState } from 'react';

function CountApp() {
  // Declare a new state variable, which we'll call "count"
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
export default CountApp;
```

```
import React,{useState} from 'react'
function App()
{
    const[data,setData]=useState("Sanober")
    return(
        <div>
            <h1>Hi {data}</h1>
        </div>
    )
}
export default App;
```

```
import React,{useState} from 'react'
function App()
{
    const[data,setData]=useState("Sanober")
    return(
        <div>
            <h1>Hi {data}</h1>
            <button onClick={()=>setData("Priya")}>Update Data</button>
        </div>
    )
}
export default App;
```

Hooks Effect

- The Effect Hook allows us to perform side effects (an action) in the function components.
- It does not use components lifecycle methods which are available in class components.
- Effects Hooks are equivalent to componentDidMount(), componentDidUpdate(), and componentWillUnmount() lifecycle methods.
- A functional React component uses props or state to calculate the output.
- If the functional component makes calculations that don’t target the output value, then these calculations are named *side-effects*.
- Examples of side-effects are fetch requests, manipulating DOM directly, using timer functions like setTimeout()

```
import React,{useEffect, useState} from 'react'
function App()
{
    const[count,setCount]=useState(10)
    const[data,setData]=useState(100)
    useEffect(()=>{
        console.warn("called with count state")
    },[count])

    return(
        <div>
        <h1>Count:{count}</h1>
       <h1> Data:{Data}</h1>
        <button onClick={(()=>setCount(count+1))}> Update Count </button>
        <button onClick={(()=>setData(data+1))}> Update Data </button>
        </div>
    )
}
export default App;
```

```
import React,{useEffect, useState} from 'react'
function App()
{
    const[count,setCount]=useState(10)
    const[data,setData]=useState(100)
    useEffect(()=>{
        console.warn("called with count state")
    },[count])

    return(
        <div>
        <h1>Count:{count}</h1>
       <h1> Data:{Data}</h1>
        <button onClick={(()=>setCount(count+1))}> Update Count </button>
        <button onClick={(()=>setData(data+1))}> Update Data </button>
        </div>
    )
}
export default App;
```

```
import React,{useEffect, useState} from 'react'
function App()
{
  const[count,setCount]=useState(10)
  const[data,setData]=useState(100)

useEffect(()=>{
    console.warn("called with count state")
  },[count])
useEffect(()=>{
    console.warn("called with data state")
  },[data])
  return(
    <div>
    <h1>Count:{count}</h1>
         <h1> Data:{Data}</h1>
    <button onClick={(()=>setCount(count+1))}> Update Count </button>
    <button onClick={(()=>setData(data+1))}> Update Data </button>
    </div>
  )
}
export default App;
```

```
import React,{useEffect, useState} from 'react'
function App()
{
 const[count,setCount]=useState(10)
 const[data,setData]=useState(100)
 useEffect(()=>{
 console.warn("called with count state")
 },[data])
 useEffect(()=>{
                alert("Count is" + count)
                },[count])
 return(
 <div>
 <h1>Count:{count}</h1>
                <h1> Data:{data}</h1>
 <button onClick={(()=>setCount(count+1))}> Update Count </button>
 <button onClick={(()=>setData(data+1))}> Update Data </button>
 </div>
)
}
export default App;
```

- useEffect() accepts two arguments
- useEffect(callback[,dependencies])
- callback is the function containing the side-effect logic
- Callback is executed right after changes were being pushed to DOM.
- Dependencies is an optional array of dependencies.
- useEffect() executes callback only if the dependencies have changed between renderings.

With props: User.js

```
import React,{useEffect, useState} from 'react'
function User(props)
{
  useEffect(()=>{
 console.warn("use effect")
 })
 return(
 <div>
 <h1>Count Props:{props.count}</h1>
 <h1> Data Props:{props.data}</h1>

 </div>
)
}
export default User;
```

App.js

```
import React,{useEffect, useState} from 'react'
import User from './User'
function App()
{
 const[count,setCount]=useState(10)
 const[data,setData]=useState(100)

 return(
 <div>

                 <User count={count} data={data} />
 <button onClick={(()=>setCount(count+1))}> Update Count </button>
 <button onClick={(()=>setData(data+1))}> Update Data </button>
 </div>
)
}
export default App;
```

The dependencies of useEffect()

- Dependencies lets you control when the side effect runs.
- When dependencies are:

A) Not provided: the side-effect runs after every rendering.

```
•   import { useEffect } from 'react';
•   function MyComponent() {
•     useEffect(() => {    // Runs after EVERY rendering
•     });
•   }
```

B. An Empty Array[]: the side-effect runs once after the initial rendering.

```
 import { useEffect } from 'react';
function MyComponent() {
  useEffect(() => {
    // Runs ONCE after initial rendering
  }, []);
}
```

C) Has props or state values [prop1, prop2,…state1, state2]:

- the side-effect runs only when any dependency value changes.

```
•  import { useEffect, useState } from 'react';
•  function MyComponent({ prop }) {
•    const [state, setState] = useState('');
•    useEffect(() => {
•      // Runs ONCE after initial rendering
•      // and after every rendering ONLY IF `prop` or `state` changes
•    }, [prop, state]);
•  }
```

3. The side-effect on component did mount

- Use an empty dependencies array to invoke a side-effect once after component mounting:

```
•   import { useEffect } from 'react';
•   function Greet({ name }) {
•     const message = `Hello, ${name}!`;

•     useEffect(() => {
•       // Runs once, after mounting
•       document.title = 'Greetings page';
•     }, []);

•     return <div>{message}</div>;
•   }
```

- useEffect(…,[]) was supplied with an empty array as the dependencies argument. When configured in such a way, the useEffect() executes the callback *just once*, after initial mounting.
- Even if the component re-renders with different name property, the side-effect runs only once after the first render:

```
// First render
<Greet name=“Ritu" />   // Side-effect RUNS

// Second render, name prop changes
<Greet name=“Neeta" />   // Side-effect DOES NOT RUN

// Third render, name prop changes
<Greet name=“Rupali"/> // Side-effect DOES NOT RUN
```

4. The side-effect on component did update

- Each time the side-effect uses props or state values, you must indicate these values as dependencies:

```
•   import { useEffect } from 'react';

•   function MyComponent({ prop }) {
•     const [state, setState] = useState();

•     useEffect(() => {
•       // Side-effect uses `prop` and `state`
•     }, [prop, state]);

•     return <div>....</div>;
•   }
```

- The useEffect(callback, [prop, state]) invokes the callback after the changes are being committed to DOM and if and only if any value in the dependencies array [prop, state] has changed.
- Using the dependencies argument of useEffect() you control when to invoke the side-effect, independently from the rendering cycles of the component.

```
•  import { useEffect } from 'react';

•  function Greet({ name }) {
•    const message = `Hello, ${name}!`;

•    useEffect(() => {
•      document.title = `Greetings to ${name}`;
•    }, [name]);

•    return <div>{message}</div>;
•  }
```

- name prop is mentioned in the dependencies argument of useEffect(…,[name]).
- useEffect() hook runs the side-effect after initial rendering, and on later renderings only if the name value changes.

- // First render
- <Greet name=“Aditya" />   // Side-effect RUNS

- // Second render, name prop changes
- <Greet name=“Sonali" />   // Side-effect RUNS

- // Third render, name prop doesn't change
- <Greet name=“Aditya" />   // Side-effect DOES NOT RUN

- // Fourth render, name prop changes
- <Greet name=“xyz"/> // Side-effect RUNS

5. Data Fetching

- useEffect() can perform data fetching side-effect.

```
• import { useEffect, useState } from 'react';
•  function FetchEmployeesByQuery({ query }) {
• const [employees, setEmployees] = useState([]);
• useEffect(() => {
• async function fetchEmployees() {
•  const response = await fetch( `/employees?q=${encodeURIComponent(query)}` );
• const fetchedEmployees = await response.json(response);
• setEmployees(fetchedEmployees);
•  }
•  fetchEmployees();
•  }, [query]);
•  return (
• <div> {employees.map(name => <div>{name}</div>)} </div> ); }
• function FetchEmployeesByQuery({ query }) {
•   const [employees, setEmployees] = useState([]);

•   useEffect(() => {  // <--- CANNOT be an async function
•     async function fetchEmployees() {
•       // ...
•     }
•     fetchEmployees(); // <--- But CAN invoke async functions
•   }, [query]);

•   // ...
• }
```

The side effect clean up

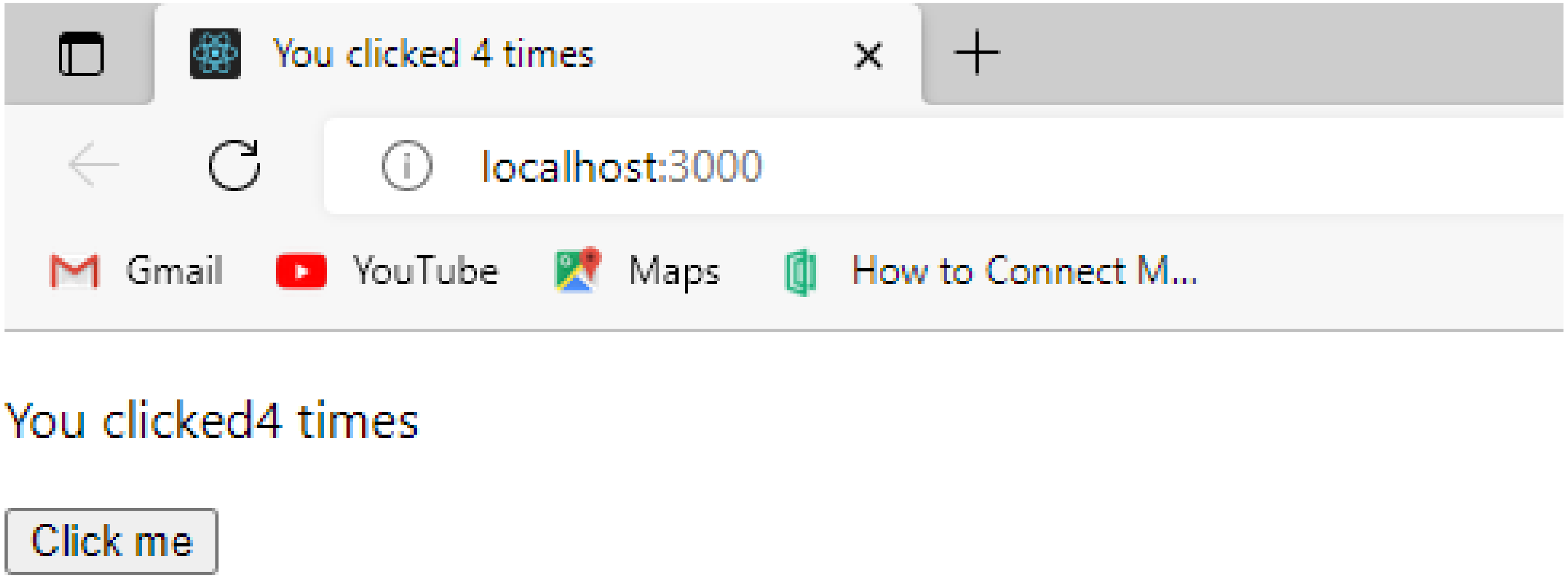
- Some side-effects need cleanup: close a socket, clear timers.
- If the callback of useEffect(callback, deps)return a function then useEffect() considers this as an effect cleanup:
- useEffect(() => {
-   // Side-effect...

-   return function cleanup() {
-     // Side-effect cleanup...
-   };
- }, dependencies);

Custom Hook

- A custom Hook is a JavaScript function.
- The name of custom Hook starts with "use" which can call other Hooks.
- A custom Hook is just like a regular function, and the word "use" in the beginning tells that this function follows the rules of Hooks.
- Building custom Hooks allows you to extract component logic into reusable functions.

```
• import React, { useState, useEffect } from 'react';
•   const useDocumentTitle = title => {
•   useEffect(() => {
•     document.title = title;
•   }, [title])
• }
• function CustomCounter() {
•   const [count, setCount] = useState(0);
•   const incrementCount = () => setCount(count + 1);
•   useDocumentTitle(`You clicked ${count} times`);
•     return (
•     <div>
•       <p>You clicked {count} times</p>
•       <button onClick={incrementCount}>Click me</button>
•     </div>
•   )
• }
• export default CustomCounter;
```



You clicked4 times

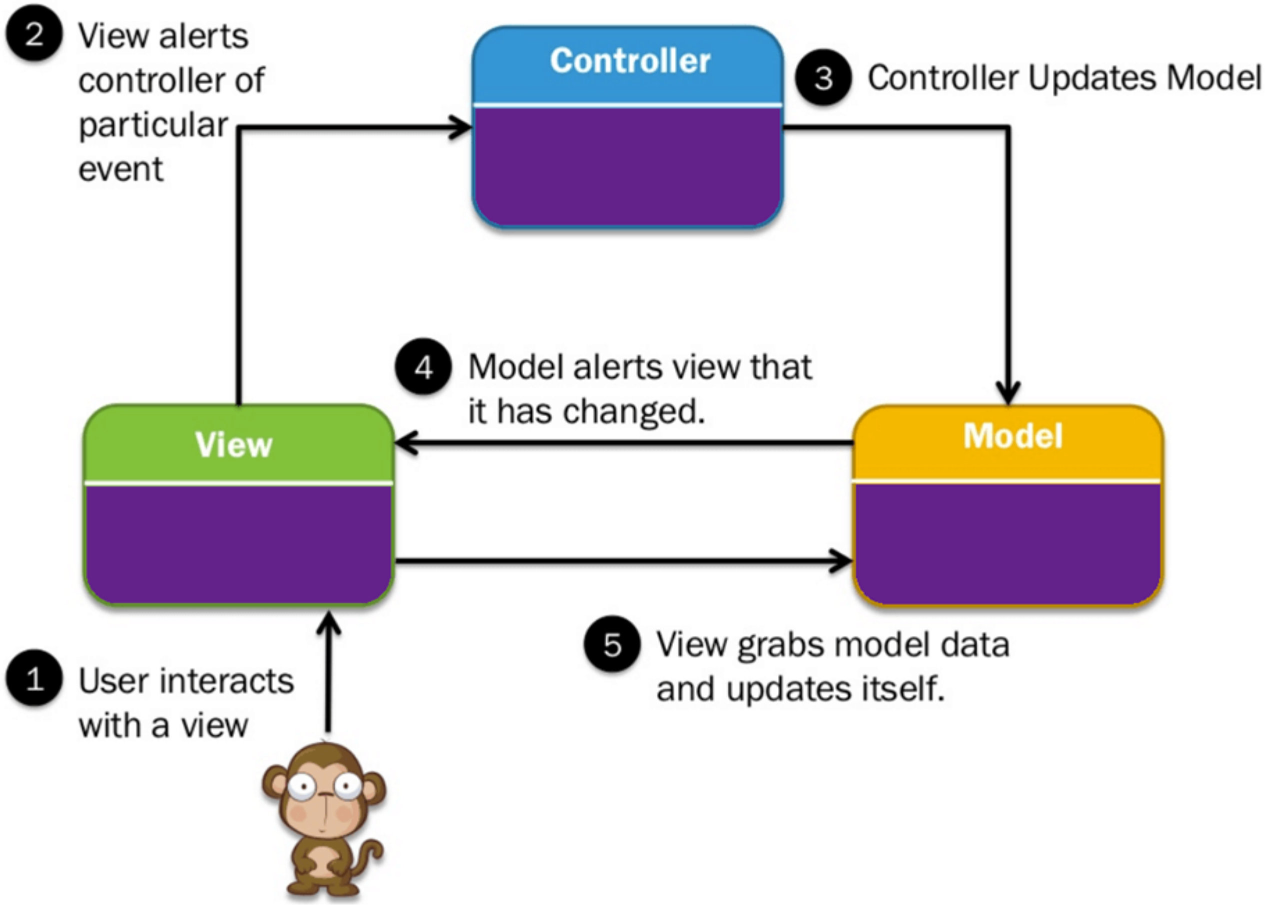
- seDocumentTitle is a custom Hook which takes an argument as a string of text which is a title.
- Inside this Hook, we call useEffect Hook and set the title as long as the title has changed.
- The second argument will perform that check and update the title only when its local state is different than what we are passing in.

- useState
- useEffect
- useContext

- useReducer
- useCallback
- useMemo
- useRef

Model View Controller Framework

- The **Model-View-Controller (MVC)** framework is an architectural pattern that separates an application into three main logical components Model, View, and Controller.
- Each architecture component is built to handle specific development aspect of an application.
- MVC separates the business logic and presentation layer from each other.
- It was traditionally used for desktop graphical user interfaces (GUIs).
- MVC architecture in web technology has become popular for designing web applications as well as mobile apps.



1 User interacts with a view

2 View alerts controller of particular event

3 Controller Updates Model

4 Model alerts view that it has changed.

5 View grabs model data and updates itself.

Model

- The model component stores data and its related logic.
- It represents data that is being transferred between controller components or any other related business logic.
- For example, a Controller object will retrieve the customer info from the database.
- It manipulates data and sends back to the database or uses it to render the same data.
- It responds to the request from the views and also responds to instructions from the controller to update itself.
- It is also the lowest level of the pattern which is responsible for maintaining data.
- It contains no logic how to present the data to a user.

View

- A View is that part of the application that represents the presentation of data.
- Views are created by the data collected from the model data.
- A view requests the model to give information so that it resents the output presentation to the user.
- The view also represents the data from charts, diagrams, and tables.
- For example, any customer view will include all the UI components like text boxes, drop downs, etc.

Controller

- The Controller is that part of the application that handles the user interaction.
- The controller interprets the mouse and keyboard inputs from the user, informing model and the view to change as appropriate.
- A Controller send’s commands to the model to update its state(E.g., Saving a specific document).
- The controller also sends commands to its associated view to change the view’s presentation (For example scrolling a particular document).

Advantages:

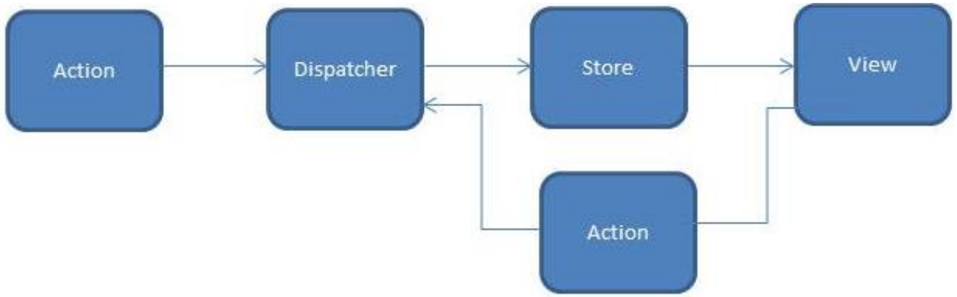
- Easy code maintenance which is easy to extend and grow
- MVC Model component can be tested separately from the user
- Easier support for new types of clients
- Development of the various components can be performed parallelly.
- It helps you to avoid complexity by dividing an application into the three units. Model, view, and controller
- Multiple developers can work simultaneously on the model, controller and view.
- Models can have multiple views.

Disadvantages:

- Difficult to read, change, unit test, and reuse this model
- The framework navigation can some time complex as it introduces new layers of abstraction which requires users to adapt to the decomposition criteria of MVC.
- Knowledge of multiple technologies is required.
- Maintenance of lots of codes in Controller
- Developers using MVC need to be skilled in multiple technologies.

React Flux:

- Flux is a Javascript architecture or pattern for UI which runs on a unidirectional data flow and has a centralized dispatcher.
- It is useful when your project has dynamic data and you need to keep the data updated in an effective manner.
- It was created by Facebook, and complements React as view.
- This model is used to ease maintenance.
- It has three primary components: Views, Stores, and Dispatcher.



- **Action**: Action describes user interaction that occurred in a component, such as a user clicking a button.
- **Dispatcher**: Dispatcher methods are invoked by an action component.
- This emits an event with data that needs to go to a store, which is a singleton registry.
- **Store**: The store listens to certain events from the dispatcher.
- When it listens to an event from the dispatcher, it will then modify its internal data and emit a different event for views.
- **View**: Views are typically a react component.
- Views get data from the store and set up a listener to refresh itself when the store emits any changed events.

- In Flux application, data flows in a single direction(unidirectional).
- This data flow is central to the flux pattern.
- The dispatcher, stores, and views are independent nodes with inputs and outputs.
- The actions are simple objects that contain new data and type property.

Dispatcher:

- It is a central hub for the React Flux application and manages all data flow of your Flux application.
- It is a registry of callbacks into the stores.
- It has no real intelligence of its own, and simply acts as a mechanism for distributing the actions to the stores.
- All stores register itself and provide a callback.
- It is a place which handled all events that modify the store.
- When an action creator provides a new action to the dispatcher, all stores receive that action via the callbacks in the registry.

Dispatcher API has five methods

SN	Methods	Descriptions
1.	register()	It is used to register a store's action handler callback.
2.	unregister()	It is used to unregisters a store's callback.
3.	waitFor()	It is used to wait for the specified callback to run first.
4.	dispatch()	It is used to dispatches an action.
5.	isDispatching()	It is used to checks if the dispatcher is currently dispatching an action.

stores

- It primarily contains the application state and logic.
- It is used for maintaining a particular state within the application, updates themselves in response to an action, and emit the change event to alert the controller view.

views

- It is also called as controller-views.
- It is located at the top of the chain to store the logic to generate actions and receive new data from the store.
- It is a React component listen to change events and receives the data from the stores and re-render the application

Action

- The dispatcher method allows us to trigger a dispatch to the store and include a payload of data, which we call an action.
- It is an action creator or helper methods that pass the data to the dispatcher.

Advantages

- It is a unidirectional data flow model which is easy to understand.
- It is open source and more of a design pattern than a formal framework like MVC architecture.
- The flux application is easier to maintain.
- The flux application parts are decoupled.

Comparison of MVC and Flux:

MVC	Flux
Supports Bi directional data flow.	Supports unidirectional data flow
Data binding is the key	Events or actions are the keys
It is synchronous	It is asynchronous
Controller handles logic	Store handles logic
Hard to debug	Easy to debug because it has common initiating point i.e. dispatcher
Difficult to understand as project size increases	Easy to understand
Difficult to maintain as project scope goes huge	Easy to maintain
Difficult to test the application	Easy to test the application
Scalability is complex	Easily scalable

React Redux

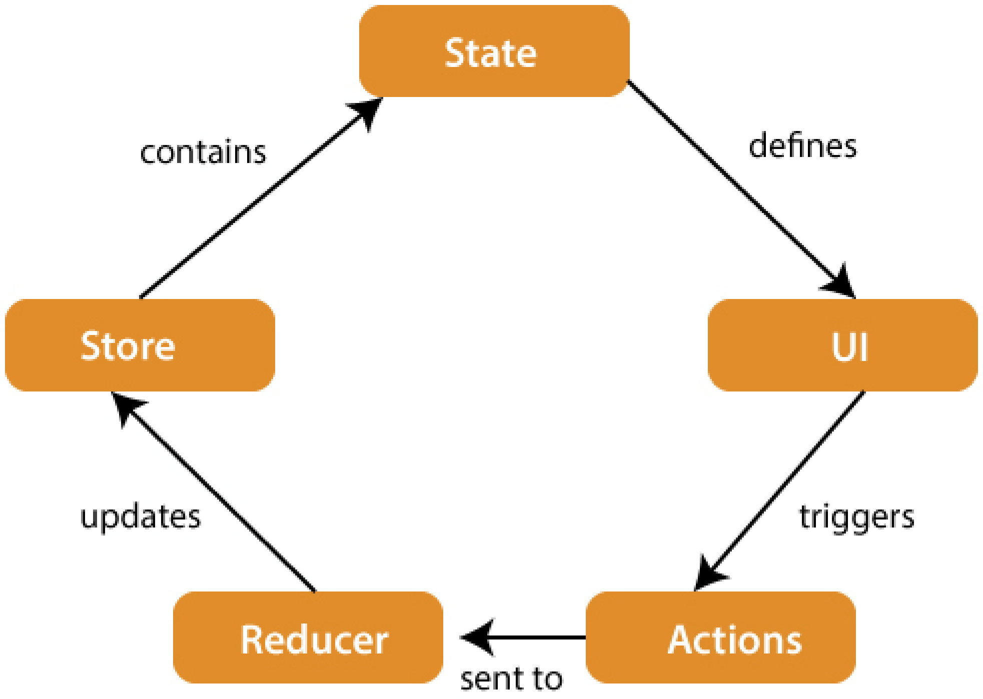
- Redux is an open-source JavaScript library used to manage application state.
- React Redux is the official React binding for Redux.
- It allows React components to read data from a Redux Store, and dispatch **Actions** to the **Store** to update data.
- Redux helps apps to scale by providing a sensible way to manage state through a unidirectional data flow model.
- React Redux is conceptually simple.
- It subscribes to the Redux store, checks to see if the data which your component wants have changed, and re-renders your component.

- Redux was developed by taking inspiration from flux.
- Redux omitted unnecessary complexity of flux architecture as
  - Redux does not have Dispatcher concept.
  - Redux has an only Store whereas Flux has many Stores.
  - The Action objects will be received and handled directly by Store.

Why?

- React Redux is the official **UI bindings** for react Application.
- It is kept up-to-date with any API changes to ensure that your React components behave as expected.
- Using React Redux one can design good ‘React’ architecture.

- React components re-render only when they are actually required.



- **STORE:** A Store is a place where the entire state of your application lists.
- It manages the status of the application and has a dispatch(action) function.
- It is like a brain responsible for all moving parts in Redux.
- **ACTION:** Action is sent or dispatched from the view which are payloads that can be read by Reducers.
- It is a pure object created to store the information of the user's event.
- It includes information such as type of action, time of occurrence, location of occurrence, its coordinates, and which state it aims to change.
- **REDUCER:** Reducer read the payloads from the actions and then updates the store via the state accordingly.
- It is a pure function to return a new state from the initial state.

Installation

- npm install redux react-redux --save