**React Docs**

**DOM vs Virtual DOM:**

DOM or Document object model is a tree like representation of the html document that browser keeps, the DOM represents the document as nodes and objects, that way programming languages can interact with the page.

V-DOM or Virtual DOM is a light weight representation of the original DOM which react keeps in memory and in-sync with the DOM.

Now making changes directly to the DOM is very expensive and slower than making changes to the virtual DOM as virtual DOM doesn’t has the power to directly change what’s on the screen hence nothing gets drawn on screen.

When anything new is added to the application, a virtual DOM is created and it is represented as a tree. Each element in the application is a node in this tree. So, whenever there is a change in the state of any element, a new Virtual DOM tree is created. This new Virtual DOM tree is then compared with the previous Virtual DOM tree and make a note of the changes. After this, it finds the best possible ways to make these changes to the real DOM. Now only the updated elements will get rendered on the page again.

**React Fiber:**

Fiber is the new reconciliation engine in React16. Its main goal is to enable incremental rendering of the virtual DOM.

The entire process of transforming changes to the real DOM is called Reconciliation.

**Events in React:**

React supported events are:

* Clipboard events (onCopy, onCut, onPaste)
* Composition events
* Keyboard events (onKeyDown onKeyPress, onKeyUp)
* Focus events( onFocus, onBlur)
* Form events (onChange, onInput, onInvalid, onReset, onSubmit)
* Generic events (onError, onLoad)
* Mouse events ( onClick, onDoubleClick, onDrag, onMouseEnter, …)
* Pointer events (onPointerDown, onPointerMove, ….)
* Selection events (onSelect)
* Touch events (onTouchCancel, onTouchEnd, onTouchMove, onTouchStart)
* UI events (onScroll)
* Wheel events (onWheel)
* Media events (onLoad, onError)
* Animation events (onAnimationStart, onAnimationEnd, onAnimationIteration)
* Other events (onToggle)

**Stateful components:**

In React, a stateful component is a component that holds some state. Stateless component, by contrast, have no state. Note that both types of components can use props.

**Binding in react event handlers:**

**ReactJS bind() method:**

The bind() is an inbuilt method in React that is used to pass the data as an argument to the function of a class based component.

**Syntax:** onClick = this.function.bind(this, [arg1….]);

**Parameter:** It accepts two parameters, the first parameter is the this keyword used for binding and the second parameter is the sequence of arguments that are passed as a parameter and are optional.

**Arrow method binding:**

Arrow methods were introduced ES6 and they can be used too to pass arguments to a handler event in react like such.

**Syntax:** onClick = () => {function(arg1...)}

**prop Types:**

As React is a javaScript library and javaScript is a dynamically typed language we use extensions like typeScript to get that sweet type-safety and catch bugs early on in the development phase, but even if we don’t use it React has a built in tool known as propTypes with which we can easily define the type of props that a component will be receiving.

**Working of react/ imp features:**

**Features of ReactJS:**

i) JSX: JavaScript Syntax Extension is a combination of HTML and JS. You can embed javaScript objects inside the HTML elements. JSX is not supported by the browsers, as a result Babel compiler transcompile the code into JS code. JSX makes codes easy and understandable. It is easy to learn if you know HTML and JS.

ii)Virtual DOM: DOM stands for Document Object Model. It is the most imp part of the webapp as it divides into modules and executes the code. Usually, JS Frameworks updated the whole DOM at once, which made the web applications slow. But react uses virtual DOM which is an exact copy of the real DOM. Whenever there is a modification in the web application, the whole virtual DOM is updated first and finds the difference between real DOM and Virtual DOM. Once it finds the difference, then DOM updates only the part that has changed recently and everything remains the same and this process is known as reconciliation.

iii) One-way Data Binding: One-way data binding, the name itself says that its a one-directional flow. The data in react flows only in one direction i.e. the data is transferred from top to bottom i.e. from parent components to child components. The properties(props) in the child component cannot return the data to its parent component but it can have communication with the parent components to modify the states according to the provided inputs. This is the working process off one-way data binding. This keeps everything modular and fast.

iv)Performance: As React uses VDOM and updates only the modified parts. So, this makes the DOM to run faster. DOM executes in memory so we can create separate components which makes the DOM run faster.

v)Extension: React has many extensions that we can use to create full-fledged UI applications. It supports mobile app development and provide server-side rendering. React is extended with Flux. Redux, React Native, etc. which helps us to create good-looking UI.

vi) Conditional Statements: JSX allows us to write conditional statements. The data in the browser is displayed according to the conditions provided inside the JSX.

vii) Components: ReactJS divides the web page into multiple components as it is components-based. Each component is a part of the UI design which has its own logic and design. So the component logic which is written in JS makes it easy and run faster and can be reusable.

viii)Simplicity: React is component-based which makes the code reusable and React.js uses JSX which is a combination of HTML and JS and makes the code **declarative**. This makes code easy to understand and easy to debug and has less code.

**JSX:**

JSX, is a syntax extension/ syntactic sugar to JS. We recommend using it with React to describe what the UI should look like. JSX produces React “elements”.

React doesn’t require JSX, but most people find it helpful as a visual aid when working with UI inside the JavaScript code.

**Can browsers read JSX:**

No, Browsers can’t read JSX because there is no inherent implementation for the browser engines to read and understand them. JSX is not intended to be implemented by the engines or browsers.

**Features, advantages and Limitations of React:**

**Features/ Advantages:**

Easy to learn and use.

Fast.

3rd party community library and packages.

React supports handy dev tools extension for firefox and chromium based browsers.

Reusable components.

**Limitations:**

As React is more of a library rather than a framework it comes with a lesser number of development tools out of the box.

No limitations or set of protocols to develop a webapp sometimes can be seen as an advantage but sometimes seems out to be a limitation as many people will have many different solution for the same problem and handing off in this setting becomes problematic.

**How rendering works in React:**

Uses of **Virtual DOM,**

**Render:** Rendering is a process that is triggered by a change of state in some component of your application when a state change occurs in React.

**Reconciliation:** Once the re-rendering has happened, React has the context of two versions of the <Component/> output, the version executed before the state change occurred, and the version executed after the state has changed.

At this point two objects are describing the UI, React through heuristic algorithm will be able to determine which elements need to be represented again.

**Commit:** After React calculated all the changes that should be made in the application tree.

**States vs Props:**

The Sate represents parts of an application that can change. Each component can have its state. The state is mutable and it is local to the component only.

Props also known as properties, they are used to pass data from one component to another. Props cannot be modified, read-only, and immutable.

**Higher order components(HOC’s) need/use:**

A higher-order component is a function that takes a component and returns a new component.

Whereas a component transforms props into UI, a higher-order component transforms a component into another component.

Higher-order components or HOC is the advanced method of reusing the component functionality logic. It simply takes the original component and returns the enhanced component.

**Reason to use Higher-Order component:**

Easy to handle

Get rid of copying the same logic in every component

Makes code more readable

**3 phases of a components life cycle:**

The three phases are: **Mounting, Updating** and **Unmounting.**

**Mounting:**

Mounting means putting elements into the DOM.

React has 4 built in methods that gets called, in this order, when mounting a component:

i)constructor()

ii)getDerivedStateFromProps()

iii)render()

iv)componentDidMount()

The render() method is required and will always be called, the others are optional and will be called if you define them.

**Updating:**

The next phase in the lifecycle is when a component is updated.

A component is updated whenever there is a change in the component’s **state** or **props.**

React has 4 built-in methods that gets called, in this order, when a component is updated.

i)getDerivedStateFromProps()

ii)shouldComponentUpdate()

iii)render()

iv)componentDidUpdate()

The **render()** method is required and will always be called, the others are optional and will be called if you define them.

**Unmounting:**

The next phase in the lifecycle is when a component is removed from the DOM, or *unmounting* as React likes to call it.

React has only one built-in method that gets called when a component is unmounted:

componentWillUnmount()

This method gets called when the component is about to be removed from the DOM.

**Synthetic events:**

In order to work as a cross-browser application, React has created a wrapper same as the native browser in order to avoid creating multiple implementations for multiple methods for multiple browsers, creating common names for all events across browsers. Another benefit is that it increases the performance of the application as React reuses the event object.

It pools the event already done hence improving the performance.

**Syntax:**

e.preventDefault() prevents all the default behavior by the browser.

e.stopPropagation() prevents the call to the parent component whenever a child component gets called.

**refs in React:**

When you want a component to “remember” some information, but you don’t want that information to *trigger new renders,* you can use a ref.

Refs can also be used to manipulate the DOM. As in *some\_ref.current.focus()*

**Purpose of render():**

React renders HTML to the web page by using a function called render(). The purpose of the function is to display the specified HTML code inside the specified HTML element. In the render() method, we can read props and state and return our JSX code to the root component of our app.

**Controlled v/s Uncontrolled components: \*\***

**Controlled components:** In React, controlled components are those in which form’s data is handled by the component’s state. It takes its current value through props and makes changes through callbacks like onClick, onChange, etc. A parent component manages its own state and passes the new values as props to the controlled component

**Uncontrolled components:** Uncontrolled components are the components that are not controlled by the React state and are handled by the DOM. So in order to access any value that has been entered we take the help of refs.

For instance, if we want to add a file as an input, this cannot be controlled as this depends on the browser so this is an example of an uncontrolled input.

**Pure Components:**

A React component is considered pure if it renders the same output for the same state and props. For this type of class component, React provides the PureComponent base class. Class components that extend the React.PureComponent class are treated as pure components.

Pure components have some performance improvements and render optimizations since React implements the shouldComponentUpdate() method for them with a shallow comparison for props and state.

Functional components are not pure components.

**Keys?**

A “key” is a special string attribute you need to include when creating lists of elements in React. Keys are used in React to identify which items in the list are changed, updated, or deleted. In other words, we can say that keys are used to give an identity to the elements in the lists.

**React-Router?**

Create-react-app doesn’t include page routing. React-router is the most popular solution.

<BrowserRouter>

<Routes>

<Route path="/" element={<Layout />}>

<Route index element={<Home />} />

<Route path="blogs" element={<Blogs />} />

<Route path="contact" element={<Contact />} />

<Route path="\*" element={<NoPage />} />

</Routes>

</BrowserRouter>

We wrap our content first with **<BrowserRouter>**.

Then we define our **<Routes>.** An application can have multiple <Routes>. Our basic application uses only one. **<Route>**s can be nested. The first <Route> has a path of / and renders the **Layout** component.

Provides different components like **<Link to=”...*some\_path*”/>** which should be used as a replacement for an <href> tag

**Use of <Switch /> in react-router-dom ?**

<Switch/> is deprecated in react-router-dom v6 we use <Routes/> now instead but <Switch /> was used to route to pages exclusively, earlier different <Route/> were used to render one by one and all the routes were used to render relatively which tended to introduce some bugs.

**React-router-dom Hooks?**

**useHistory():** not available in v6 only until v5 gives access to the history instance that you may use to navigate.

**useNavigation():** returns a function that lets you navigate programmatically, for example in an effect or event.

**useParams():** The useParams hook returns an object of key/value pairs of the dynamic params from the current URL that were matched by the <Route path>. Child routes inherit all params from their parent routes.

Ex:

let { userId } = useParams();

<Route path=”:userId” element={ <ProfilePage />} />

**useLocation():** The useLocation hook returns the location object with which its easier to do some side-effection if the location changes

**How is routing in react diff from conventional**

**routing:**

React Router is a library for React that provides routing functionality. It is different from conventional routing in a few ways.

**First,** React Router is declarative. This means that you specify what you want your route to look like, rather than specifying how to get there.

**Second,** React Router is modular. This means that you can use only the features you need, rather than having to include everything in the library. This makes it lightweight and efficient.

**Third,** React Router is asynchronous. This means that routes can be loaded on-demand, rather than all at once. This makes the application more responsive and efficient.

**Fourth,** React Router is composable. This means that you can create complex routes by combining multiple routes together. This makes the routing process more flexible.

**Why is Router required in React**

It is mainly used to create SPA since it retains the application’s regular structure and functionality. The Router in React JS is primarily used to create Single Page Web Apps. In the application, React Router is utilized to define various routes.

**GFG:** React Router is a standard library for routing in React. It enables navigation among views of various components in a React Application, allows changing the browser URL, and keeps the UI in sync with the URL.

**How can you tell react to build in prod mode:**

Go to the root directory of the project and run:

**npm run build** command.

**Diff b/w clone element & create Element:**

**React.createElement**(): The method is used to create elements. Whenever we write code in JSX, JSX converts it to React.createElement(). This createElement method is not recommended to use as it is very hard to maintain or debug. We’ve to call the React.createElement() method every time for the creation of a React element, even if it is just a span tag with no attributes.

**React.cloneElement**(): The method is used when a parent component wants to add or modify the props of its children. The React.cloneElement() function creates a clone of a given element, and we can also pass props and children into the function.

**Strict mode component:**

**<React.StrictMode />** is sort of a helper component that will help you write better React components, you can wrap a set of components with <React.StrictMode /> and it’ll basically:

i) Verify that the components inside are following some of the recommended practices and warn you if not in the console.

ii) Verify the deprecated methods are not being used, and if they’re used strict mode will warn you in the console.

iii) Help you prevent some side effects by identifying potential risks.

As the documentation says, strict mode is development oriented, so you don’t need to worry about it impacting on your production build.

I’ve found it especially useful to implement strict mode when I’m working on new code bases and I want to see what kind of code/components I’m facing. Also, if you’re on bug hunting mode, sometimes it’s a good idea to wrap with <StrictMode /*> the components* blocks of code you think might be the source of the problem.

**Hooks**

Points to remember when using hooks:

i)You can only use hooks inside functional components or custom hooks.

ii)React hooks must be called in the same exact order in every component render so they can’t be called conditionally. Cannot be nested in anything.

**Must know hooks:**

**i)useState:**

So this is the most important React hook. useState hook takes the initial value for any state and returns an object containing the state value and a function which lets you update your state. When the state is updated the component performs a re-render.

**ii)useEffect:**

The useEffect Hook allows you to perform side effects in your components. Some example of side effects are: fetching data, directly updating the DOM, and timers.

useEffect accepts two arguments. The second argument is optional.

useEffect(<function>, <dependency\_array>)

If no dependency array is provided the effect runs on every render. If empty array is provided the effect will only run on initial render and if variables are provided inside the dependency array the useEffect will only run when the values of the variables in the dependency array changes.

Some effects require cleanup to reduce memory leaks. Timeouts, subscriptions, event listeners, and other effects that are no longer needed should be disposed. We do this by including a return function at the end of the useEffect Hook.

Syntax:

useEffect(() => {

let timer = setTimeout(()=>{

setCount((count) => count + 1);

}, 1000)

return () => clearTimout(timer) // *Cleanup called*

}, [ ]); //*empty dependency array means effect will only run on initial render*

**iii)useContext:**

Context is a method to avoid prop-drilling, by storing the props in a store and using these props from the store by child components without actually passing them manually at each level of the component tree.

In order to use the context in a child component, we need to access it using the useContext hook.

**iv)useRef:**

A ref is an object with a single **.current** property which is set to the current value of the ref. What makes refs so powerful is the fact that they are persisted between renders. So refs can be very similar to state, since they persist between renders, but refs do not cause a component to re-render when changed.

The most common use case for refs in React is to reference a DOM element. Because of how common this use case is every DOM element has a ref property you can use for setting a ref to that element.

**v)useMemo:**

useMemo hook let’s you memoise/ cache a value so that you don’t have to calculate it every time.

Syntax:

const doubleNumber = useMemo(()=>{

return slowFunction(number)

}, [number])

this will only run the slowFunction when the variable ‘number’ changes.

Second, use case of the useMemo is referential equality so whenever you have to make sure the reference of an object or an array is exactly the same as it was the last time you rendered.

Syntax:

const themeStyles = useMemo(()=>{

return {

backgroundColor: dark? ‘black’ : ‘white’,

color: dark? ‘white’ : ‘black’

}

}, [dark])

**vi)useCallback:**

useCallback hook is very similar to the useMemo hook but the main difference is useCallback is used specifically for caching functions instead of caching values. So whenever the component re-renders no new functions should be created.

**vii)useReducer:**

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

useReducer is a React Hook that lets you add a reducer to your component.

useReducer is very similar to useState, but it lets you move the state update logic from event handlers into a single function outside of your component. useState and useReducer are basically equivalent it mostly boils down to personal preference but reducer functions shine while debugging.

useReducer returns an array with exactly two items:

i)The *current\_state* of this state variable, initially set to the *initial state* you provided.

ii)The dispatch function that lets you change it in response to interaction.

Syntax:

function reducer (state, action){

switch(action.type){

case ‘incremented\_age’:

return { age: state.age + 1}

}

//….

}

const [state, dispatch] = useReducer(reducer, { age: 42 } );

function handleClick(){

dispatch({type: ‘incremented\_age’});

}

**Redux? Principles? Components of Redux?**

**Advantages?**

**Redux** is a predictable state container for JS applications. It helps you write apps that behave consistently, run in different environments (client, server, and native), and are easy to test.

1. Redux is a state management tool.
2. Redux can be used with any JS framework or library.
3. Redux stores the state of the application, and the components can access the state from a state store anytime.

**Principles of Redux:**

1. **Single source of truth:** The state of your whole application is stored in an object tree with a single store.
2. **State is read-only:** The only way to change the state is to emit an action, an object describing what happened.
3. **Changes are made with pure functions:** To specify how actions transform the state tree, you write pure **reducers.**

**Pillars of redux:**

1. **Store:** A store is an object that holds the application’s state tree. There should only be a single store in a Redux app, as the composition happens at the reducer level.
2. **Action:** An action is a plain object that represents an intention to change the data.
3. **Reducers:** Reducers specify how the applications state changes in response to actions sent to the store.

**Pros of Redux:**

There is always one source of truth, the store, with no confusion about how to sync the current state with actions and other parts of the application.

Having a predictable outcome and strict structure makes the code easier to maintain.

Redux is more stringent about how code should be organized, which makes code more consistent.

Very useful, especially for the initial render, making for better user experience or search engine optimization.

Developers can track everything going on in the app in real-time, from actions to state changes.

**Cons of Redux:**

No Encapsulation. Any component can access data that can cause security issues.

Boilerplate code.

Memory intensive.

**Imports:**

-> npm install redux react-redux @reduxjs/toolkit

**Some syntax:**

**Creating Store:**

import { configureStore } from “@reduxjs/toolkit”

import { Provider } from ‘react-redux’ // Provider is wrapped around <App/>

import userReducer from ‘…’

const store = configureStore({

reducer: {

user: userReducer

}

})

//..*your store should contain a collection of reducers (all of your reducers) that might be used in your application.*

<Provider store={store}>

<App />

<Provider />

**Creating slices/ reducers:**

import { createSlice } from “reduxjs/toolkit”

const initialStateValue = { name: ‘’, age: 0, email: ‘’ };

export const userSlice = createSlice({

name: “user”,

initialState: { value: initialStateValue },

reducers: {

login: (state, action) => {

state.value = action.payload

}

logout: (state) => {

state.value = initialStateValue

}

}

})

export const {login, logout} = userSlice.actions;

export default userSlice.reducer

import { useSelector } from “react-redux” //..for accessing states

Import { useDispatch } from “react-redux” //… for changing/ updating states

**Reading and updating states:**

const user = useSelector ((state) => state.user.value)

const dispatch = useDispatch();

dispatch( login( { name: “Shubham”, age: 22, email: “email.gmail.com” } ) );

dispatch( logout( ) );