Sr. Software Engineer (SSE)

ABSTRACT

This is one of the subject from my personal notes series named “Coding-With-Arqam” that I am developing from the start of my professional development career.

Subject

React JS

Portfolio: https://arqam-dev.github.io/

REACT JS

--> Commands:

-> npm init

-> create-react-app my-app / npx create-react-app my-app

-> cd my-app

-> npm start

-> npm i react-router

-> npm i react-router-dom (BrowserRouter, Route, Link ,Switch, etc)

--> React Environment Setup:

-> Pre-requisite for ReactJS:

-> NodeJS and NPM

-> React and React DOM

-> Webpack

-> Babel = transpiler

-> Ways to install ReactJS:

-> Using the npm command:

-> npm init -y

-> npm install react react-dom --save

-> npm install react --save

-> npm install react-dom --save

-> npm install webpack --save

-> npm install webpack-dev-server --save

-> npm install webpack-cli --save

-> npm install babel-core --save-dev

-> npm install babel-loader --save-dev

-> npm install babel-preset-env --save-dev

-> npm install babel-preset-react --save-dev

-> npm install babel-webpack-plugin --save-dev

-> touch index.html

-> touch App.js

-> touch main.js

-> touch webpack.config.js

-> touch .babelrc

-> Follow the link for further details: https://www.javatpoint.com/react-installation

-> Using the create-react-app command:

-> npm install -g create-react-app

-> create-react-app jtp-reactapp

-> cd my-app

-> npm start

--> Notes:

-> React is a front-end library developed by Facebook.

-> It is used for handling the view layer for web and mobile appsReactJS allows us to create reusable UI components.

-> It is currently one of the most popular JavaScript libraries and has a strong foundation and large community behind it.

-> Lots of people use React as the V in MVC.

-> React update the only item of the page although we render the whole page in render function.

-> JS is too fast but DOM is too slow.

-> React creates new virtual dom when we update anything, and then it compares with old virtual dom and update that specific part of that specific page only.

-> The state, the source of truth that drives our app.

-> The view, a declarative description of the UI based on the current state.

-> The actions, the events that occur in the app based on user input, and trigger updates in the state.

-> ReactJS is a declarative, efficient, and flexible JavaScript library for building reusable UI components.

-> It is an open-source, component-based front end library which is responsible only for the view layer of the application.

-> It was initially developed and maintained by Facebook and later used in its products like WhatsApp & Instagram.

-> It is the V(view part) in the MVC (Model-View-Controller) model.

--> Best Practices:

-> Keep components small and function-specific

-> Reusability is important, so keep creation of new components to the minimum required

-> Consolidate duplicate code – DRY your code

-> Put CSS in JavaScript

-> Comment only where necessary

-> Name the component after the function

-> Use capitals for component names (Title Case)

-> Mind the other naming conventions

-> Separate stateful aspects from rendering

-> All files related to any one component should be in a single folder

-> Use tools like Bit

-> Use snippet libraries

-> Write tests for all code

-> Follow linting rules, break up lines that are too long

--> Why we use ReactJS?

-> The main objective of ReactJS is to develop User Interfaces (UI) that improves the speed of the apps.

-> It uses virtual DOM (JavaScript object), which improves the performance of the app.

-> The JavaScript virtual DOM is faster than the regular DOM.

-> We can use ReactJS on the client and server-side as well as with other frameworks.

-> It uses component and data patterns that improve readability and helps to maintain larger apps.

--> React Features:

-> JSX:

-> JSX is JavaScript syntax extension.

-> Its an XML or HTML like syntax used by ReactJS.

-> It extends the ES6 so that HTML like text can co-exist with JavaScript react code.

-> It isn't necessary to use JSX in React development, but it is recommended.

-> It is faster because it performs optimization while compiling code to JavaScript.

-> It is also type-safe and most of the errors can be caught during compilation.

-> It makes it easier and faster to write templates, if you are familiar with HTML.

-> JSX is a React extension which allows writing JavaScript code that looks like HTML.

-> JSX is an HTML-like syntax used by React that extends ECMAScript so that HTML-like syntax can co-exist with JavaScript/React code.

-> Advantages:

-> It is faster than regular JavaScript because it performs optimization while translating the code to JavaScript.

-> It is type-safe, and most of the errors can be found at compilation time.

-> It makes easier to create templates.

-> Styling:

-> <h1 style = {myStyle}>www.javatpoint.com</h1> // where myStyle is the object defined above(in render function and outside return).

-> Components:

-> React is all about components.

-> You need to think of everything as a component. This will help you maintain the code when working on larger scale projects.

-> Unidirectional data flow and Flux:

-> React implements one-way data flow which makes it easy to reason about your app.

-> Flux is a pattern / architecture that helps keeping your data unidirectional.

-> The benefits of one-way data binding give you better control throughout the application.

-> Virtual DOM.

-> Simplicity

-> Performance

--> React Pros and Cons:

-> Advantages:

-> Performance Enhancement: Uses virtual DOM which is a JavaScript object. This will improve apps performance, since JavaScript virtual DOM is faster than the regular DOM.

-> Can be used on client and server side as well as with other frameworks.

-> Component and data patterns improve readability, which helps to maintain larger apps.

-> Easy to Learn and Use.

-> Creating Dynamic Web Applications Becomes Easier.

-> Reusable Components.

-> Known to be SEO Friendly.

-> Scope for Testing the Codes.

--> React Limitations:

-> Covers only the view layer of the app, hence you still need to choose other technologies to get a complete tooling set for development.

-> Uses inline templating and JSX, which might seem awkward to some developers.

--> JS VS JSX:

-> JS is standard javascript

-> JSX is an HTML-like syntax that you can use with React to (theoretically) make it easier and more intuitive to create React components

-> Without JSX, creating large, nested HTML documents using JS syntax would be a large pain in the rear

-> JSX code by itself cannot be read by the browser; it must be transpiled into traditional JavaScript using tools like Babel and webpack.

--> DOM:

-> General DOM:

-> DOM is an object which is created by the browser each time a web page is loaded.

-> It dynamically adds or removes the data at the back end and when any modifications were done, then each time a new DOM is created for the same page.

-> This repeated creation of DOM makes unnecessary memory wastage and reduces the performance of the application (REACT removes it through virtual DOM).

-> Virtual DOM(React uses):

-> ReactJS still used the same traditional data flow, but it is not directly operating on the browser's Document Object Model (DOM) immediately; instead, it operates on a virtual DOM

-> An in-memory representation of Real DOM.

-> The React Virtual DOM exists entirely in memory and is a representation of the web browser's DOM.

-> The representation of a UI is kept in memory and synced with the "real" DOM.

-> It's a step that happens between the render function being called and the displaying of elements on the screen.

-> Render() => Virtual Dom => Display on UI.

-> This entire process is called reconciliation.

--> Directory Structure:

-> Components Folder:

-> Common things for re-use like buttons etc.

-> Views Folder:

-> Screens using components.

-> Layout Folder:

-> Common views (Header + SideBar + Footer + etc)

--> Components:

-> Class:

-> Stateful component.

-> Class components are more complex than functional components.

-> It requires you to extend from React.

->

-> when we need to use forms.

-> Fucntional:

-> Stateless components.

-> A way to write components that only contain a render method and don't have their own state.

-> They are simply JavaScript functions that may or may not receive data as parameters.

-> We can create a function that takes props(properties) as input and returns what should be rendered.

-> Example:

```
function WelcomeMessage(props) {
    return <h1>Welcome to the , {props.name}</h1>;
}
```

-> Data dealing perspective:

-> Controlled Components:

-> Support instant field validation, allow you to conditionally disable/enable buttons, enforce input formats, and are more "the React way".

-> It's typically recommended that you favor controlled components over uncontrolled components.

-> Un Controlled Components:

-> Where your form data is handled by the DOM, instead of inside your React component.

-> Typically easier to implement since you just grab the value from the DOM using refs.

->

-> Stateless Components:

-> Stateless components are simple functional component without having a local state.

-> But remember there is a hook in react to add state behavior in functional component as well.

-> Statelfull Components:

-> Stateful component can contains the state object and event handling function, user actions as well.

--> Container:

-> Container is an informal term for a React component that is connect -ed to a redux store.

-> Containers receive Redux state updates and dispatch actions

-> Component which is responsible for fetching data and displaying is called smart or container components.

-> Data can be come from redux, any network call or third party subscription.

-> Containers are very similar to components, the only difference is that containers are aware of application state.

-> If part of your webpage is only used for displaying data (dumb) then make it a component. If you need it to be smart and aware of the state (whenever data changes) in the application then make it a container.

--> Components VS Containers:

-> Component:

-> Presentational Component / Dumb Component

-> Components let you split the UI into independent, reusable pieces, and think about each piece in isolation.

-> They are sometimes called "presentational components" and the main concern is how things look.

-> No redux.

-> To read data, it uses props.

-> To change data, it invokes callbacks from props.

-> Containers:

-> Smart Component

-> Containers are just like components without UI and Containers are concerned with how things work.

-> How things work. It mainly talks to the redux store for getting and updating the data.

-> Uses Redux.

-> To read data, it subscribe to redux state.

-> To change data, it dispathes redux actions.

--> State:

-> State is the place where the data comes from.

-> We should always try to make our state as simple as possible and minimize the number of stateful components.

-> The state object is where you store property values that belongs to the component

-> When the state object changes, the component re-renders.

-> If we have, for example, ten components that need data from the state, we should create one container component that will keep the state for all of them.

-> If you try to update state directly (without setState method) then it won't re-render the component.

-> state is managed within the component (similar to variables declared within a function).

-> State on the other hand is still variables, but directly initialized and managed by the component.

-> Satate only get used of the component han an internal value that only affects only that component rather than other components.

-> state is managed within the component similar to variables declared within a function.

--> Props:

-> Props stand for "Properties."

-> Used for passing data from one component to another.

-> Data is read-only, which means that data coming from the parent should not be changed by child components.

-> They are read-only components.

-> It is an object which stores the value of attributes of a tag and work similar to the HTML attributes.

-> It gives a way to pass data from one component to other components. It is similar to function arguments.

-> Both are plain JavaScript objects.

-> props get passed to the component (similar to function parameters).

-> Props are immutable. When we need immutable data in our component, we can just add props to reactDOM.render() function in main.js and use it inside our component.

-> In a React component, props are variables passed to it by its parent component.

-> Props used when we want to use a thing in other components (means affects the other compoentns too).

-> Props Like injecting a props into the header from "Layout" component like <Header title={title_Var_decalred_above}/>.

When we access props in the header compoent file, this will show an object containing title etc.

-> Props get passed to the component similar to function parameters.

-> Validations:

->

->

--> Props VS State:

-> Props are read-only. => State changes can be asynchronous.

-> Props are immutable. => State is mutable.

-> Props allow you to pass data from one component to other components as an argument. => State holds information about the components.

-> Props can be accessed by the child component. => State cannot be accessed by child components.

-> Props are used to communicate between components. => States can be used for rendering dynamic changes with the component.

-> Stateless component can have Props. => Stateless components cannot have State.

-> Props make components reusable. => State cannot make components reusable.

-> Props are external and controlled by whatever renders the component. => The State is internal and controlled by the React Component itself.

-> States can only be used by class components.

-> Props can be used by class as well as other components.

-> State is local to a component and cannot be used in other components.

-> Props allow child components to read values from parent components.

-> Conditions:

-> Can get initial value from parent Component: Yes => Yes

-> Can be changed by parent Component: Yes => No

-> Can set default values inside Component: Yes => Yes

-> Can change inside Component: No => Yes

-> Can set initial value for child Components: Yes => Yes

-> Can change in child Components: Yes => No

-> Similarities:

-> Both are plain JS object.

-> Both can contain default values.

-> Both are read-only when they are using by this.

--> Constructor:

-> Method used to initialize an object's state in a class.

-> It automatically called during the creation of an object in a class.

-> The constructor in a React component is called before the component is mounted.

-> When you implement the constructor for a React component, you need to call super(props) method before any other statement.

-> If you do not call super(props) method, this.props will be undefined in the constructor and can lead to bugs.

-> In React, constructors are mainly used for two purposes:

-> It used for initializing the local state of the component by assigning an object to this.state.

-> It used for binding event handler methods that occur in your component.

-> Note: If you neither initialize state nor bind methods for your React component, there is no need to implement a constructor for React component.

-> You cannot call setState() method directly in the constructor().

-> If the component needs to use local state, you need directly to use 'this.state' to assign the initial state in the constructor.

-> The constructor only uses this.state to assign initial state, and all other methods need to use set.state() method.

--> Component API:

-> ReactJS component is a top-level API. It makes the code completely individual and reusable in the application.

-> Important Methods:

-> setState():

-> This method is used to update the state of the component.

-> This method does not always replace the state immediately. Instead, it only adds changes to the original state.

-> It is a primary method that is used to update the user interface(UI) in response to event handlers and server responses.

-> Note: In the ES6 classes, this.method.bind(this) is used to manually bind the setState() method.

-> forceUpdate():

-> This method allows us to update the component manually.

-> Syntax:

-> Component.forceUpdate(callback);

-> Example:

-> To generate the random numbers.

-> findDOMNode():

-> For DOM manipulation, you need to use ReactDOM.findDOMNode() method.

-> This method allows us to find or access the underlying DOM node.

-> syntax:

-> ReactDOM.findDOMNode(component);

--> Forms:

-> React offers a stateful, reactive approach to build a form.

-> The component rather than the DOM usually handles the React form.

-> In React, the form is usually implemented by using controlled components.

-> Form Inputs Types:

-> Uncontrolled component:

-> The uncontrolled input is similar to the traditional HTML form inputs.

-> The DOM itself handles the form data.

-> Here, the HTML elements maintain their own state that will be updated when the input value changes.

-> To write an uncontrolled component, you need to use a ref to get form values from the DOM.

-> In other words, there is no need to write an event handler for every state update.

-> You can use a ref to access the input field value of the form from the DOM.

-> Controlled component:

-> The input form element is handled by the component rather than the DOM.

-> The mutable state is kept in the state property and will be updated only with setState() method.

-> Controlled components have functions that govern the data passing into them on every onChange event, rather than grabbing the data only once, e.g., when you click a submit button.

-> This data is then saved to state and updated with setState() method.

-> A controlled component takes its current value through props and notifies the changes through callbacks like an onChange event.

-> Controlled component VS Uncontrolled component:

-> It does not maintain its internal state. => It maintains its internal states.

-> Here, data is controlled by the parent component. => Here, data is controlled by the DOM itself.

-> It accepts its current value as a prop. => It uses a ref for their current values.

-> It allows validation control. => It does not allow validation control.

-> It has better control over the form elements and data. => It has limited control over the form elements and data.

-> Top Libraries:

-> React Hook Form

-> Formik

-> React Final Form

--> Redux:

-> Redux is a pattern and library for managing and updating application state, using events called "actions".

-> It serves as a centralized store for state that needs to be used across your entire application, with rules ensuring that the state can only be updated in a predictable fashion.

-> Redux helps you manage "global" state - state that is needed across many parts of your application.

-> Redux guides you towards writing code that is predictable and testable, which helps give you confidence that your application will work as expected.

-> React uses Redux for building the user interface.

-> It allows React components to read data from a Redux Store, and dispatch Actions to the Store to update data.

-> Redux was inspired by Flux. Redux studied the Flux architecture and omitted unnecessary complexity.

-> 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.

-> Architecture/Cycle:

-> "UI" triggers "Actions".

-> "Actions" sent to "Reducers".

-> "Reducers" updates "Store".

-> "Store" contains "State".

-> "State" defines "UI".

-> When Should I Use Redux:

-> You have large amounts of application state that are needed in many places in the app.

-> The app state is updated frequently over time.

-> The logic to update that state may be complex.

-> The app has a medium or large-sized codebase, and might be worked on by many people.

-> When we have multiple components that need to share and use the same state, especially if those components are located in different parts of the application.

-> NOTE: Not all apps need Redux. So, decide first.

-> Substitute of Redux (Not a Good Approach):

-> Extract the shared state from the components, and put it into a centralized location outside the component tree.

-> With this, our component tree becomes a big "view", and any component can access the state or trigger actions, no matter where they are in the tree!

-> Redux Toolkit:

-> Recommended approach for writing Redux logic.

-> It contains packages and functions that we think are essential for building a Redux app.

-> Redux DevTools Extension:

-> Shows a history of the changes to the state in your Redux store over time.

-> This allows you to debug your applications effectively, including using powerful techniques like "time-travel debugging".

-> Redux Terms and Concepts:

-> State Management:

->

-> Immutability:

-> Mutable:

-> "Mutable" means "changeable". If something is "immutable", it can never be changed.

-> Example:

-> const obj = { a: 1, b: 2 } , obj.b = 3

-> const arr = ['a', 'b'] , arr.push('c')

-> This is called mutating the object or array.

-> It's the same object or array reference in memory, but now the contents inside the object have changed.

-> Immutable:

-> In order to update values immutably, your code must make copies of existing objects/arrays, and then modify the copies.

-> We can do this by hand using JavaScript's array / object spread operators, as well as array methods that return new copies of the array instead of mutating the original array

-> Example:

const arr2 = arr1.concat('c')

-> Terminology:

-> Actions:

-> An action is a plain JavaScript object that has a type field.

-> You can think of an action as an event that describes something that happened in the application.

-> All changes to the application state happen via an "action."

-> We say that actions (reports) are "dispatched" to the store to let it know which things happened

-> Actions are the only source of information for the store as per Redux official documentation.

-> Example:

-> store.dispatch({ type: 'LOGIN_SUCCEEDED' })

-> Actions should be used to report a thing that happened, not cause something to happen.

-> Action type should be past like "succeeded".

-> Reducers:

-> It's a plain old function you write that takes the current state, and the action to be processed, and returns the state as it should be, based on that action occurring.

-> Reducers are functions that calculate a new state value based on previous state + an action

-> If an action is a "news report" a reducer is a person reading the report and choosing if they want to change anything about themselves in response.

-> If a reducer updates the application state in response to an action is entirely up to the reducer.

-> So, it is not only fanncy thiig to keep the reducers. Infact it is because multiple reducers can update the single state on a single action.

-> Example:

```
import { createStore } from 'redux';
const reducer = (state, action) => {
if (action.type === 'LOGGED_IN') {
return {
isLoggedIn: true
}
}
return state
}
```

-> Redux Application Data Flow:

-> Earlier, we talked about "one-way data flow", which describes this sequence of steps to update the app:

-> State describes the condition of the app at a specific point in time

-> The UI is rendered based on that state

-> When something happens (such as a user clicking a button), the state is updated based on what occurred

-> The UI re-renders based on the new state.

-> For Redux specifically, we can break these steps into more detail:

-> Initial setup:

-> A Redux store is created using a root reducer function

-> The store calls the root reducer once, and saves the return value as its initial state

-> When the UI is first rendered, UI components access the current state of the Redux store, and use that data to decide what to render. They also subscribe to any future store updates so they can know if the state has changed.

-> Updates:

-> UI => Even Handler (Action which dispathes) => Store (having reducers and it updates the state in the store) => UI (result will render to the UI)

-> Something happens in the app, such as a user clicking a button which deposites 10$ in UI.

-> Event Handler dispath 10$ to the store with type="depopsit".

-> Store will update the previous state which was 0$ (or default state for the first time) with the new state.

-> 0$ will be replaced by 10$ on UI.

-> https://redux.js.org/basics/example

--> Flux:

-> Flux is a programming concept, where the data is uni-directional till it renders.

-> Flux is an application architecture that Facebook uses internally for building the client-side web application with React.

-> It is not a library nor a framework.

-> It is a kind of architecture that complements React as view and follows the concept of Unidirectional Data Flow model.

-> It is useful when the project has dynamic data, and we need to keep the data updated in an effective manner.

-> It reduces the runtime errors.

-> Here, you should not be confused with the Model-View-Controller (MVC) model.

-> Although, Controllers exists in both, but Flux controller-views (views) found at the top of the hierarchy.

-> It retrieves data from the stores and then passes this data down to their children.

-> Flow:

Actions => Dispather => Store => View (Actions => State => View)

-> Flux Elements:

-> Actions:

-> Actions are sent to the dispatcher to trigger the data flow.

-> It is an action creator or helper methods that pass the data to the dispatcher.

-> Dispatcher:

-> This is a central hub of the app. All the data is dispatched and sent to the stores.

-> 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.

-> It is a place which handled all events that modify the store.

-> The dispatcher's API has five methods:

-> register(): It is used to register a store's action handler callback.

-> unregister(): It is used to unregisters a store's callback.

-> waitFor(): It is used to wait for the specified callback to run first.

-> dispatch(): It is used to dispatches an action.

-> isDispatching(): It is used to checks if the dispatcher is currently dispatching an action.

-> Store:

-> Store is the place where the application state and logic are held.

-> Every store is maintaining a particular state and it will update when needed.

-> It is similar to the model in a traditional MVC.

-> 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.

-> View:

-> 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.

-> The view will receive data from the store and re-render the app.

-> Flux Pros:

-> Single directional data flow is easy to understand.

-> The app is easier to maintain.

-> The app parts are decoupled.

-> Using Redux:

-> npm install --save react-redux

-> mkdir actions then, "touch actions.js" in it.

-> mkdir reducers then, "touch reducers.js" in it.

-> mkdir components then, "touch AddTodo.js", "touch Todo.js", "touch TodoList.js" in it.

-> Actions:

-> Actions are JavaScript objects that use type property to inform about the data that should be sent to the store.

-> We are defining ADD_TODO action that will be used for adding new item to our list.

-> The addTodo function is an action creator that returns our action and sets an id for every created item.

-> https://www.tutorialspoint.com/reactjs/reactjs_using_flux.htm

--> Rdux VS Flux:

-> The main difference between Flux and Redux is how they handle actions.

-> In the case of Flux, we usually have multiple stores and a dispatcher, whereas Redux has a single store, which means a dispatcher is not needed.

-> Each component is built to handle specific development aspect of an application.

-> It is one of the most used web development frameworks to create scalable projects.

-> Components:

-> Model: It is responsible for maintaining the behavior and data of an application.

--> MVC VS Flux:

-> It supports Bi-directional data Flow model. => It supports Uni-directional data flow model.

-> In this, data binding is the key. => In this, events or actions are the keys.

-> It is synchronous. => It is asynchronous.

-> Controllers handle everything(logic). => Stores handle all logic.

-> It is hard to debug. => It is easy to debug because it has common initiating point: Dispatcher.

-> It is difficult to understand as the project size increases. => It is easy to understand.

-> Testing of application is difficult. => Testing of application is easy.

-> Scalability is complex. => It can be easily scalable.

--> Web Pack Dev Server Library:

-> It is just like a nodemon that automatically refreshes the page.

--> Events:

--> Types:

-> Clipboard:

-> onCopy

-> onCut

-> onPaste

-> Keyboard:

-> onKeyDown

-> onKeyPress

-> onKeyUp

-> Focus:

-> onFocus

-> onBlur

-> Form:

-> onChange

-> onInput

-> onSubmit

-> Mouse:

-> onClick

-> onDrag

-> onDrop

-> onMouseUp

-> etc.

-> Selection:

-> onSelect

-> UI:

-> onScroll

-> Media:

-> onError

-> onPause

-> onPlay

-> etc.

-> Image:

-> onLoad

-> onError

--> Handling in react:

-> The react event handling system is known as Synthetic Events.

-> The synthetic event is a cross-browser wrapper of the browser's native event.

-> Cycle:

-> App => React Virtual DOM => DOM

-> React events are named as camelCase instead of lowercase.

-> With JSX, a function is passed as the event handler instead of a string.

-> Example:

-> HTML: <button onclick="showMessage()"> </button>

-> React: <button onClick={showMessage}> </button>

--> Conditional Rendering:

-> In React, conditional rendering works the same way as the conditions work in JavaScript.

-> We use JavaScript operators to create elements representing the current state, and then React Component update the UI to match them.

-> Example:

-> If a user logged in, render the logout component to display the logout button. If a user not logged in, render the login component to display the login button.

-> Types:

-> If

-> Ternary operator

-> Logical && operator

-> Switch case operator

-> Conditional Rendering with enums

--> Lists:

-> Lists are used to display data in an ordered format and mainly used to display menus on websites.

-> The map() function is used for traversing the lists.

-> Rendering Lists inside components:

```
-> const myLists = props.myLists;
    const listItems = myLists.map((myList) =>
        <li>{myList}</li>
    ); // in render function
    <ul>{listItems}</ul> // in return within render function
```

--> Map Method:

-> It is because of the uniqueness of each stored key. It is mainly used for fast searching and looking up data.

-> Usage:

-> Traversing the list element:

-> Example:

```
const myLists = ['A', 'B', 'C', 'D', 'D'];
const listItems = myLists.map((myList) =>
        <li>{myList}</li>
); // in render
 <ul>{listItems}</ul> // in return within render. it is usage
```

-> Traversing the list element with keys:

-> Example:

```
const numbers = [1, 2, 3, 4, 5];
const listItems = numbers.map((number) =>
    <ListItem key={number.toString()}
        value={number} />
);
```

--> Keys:

-> A unique identifier.

-> It is used to identify which items have changed, updated, or deleted from the Lists.

-> It is useful when we dynamically created components or when the users alter the lists.

-> It also helps to determine which components in a collection needs to be re-rendered instead of re-rendering the entire set of components every time.

-> Example: We want to display nav items dynamically.

```
const stringLists = [ 'Home', 'Login', 'Page1' ];
const updatedLists = stringLists.map((strList)=>{
    <li key={strList.id}> {strList} </li>;
});
```

--> Refs:

-> Shorthand used for references.

-> It is similar to keys in React.

-> The ref is used to return a reference to the element.

-> Refs should be avoided in most cases, however, they can be useful when we need DOM measurements or to add methods to the components.

-> 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.

-> When to Use Refs:

-> When we need DOM measurements such as managing focus, text selection, or media playback.

-> When integrating with third-party DOM libraries.

-> It can also use as in callbacks.

-> When to not use Refs:

-> Its use should be avoided for anything that can be done declaratively.

-> 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.

-> Example:

-> this.refs.myVariable

--> Fragments:

-> In React, whenever you want to render something on the screen, you need to use a render method inside the component.

-> This render method can return single elements or multiple elements.

-> To render multiple elements, we need to wrap all elements ina div tag eithin the render function.

-> Example:

```
<div>
    <h2> Hello World! </h2>
    <p> Welcome to the JavaTpoint. </p>
    <NavMenu />
</div>
```

-> This extra node (div tag) to the DOM sometimes results in the wrong formatting of your HTML output and also not loved by the many developers.

-> So, we can use fragment instead of div tag.

-> Example:

```
<React.Fragment>
    <h2> Hello World! </h2>
    <p> Welcome to the JavaTpoint. </p>
    <NavMenu />
</React.Fragment>
```

-> Why we use Fragments:

-> It makes the execution of code faster as compared to the div tag.

-> It takes less memory.

-> Fragments Short Syntax:

-> use "<>" instead of "<React.Fragment>".

-> The shorthand syntax does not accept key attributes

-> Note: Key is the only attributes that can be passed with the Fragments.

--> Data Sharing:

-> Props: Parent to Child (data passing).

-> Navigation:

-> Routing

-> Routing Params

-> Routing Data Passing (Between Siblings)

*-> ***: Child to Parent*

--> Style Sheets:

-> JSON Object styles.

--> "This" Keyword:

->

->

--> Routing:

-> Private Routing(AuthGuard)

-> Data passing through URL.

--> Router:

-> Installation:

-> React contains three different packages for routing. These are:

-> react-router: It provides the core routing components and functions for the React Router applications.

-> Syntax: npm install react-router

-> react-router-native: It is used for mobile applications.

-> react-router-dom: It is used for web applications design.

-> It is not possible to install react-router directly in your application. To use react routing, first, you need to install react-router-dom modules in your application.

-> Syntax: npm install react-router-dom --save

-> Imports:

-> import { Route, Link, BrowserRouter as Router } from 'react-router-dom

-> Route:

-> It is used to define and render component based on the specified path. It will accept components and render to define what should be rendered.

-> Example:

<IndexRoute component = {Home} />

<Route path = "home" component = {Home} />

-> Components in React Router:

-> There are two types of router components:

-> <BrowserRouter>: It is used for handling the dynamic URL.

-> <HashRouter>: It is used for handling the static request.

-> "Link" and "to":

-> like href

-> But it will not refresh the page.

-> Example:

-> <Link to="/about">About</Link>

-> Switch:

-> Use in routes file.

-> Optional.

-> Will not check further router on matching specific route.

-> Used to render components only when the path will be matched. Otherwise, it returns to the not found component.

-> Example:

-> Create file "notfound.js".

-> index.js:

```
<Switch>
    <Route exact path="/" component={App} />
    <Route path="/about" component={About} />
    <Route path="/contact" component={Contact} />
    <Route component={Notfound} />
</Switch>
```

-> Benefits:

-> In this, it is not necessary to set the browser history manually.

-> Link uses to navigate the internal links in the application. It is similar to the anchor tag.

-> It uses Switch feature for rendering.

-> The Router needs only a Single Child element.

--> Component LifeCycle:

-> Initialization:

-> In this phase react component prepares setting up the initial state and default props. These default properties are done in the constructor of a component.

-> It is the birth phase of the lifecycle of a ReactJS component.

-> Here, the component starts its journey on a way to the DOM.

-> The initial phase only occurs once.

-> Phase Cycle:

-> getDefaultProps() => getInitialState()

-> Mounting:

-> The react component is ready to mount in the browser DOM.

-> This phase covers componentWillMount and componentDidMount lifecycle methods.

-> Mounting is the process of outputting the virtual representation of a component into the final UI representation (e.g. DOM or Native Components).

-> In a browser that would mean outputting a React Element into an actual DOM element (e.g. an HTML div or li element) in the DOM tree.

-> Phase Cycle:

-> ComponentWillMount() => render() => ComponentDidMount()

-> Updating:

-> In this phase, the component get updated in two ways, sending the new props and updating the state.

-> This phase covers shouldComponentUpdate, componentWillUpdate and componentDidUpdate lifecycle methods.

-> Phase Cycle:

-> Props:

-> ComponentWillReceiveProps() => ShouldComponentUpdate() => ComponentWillUpdate() => render() => ComponentDidUpdate()

-> State:

-> shouldComponentUpdate => ComponentWillUpdate => render => ComponentDidUpdate

-> Unmounting:

-> The component is not needed and get unmounted from the browser DOM.

-> This phase include componentWillUnmount lifecycle method.

-> Remove a mounted React component from the DOM and clean up its event handlers and state.

-> If no component was mounted in the container, calling this function does nothing.

-> Returns true if a component was unmounted and false if there was no component to unmount.

-> Phase Cycle:

-> ComponentWillUnMount()

--> Destructuring:

-> Getting specific attribute from a json.

-> Example:

-> import {Link} from 'react-router-dom';

-> This will get only "Link" from 'react-router-dom'.

--> Use of className despite class:

-> Class is a keyword in javascript and JSX is an extension of javascript.

-> That's the principal reason why React uses className instead of class.

--> Side effects meaning in React:

-> Anything that affects something outside the scope of the function being executed.

-> Example:

-> Network request, Communicating with a third party, Caches, etc.

--> ES5 VS ES6:

-> ES6: import ReactDOM from 'react-dom'

-> ES5: var ReactDOM = require('react-dom')

--> React CSS:

-> CSS in React is used to style the React App or Component.

-> The style attribute is the most used attribute for styling in React applications, which adds dynamically-computed styles at render time.

-> It accepts a JavaScript object in camelCased (like "background-color" as "backgroundColor") properties rather than a CSS string.

-> Major Types:

-> Inline Styling:

-> Example:

-> <h1 style={{color: "Green"}}>Hello JavaTpoint!</h1>

-> <p style={{backgroundColor: "lightgreen"}}>Here, you can find all CS tutorials.</p>

-> Note: You can see in the above example, we have used two curly braces. t is because, in JSX, JavaScript expressions are written inside curly braces, and JavaScript objects also use curly braces.

-> Using Js objects:

-> Example:

```
const mystyle = {
        color: "Green",
        backgroundColor: "lightBlue"
}; // this obj is in the render function
<h1 style={mystyle}>Hello JavaTpoint</h1>  // it is in return within the render function
```

-> CSS Stylesheet:

-> Like in angular. Apply css to body, h1 etc that will automatically apply to the complete page but we need to import the css file.

-> CSS Module:

-> It is a CSS file where all class names and animation names are scoped locally by default.

-> It is available only for the component which imports it.

-> Create a file "myStyles.module.css".

-> Syntax Example:

```
.mystyle {
    background-color: #cdc0b0;
    color: Red;
} // define object with dot
```

-> Usage Example:

-> import styles from './myStyles.module.css';

-> <h1 className={styles.mystyle}>Hello JavaTpoint</h1>

-> Styled Components:

-> Styled-components is a library for React.

-> It uses enhance CSS for styling React component systems in your application, which is written with a mixture of JavaScript and CSS.

-> The styled-components provides:

-> Automatic critical CSS

-> No class name bugs

-> Easier deletion of CSS

-> Simple dynamic styling

-> Painless maintenance

-> Installation:

-> npm install styled-components --save

--> Animation:

-> The animation is a technique in which images are manipulated to appear as moving images.

-> It is one of the most used technique to make an interactive web application.

-> In React, we can add animation using an explicit group of components known as the React Transition Group.

-> React Transition group has mainly two APIs to create transitions. These are:

-> ReactTransitionGroup: It uses as a low-level API for animation.

-> ReactCSSTransitionGroup: It uses as a high-level API for implementing basic CSS transitions and animations.

--> Bootstrap:

-> Single-page applications gaining popularity, as a result, jQuery is not a necessary requirement for building web apps.

-> Today, React has the most used JavaScript framework for building web applications, and Bootstrap become the most popular CSS framework.

-> Adding Bootstrap for React:

-> The three most common ways are given below:

-> Using the Bootstrap CDN

-> Bootstrap as Dependency

-> React Bootstrap Package:

-> The two most popular Bootstrap packages are::

-> react-bootstrap

-> reactstrap

--> Table:

-> The react-table is a lightweight, fast, fully customizable (JSX, templates, state, styles, callbacks), and extendable Datagrid built for React.

-> It is fully controllable via optional props and callbacks.

-> Features:

-> It is lightweight at 11kb (and only need 2kb more for styles).

-> It is fully customizable (JSX, templates, state, styles, callbacks).

-> It is fully controllable via optional props and callbacks.

-> It has client-side & Server-side pagination.

-> It has filters.

-> Pivoting & Aggregation

-> Minimal design & easily themeable

-> Installation:

-> npm install react-table

-> Usage:

-> import ReactTable from "react-table";

-> Example:

```
const data = [{
    name: 'Ayaan',
    age: 26
    },{
    name: 'Ahana',
    age: 22
} // data inside the table
const columns = [{
    Header: 'Name',
    accessor: 'name'
    },{
    Header: 'Age',
    accessor: 'age'
}]  // columns of the table
<ReactTable
    data={data}
    columns={columns}
    defaultPageSize = {2}
    pageSizeOptions = {[2,4, 6]}
/>  // usage in the return function
```

--> Higher-Order Components (HOC):

-> HOC is an advanced technique for reusing component logic.

-> It is a function that takes a component and returns a new component.

-> In other words, it is a function which accepts another function as an argument.

-> You can do many tasks with HOC, some of them are given below:

-> Code Reusability

-> Props manipulation

-> State manipulation

-> Render highjacking

--> Code Splitting:

-> The React app bundled their files using tools like Webpack or Browserfy.

-> Bundling is a process which takes multiple files and merges them into a single file, which is called a bundle.

-> Example:

```
import { add } from './math.js';
console.log(add(16, 26)); // 42
```

-> React.lazy:

-> The best way for code splitting into the app is through the dynamic import() syntax.

-> The React.lazy function allows us to render a dynamic import as a regular component.

-> Example:

-> Without using it:

-> import ExampleComponent from './ExampleComponent';

-> With using it:

-> const ExampleComponent = React.lazy(() => import('./ExampleComponent'));

-> The above code snippet automatically loads the bundle which contains the ExampleComponent when the ExampleComponent gets rendered.

-> Suspense:

->

-> Error boundaries:

->

-> Route-based code splitting:

->

-> Named Export:

->

--> Context:

-> Context allows passing data through the component tree without passing props down manually at every level.

-> In React application, we passed data in a top-down approach via props.

-> Sometimes it is inconvenient for certain types of props that are required by many components in the React application.

-> Context provides a way to pass values between components without explicitly passing a prop through every level of the component tree.

-> Usage:

-> Setup a context provider and define the data which you want to store.

-> Setup a context provider and define the data which you want to store.

-> Use a context consumer whenever you need the data from the store.

-> When to use Context:

-> Context is used to share data which can be considered "global" for React components tree.

--> 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.

-> When to use a Hooks:

-> If you write a function component, and then you want to add some state to it, previously you do this by converting it to a class.

-> But, now you can do it by using a Hook inside the existing function component.

-> Rules of Hooks:

-> Only call Hooks at the top level.

-> Only call Hooks from React functions.

--> Portals:

-> The React 16.0 version introduced React portals.

-> A React portal provides a way to render an element outside of its component hierarchy, i.e., in a separate component.

-> Before React 16.0 version, it is very tricky to render the child component outside of its parent component hierarchy.

-> Syntax:

-> ReactDOM.createPortal(child, container)

->Here, the first argument (child) is the component, which can be an element, string, or fragment, and the second argument (container) is a DOM element.

--> Error Boundaries:

-> In the past, if we get any JavaScript errors inside components, it corrupts the ReactJS internal state and put React in a broken state on next renders.

-> There are no ways to handle these errors in React components, nor it provides any methods to recover from them.

-> React 16 introduces a new concept to handle the errors by using the error boundaries.

-> Now, if any JavaScript error found in a part of the UI, it does not break the whole app.

-> Error boundaries are React components which catch JavaScript errors anywhere in our app, log those errors, and display a fallback UI.

-> For simple React app, we can declare an error boundary once and can use it for the whole application.

-> For a complex application which have multiple components, we can declare multiple error boundaries to recover each part of the entire application.

-> Rollbar:

-> We can also report the error to an error monitoring service like Rollbar.

-> his monitoring service provides the ability to track how many users are affected by errors, find causes of them, and improve the user experience.

-> Error boundary in class:

->

-> How to implement error boundaries:

-> Step-1 Create a class which extends React component and passes the props inside it.

-> Step-2 Now, add componentDidCatch() method which allows you to catch error in the components below them in the tree.

-> Step-3 Next add render() method, which is responsible for how the component should be rendered. For example, it will display the error message like "Something is wrong."

-> Example:

->

--> General React Interview Questions:

-> Why can't browsers read JSX?

-> Browsers cannot read JSX directly because they can only understand JavaScript objects, and JSX is not a regular JavaScript object.

-> Thus, we need to transform the JSX file into a JavaScript object using transpilers like Babel and then pass it to the browser.

-> Explain the working of Virtual DOM:

-> Step 1: Whenever any data changes in the React App, the entire UI is re-rendered in Virtual DOM representation.

-> Step 2: Now, the difference between the previous DOM representation and the new DOM is calculated.

-> Step 3: Once the calculations are completed, the real DOM updated with only those things which are changed.

-> What do you understand from "In React, everything is a component."?

-> Components divide the entire React application's UI into small, independent, and reusable pieces of code.

-> React renders each of these components independently without affecting the rest of the application UI.

-> What is arrow function in React? How is it used?

-> New feature of the ES6 standard.

-> If you need to use arrow functions, it is not necessary to bind any event to 'this'.

-> Here, the scope of 'this' is global and not limited to any calling function.

-> It is also called 'fat arrow '(=>) functions.

-> What is an event in React?

-> An event is an action which triggers as a result of the user action or system generated event like a mouse click, loading of a web page, pressing a key, window resizes, etc.

-> Handling events with React have some syntactical differences, which are:

-> React events are named as camelCase instead of lowercase.

-> With JSX, a function is passed as the event handler instead of a string.

-> Is the Shadow DOM the same as the Virtual DOM?

-> No, they are different.

-> The Shadow DOM is a browser technology designed primarily for scoping variables and CSS in web components.

-> The virtual DOM is a concept implemented by libraries in JavaScript on top of browser APIs.

-> What is “React Fiber”?

-> Fiber is the new reconciliation engine in React 16.

-> Its main goal is to enable incremental rendering of the virtual DOM.

-> What are synthetic events in React?

-> A synthetic event is an object which acts as a cross-browser wrapper around the browser's native event.

->

-> How are forms created in React?

-> Example:

```
import React, { Component } from 'react';
class App extends React.Component {
constructor(props) {
super(props);
this.state = {value: ''};
this.handleChange = this.handleChange.bind(this);
this.handleSubmit = this.handleSubmit.bind(this);
}
handleChange(event) {
this.setState({value: event.target.value});
}
handleSubmit(event) {
alert('You have submitted the input successfully: ' + this.state.value);
event.preventDefault();
}
render() {
return (
<form onSubmit={this.handleSubmit}>
<h1>Controlled Form Example</h1>
<label>
Name:
<input type="text" value={this.state.value} onChange={this.handleChange} />
</label>
<input type="submit" value="Submit" />
</form>
);
}
}
export default App;
```

-> What are Pure Components?

-> The React.Component and React.PureComponent differ in the shouldComponentUpdate() React lifecycle method.

-> This method decides the re-rendering of the component by returning a boolean value (true or false).

-> In React.Component, shouldComponentUpdate() method returns true by default.

-> But in React.PureComponent, it compares the changes in state or props to re-render the component.

-> The pure component enhances the simplicity of the code and performance of the application.

Portfolio: https://arqam-dev.github.io/

Reference Links

---

https://redux.js.org/tutorials/essentials/part-1-overview-concepts (REDUX)

https://dev.to/skd1993/creating-a-simple-login-function-with-redux-and-thunk-in-react-native-33ib (REDUX) (Best Link for redux)

https://www.javatpoint.com/ (BEST Link for react)

https://medium.com/frontend-digest/the-best-react-form-library-to-use-in-2020-11e93c267e4 (Forms)