★ React Portals

React Portals provides a first class way to render children into a DOM node i.e. exists outside the DOM hierachy of the parent component. It is mostly useful to creating modals, tooltips, popups, where you need to render componend somewhere else in the DOM node.

```
const Modal = ({message, isOpen, onClose, children}) => {
    if(!isOpen) return null;
    return ReactDOM.createPortal(
        <div className="modal">
            <span>{message}</span>
            <button onClick={onClose}>Close</button>
        </div>,
        document.body);
};

const component = () => {
    const [open, setOpen] = useState(false);
    return (
        <div className="component">
            <button onClick={() => setOpen(true)}>Open</button>
            <Modal message="Hi" isOpen={open}
                onClose={() => setOpen(false)} />
        </div>
```

★ What is babel

- It is transpiller which converts es6 to older JS i.e. compatible with the browser.
- Babel is a toolchain i.e. mainly used to convert ECMAScript 2015+ code into a backwards compatible version of JS.

★ what is useStrict()

- StrictMode is a tool for highlighting potential problems in an action
- This would affect your UI like fragement
- StrictMode only runs in development mode, it wont impact on production build.

✱ React Router DOM

- <BrowserRouter> : An application should have one <BrowserRouter>, which wraps one or more <Routes>

- <Routes> : It check all its children <Route> element to find the best match, and renders that part of the UI.

- <Route> : It is defined as an object or a route element.
  If it is object, it has shape of { path, element }
  If it is route element, it has shape of <Route path element>
  when the path pattern matches the current URL, the 'element' prop is rendered.

- <Link> : This renders an accessible <a> element with a real href that points to the resource it is linking to.

  <Link to='page_one'> Page One </Link>

- <Outlet> : This component renders the active child route. It brings up nested routes, where each routes can have child routes to occupy a portion of the URL. Nested routes typically uses a relative links.

```
<ul>
    <Li> <Link>  </Link> </li>
    — - —
</ul>
<Outlet />
```

- useRoutes() : This is a hook used instead of <Routes>. We can do all the things with useRoutes() which we can do with the <Routes>

```
import { BrowserRouter, useRoutes } from 'react-router-dom';

function App() {
  const routes = useRoutes([
    {
      path: '/',
      element : <MainPage />
      children : [
        { index : true, element : <div> No Page Selected </div> },
        { path : '*', element : <PageOne /> },
        { path : 'two', element : <PageTwo /> }
      ]
    }
  ]);
  return routes;

const AppWrapper = () => {
  return (
      <BrowserRouter>
        <App />
      </BrowserRouter>
  );
}
export default AppWrapper;
```

- useLocation : This is an object which represents the URL location. It is based on the 'windows.location'

```
function App () {

    const location = useLocation ();
    useEffect (() => {
        console.log ("Location : ' + location);
    }, [location]);
```

- useParams : This hooks returns an object of key-value pairs of the dynamic params from the current URL.

```
URL = 'localhost:3000/users/edit/3'

const params = useParams ();
const id = params.id        // contain value 3
```

- useNavigate : This hook returns a function that can be used to navigate programmatically.

```
const navigate = useNavigate ();
<ul>
    <li> <button onclick = (() => navigate ('one', { replace: false }) } >
        Page One
    </button> { ' ' }
    - <Link to = 'one/1' > P1 </Link>
</li>
</ul>
```

- useHref() : It returns a URL that may be used to link to the given 'to' location, even outside of React Router.

- useLinkClickHandler() : It returns a click event handler for navigation when building a custom link.

- useInRouterContext() : It returns 'true' if the component is being rendered in the context of a <Router>, vice versa 'false'.

- useNavigationType() : It returns how the user came to the current page, via 'action.pop', 'action.push', 'action.replace' on the history stack.

- useMatch() : It returns match data about a route at the given path relative to the current location.

- useOutlet() : It return the element of for the child route at this level of the route hierarchy.

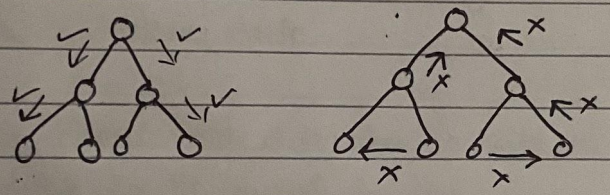
- useResolvedPath() : It resolved the 'pathname' of the location in the given 'to' value against the pathname of the current location.

- useSearchParams() : It is used to read & modify the query string in the URL for the current location.

✱ React - Redux

- Redux is a predictable state container.
- It is a pattern or a library for managing and updating application state.

- React follows the component based approach, where the data always flow through the component.
- It follows uni-directional data flow i.e. data always flow from parent to child component; where child components can never pass data back to the parent component.



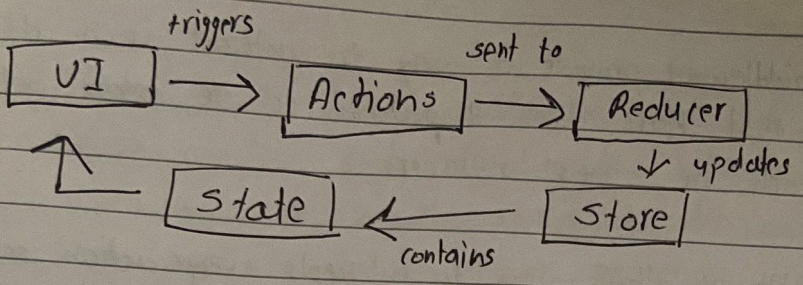
So to pass data between two non parent component redux comes into the pictures.

- Redux ~~offers~~ offers a sol^{n} of storing all your appl^{n} state in one place i.e. 'store'.

- Components then 'dispatch' state changes to the store. And not directly to the other components.

- The components that needs to be update of state changes can 'subscribe' to the 'store'.

★ Redux Flow



- UI : This is where a change is triggered.
  eg. user clicking a 'tic button in a simple counter app

- Actions : Actions are plain JS objects & they must have 'type' property. (eg. ADD_ONE)

- Reducer : Reducer specify how the application's state should change in response to each action.
  eg. A new state should be one int value, higher than our old state.

- store : store brings everything together. It hold application's state.
  methods -
  i) getState() : which allows access to the state object.
  ii) dispatch(action) : which allows state to be updated.
  iii) subscribe(listner) : which registers listners, allowing code to trigger every time a change takes place.

- state : state is contained within the store. It is an object that represent the dynamic parts of the app : anything that may change on client side.

★ What is Redux Thunk

- Redux Thunk is a middleware that let you call action creators that return a fun" instead of object. That fun" receives "the store's dispatch method; which is then used to dispatch regular sychronous action inside the body of the function once the asynchronous operation have completed.

- Redux Thunk is a middleware that allows us to make some action asynchronous.

- Thunk can be used to delay the dispatch of an action, or to dispatch only if a certain condition is met.

```
const yell = (text) => { console.log(text) }
yell('Redux')        // Redux

const thunkedYell=(text) => {
    return function thunk() {
        console.log(text)
    }
}
const thunk = thunkedYell('Redux');      // No action yet
thunk();      // Redux;
```

★ What is Redux-Saga

- Redux saga is a liabrary that aims to make appl^{n} side effects (i.e. asynchronous things like data feetching and impure things like accessing the browser cache), easier to manage, more efficient to execute, easy to test and better at handling failures.

- Redux-saga uses generator instead of normal functions. Because of that we're able to test, write & read asynch calls easily.

- Generators are fun^{n} that can be exited and later re-entred.
- Generator calls does not executes its body immediately; an 'iterator' object for the fun^{n} is return itself.

- When the iterators 'next()' method is called, the generators fun^{n} body is executed until the first 'yield' expression, which specifies the value to be returned from the iterator or with 'yield*' delegates to another generator fun^{n}.

- Redux-saga has two main function

1) A worker function : This will make the API call from 'redux-saga/effect'. after the data loaded we will dispatch another action.
2) A watcher function : It is generator fun^{n} watching for every action that we are executing. In resepoase to that action watcher will call a worker fun^{n}.

```
// watcher saga
export function* watchgetItems() {
    yield takeEvery('GET_REQ_ACTION', getItems);
}

// worker saga
export function* getItems() {
    const items = yield call(api.getItems);
    yield put({
        type: 'GET_ITEMS_SUCCESS_ACTION',
        payload: items
    });
}
```

★ Difference b/w redux-thunk & redux-saga

Thunk	Saga
• Easy to learn / code	• difficult, as compare to thunk, due to generator
• Actual API calls for testing	• Easy Testability (ignoring actual API calls)
• Easy process	• lots of helper fun^n throttling, debouncing, race conditions and cancellation
•	

☆ What is Error Boundries

- Error Boundries helps us to not allow to crash our appl^n
- Error Boundries only available in class component
- In class component we uses couple of methods,

- static getDerivedStateFromError (error) :

  It accepts the error that was thrown as a parameter. This method allows us to handle the error boundries of the appl^n.

- componentDidCatch (error, errorInfo) :

  It will contain 2 arguments to log the error and showing the info which will help to debug code. This method is invoked if some error occurs during the rendering phase.

```
static getDerivedStateFromError ()
{
   return {
          hasError : true ;      //Changing the state if error occurs.
          }
}

componentDidCatch (error, errorInfo)
{
   console.log ("Error Occurred". error, errorInfo );
}
```

- This wont work in Functional compo. For that we can use try/catch.

★ What is SSR

- SSR is a technique used to improve the page load time

- In non-ssr scenario, the react app is served us as a bunch of static files. It follows,
  - Browser sends a req to a URL
  - Browser receives the index.html as a response.
  - Browser then sends requests to download any external link/script.
  - Browser waits till the scripts are downloaded.
  - The react renders the component, and it will mount component.
  - Uuhil the code executes, the user basically just sees a blank screen.

- How SSR works
  - Browser sends a request to the URL.
  - The server responds by rendering the related compo in a string.
  - The rendered component injected into index.html.
  - The `index.html` sent back to the browser.
  - The browser renders the 'index.html' & download all other dependency.
  - Once all the scripts are loaded, the react compo will render.
  - The only difference are, it hydrates the existing view instead of overwriting it.
  - `Hydrating` view means, if it attaches any event handlers to the rendered DOM elements but keeps the rendered DOM elements intact. This way the state of your DOM elements is preserved & there is no view reset that happens.

★ What is webpack

- In early years website was just a small bundle of good old .html, .css and .js files. After some times lots of new framework and liabraries are introduced.
- Website before were no more just a small package with an odd numbers of files in them. They started getting bulky, with the introduction of JS Modules.
- Overall size of the appli" becomes challenge, as well there was a huge gap in the kind of code developers were writing & the kind of code browser could understand.
- Developers had to use a lot of helper code called 'polyfills' to make sure that the browser were able to interpret the code in their packages.

- To answer these issues, webpack is created.
- Webpack is a static module bundler.

☆ How does it work

- Webpack always needs at least one 'entry point', you can have multiple ones as well. Webpack will analyze the dependencies of this file & to another file which then, in turn has more dependencies, so webpack can ~~build~~ build up a dependencies graph.
- The output properties tells webpack where to emit the bundles it creates and how to name those files.

Webpack.config.js

```
const path = require('path');

module.exports = {
    entry : './path/file.js',
    output : {
        path: path.resolve(__dirname, 'dist'),
        filename: 'my-webpack-bundle.js'
    }
};
```

- Between the entry point & output, there are 2 imp steps:

- Loaders : allow webpack to process other types of file & convert them into valid modules. (eg. css-loader, babel-loader)
- Plugins : can be leveraged to perform a wider range of tasks (e.g. bundle optimization, asset management, injection of env variable.

- What is semantic UI

  - It provides a way to compose React components through the as prop.
  - It allows composing component features and props without adding extra nested components.

- what is synthetic event.

  - It is a wrapper of an JS event
  - It has similar interface like browser events, eg. preventDefault()
  - Whenever we are using / triggering an event in a ReactJS, we are not actually trying to trigger the real DOM event, instead we are using the we are using the ReactJS custom event type, which is the synthetic event.

  eg. onClick(), onBlur(), onChange()

★ Pure Componets

The def^n says that for a specific i/p parameters, we always have a specific o/p. The o/p is solely dependent on i/p parameters & no other external variable.

```
const add = (var1, var2) => {
    return var1 + var2;
}
```

★ Memo

If we are using memo it skips rendering a componet if its props have not changed.
It increase the performance.
Mostly used in pure componet.

```
const App = ({name}) => {
    return (
        <>
            <label> {name} </label>
        </>
    );
}
export default memo (App);
```

★ What is ReactJS ?

- It is JS library for building dynamic UI.
- It is an open-source, component based front end library responsible only for the 'view' layer of the application.
- ReactJS works as views using a component based system.

★ What is JSX ?

- It is stand for JavaScript XML.
- It has HTML/XML like syntax
- JSX allows us to put HTML code into JS.

★ What is DOM ?

- Document Object Model is an interface to web pages.
- It allows programs to read & manipulate the page's content, structure & styles.

- It is always valid HTML
- It is ~~always~~ living model that can be modified by JS
- It doesent include pseudo-elements (::after, ::before)
- It includes hidden elements (display:none)

★ What is Virtual DOM ?

- VDOM is light weight tree representation of actual DOM.
- VDOM is only a virtual representation of the DOM.
- Everytime the state of our application changes, the VDOM get updated instead of the real DOM.

When new elements are added to the UI, a VDOM is represent in a tree structure. Each element is a node on this tree. If the state of the elements changes, a new VDOM tree is created. This tree is then compared with the previous VDOM tree. Once this is done, the VDOM calculates the best possible method to make these changes to the real DOM.
↳ This is how virtual DOM is faster.

- VDOM is virtual representation of UI is kept in memory & synched with the 'real' DOM. This process is called 'reconciliation'.

- The process of checking difference between the new VDOM tree & the old VDOM tree is called as 'diffing'.

- "A Shadow DOM" is known as "DOM within DOM"

★ vDOM Vs DOM

- vDOM reduced memory usage.
- vDOM helps to improve performance
- vDOM does not need to rebuild the entire tree.
- In vDOM there is no state less when we rerender.
- It is faster & easier to rerender.

★ What are building blocks of ReactJS ?

- Component :
  It is basically a class / function. It accept i/p and returns react element that define how UI should work.

- state :
  It is collection of data that resides insides your component. State is used to keep track of information between components. states are mutable (changeable).

- Props :
  It is basically properties passed as an arguments in components. Props are how components communicate to each other. Props flows down words from parent to child. Props are immutable. (not changing).

★ What is HOC ?

- Higher Order Component is an advanced technique in React for reusing component logic.
- A HOC is a function that takes a component and returns a new component.

★ Hooks in ReactJS

- Hooks are JS functions. This let you use react features without ~~using~~ writing a long logic.

1. useState() : It allows you to add react state to function comp.

```
const [state, setState] = useState (initialState);
```

2. useEffect(): It allow you to perform side effects in component.

```
useEffect ( () => {
    if(! login)
        redirect ( login-page)
  },[]
);
```

3. useContext() : It is used to send props down through multiple, nested child components

Child compo
↖ const blogDetails = useContext(blogIntoContext);

↙ const blogIntoContext = React.createContext(blogInfo);
Parent Compo

4. useReducer() : This is similar to useState() hook to manage complex state. It is uses the same concept as the reducers in Redux. It needs same numbers of i/p equals to o/p.

Reducers : Suppose reducers is like "coffee maker". The coffee makers needs coffee powder & water, then it returns a freshly brewed cup of coffee.
Based on this reducers are functions that take in the current state (coffee powder) and actions (water) and brew a new state (fresh coffee).

Reducers are pure functions that takes in a state and action and returns a new state.

```
const initialstate = { count : 0 }

function countReducer (state, action) {
    switch (action.type) {
      case 'increment' :
          return { count : state.count + 1 };
      case 'decrement' :
          return { count : state.count - 1 };
      default :
          throw new Error ();
    }
}

function counter () {
    const [state, dispatch] = useReducer ( reducer, initialState);
    return (
      <>
        Count : { state.count }
        <button onClick = {() => dispatch ({ type : "decrement" })}> - </button>
        <button onClick = {() => dispatch ({ type : "increment" })}> + </button>
      </>
    );
}
```

5. useCallback() : This can help us prevent some unnecessary renders and therefore gain a performance boost.
It is similar like useMemo(), but it memorizes functions instead of values.

```
const myCompo = (props) => {
    return 1+1; const someFun =
    const someFun = useCallback(() => { return 1+1; });

useEffect(() => {
        someFun();
}, [someFun]);
});
```

6. useMemo() : It is used to memorized the value that only changes if one of the dependency has changed.

```
const memoRizedValue = useCallback(value, [dependencies])

function memoFun () {
    const [count, setCount] = useState(100);
    const returnVal = (ipVal) => return ipVal + 10;
    var calculatedVal = useMemo(() => returnVal(10), [10]);

    setTimeOut(() => { setCount(count + 1) }, 1000);
}
```

It persist the value while rendering.

7. useRef() : It returns a mutable reference object whose current property is initialized to the passed argument.

current value ↓ initial value ↓

```
const refObj = useRef(initialValue)

const handler = () => {
    const refVal = refObj.current;    //accessing the ref val

    refObj.current = refObj.current + 1;    // updating the val
}
```

8. Custom Hook

★ Lifecycle Methods of react

constructor()
~~static getDerivedStatesFromProps()~~
componentDidMount()
componentDidUpdate()
componentWillUnmount

This all will be handled in functional components by useEffect()

★ Controlled & Uncontrolled Component

Controlled : Here form data is handled by react component. your data (state) & UI (i/p) are always in synch. We can handled i/p validation, dypamic i/p using it.
eg. Disable submit button until all fields are valid.

UnControlled : Here form data is handled by the DOM itself. We use 'Refs' here. Refs provided a way to access DOM nodes or react element created in the render method. Basically you 'pull' the value from the field when you need it.
This can happen when the form is submitted.

// Controlled

```
const app = () => {
  const [ipVal, setIpVal] = useState("");
  const handleChange = (e) => setIpVal(e.target.value)
  const handleSubmit = () => alert(ipVal);

  return (
      <div className="app">
          <input type="text" value={ipVal} onChange={handleChange} />
          <input type="submit" value="Submit" onClick={handleSubmit} />
      </div>
  );
}
```

// Uncontrolled

```
const app = () => {
  const ipRef = useRef(null);
  const handleSubmit = () => { alert(ipRef.current.value); }

  return (
      <div className="app">
          <input value={} ref={ipRef} />
          <input type="submit" value="submit" onClick={handleSubmit} />
      </div>
  );
}
```

★ Passing data from child to parents

Here you stored the property in the parent component & pass down a function that allows the child to update the prop stored in the parent component.

```
const Child = ({setCount}) => {
    return (
        <div>
            <p> Child</p>
            <button onClick={() => setCount(1)}>1</button>
        </div>
    );
}

const Parent = () => {
    const [count, setCount] = useState();
    return (
        <div>
            <p> Parent </p>
            <p> Count : {count} </p>

            <Child setCount={setCount} />
    );
}
```