**What is React?**

React is a JavaScript library for building user interfaces.

React is used to build single-page applications.

React allows us to create reusable UI components.

**Advantages of React:**

* Easy to learn
* Performance Enhancement – virtual DOM
* Reusable Components
* Component-Based Architecture
* Creating Dynamic Web Applications Becomes Easier
* The Support of Handy Tools
* SEO Friendly

**Disadvantages of React:**

* Poor Documentation
* No standard / recommended way to develop application
* Frequent Updates - React and its ecosystem evolve rapidly, which can lead to compatibility issues with third-party libraries and frameworks
* Scoped CSS - Handling CSS in React can be challenging, especially when dealing with scoped styles for individual components, which may require additional tools or libraries.
* State Management - While React provides built-in state management features, handling complex state management in large applications may require additional libraries like Redux, which adds complexity to the project.

**React Components**

React components are the building blocks of React applications. They are reusable pieces of code that define the user interface of an application.

There are two types of components in React:

|  |  |
| --- | --- |
| **Class components** | **Functional components** |
| Stateful components | Stateless components |
| Class components are defined using *JavaScript classes*, extending from the *React.Component* class. | Functional components are defined as *JavaScript functions* that accept props as arguments and return JSX. |
| Class components can also have state and lifecycle methods. | Functional components don't have state or lifecycle methods. |

**Class Components:** Class components are created by extending the React.Component class and defining a render() method that returns the component's JSX markup. Class components can also have state and lifecycle methods.

Here's an example of a class component:

import React, { Component } from 'react';

class App extends Component {

render() {

return (

<div>

<h1>Hello, World!</h1>

</div>

);

}

}

export default App;

**Functional Components:** Functional components are created by defining a function that returns the component's JSX markup. Functional components don't have state or lifecycle methods, but they are easier to read and write.

Here's an example of a functional component:

import React from 'react';

function App() {

return (

<div>

<h1>Hello, World!</h1>

</div>

);

}

export default App;

Both class and functional components can accept props as inputs and render dynamic content based on them. Components can also be composed of other components, which makes it easy to build complex user interfaces.

**Explain the differences between controlled and uncontrolled components.**

|  |  |
| --- | --- |
| **Controlled Component** | **Uncontrolled Component** |
| The component is under control of the component’s state. | Components are under the control of DOM. |
| These components are predictable as are controlled by the state of the component. | Are Uncontrolled because during the life cycle methods the data may loss. |
| Internal state is not maintained. | Internal state is maintained. |
| It accepts the current value as props. | We access the values using refs. |
| Controlled by the parent component. | Controlled by the DOM itself. |
| Have better control on the form data and values. | Has very limited control over form values and data. |
| It allows validation control. | It does not allow validation control. |

**Controlled Component Example:**

In a controlled component, the state of the input field is managed by React. The value of the input field is tied to the component's state, and any changes are controlled by React.

In this example, the inputValue state variable is directly tied to the value of the input field. The handleChange function updates the state whenever the input value changes. This way, React is in control of the input field's state.

import React, { useState } from 'react';

const ControlledComponent = () => {

  const [inputValue, setInputValue] = useState('');

  const handleChange = (event) => {

    // Update the state with the new input value

    setInputValue(event.target.value);

  };

  return (

    <div>

      <label>

        Controlled Input:

        <input

          type="text"

          value={inputValue}

          onChange={handleChange}

        />

      </label>

      <p>Input Value: {inputValue}</p>

    </div>

  );

};

export default ControlledComponent;

**Uncontrolled Component Example:**

In an uncontrolled component, the state is not managed by React. Instead, the value is directly accessed from the DOM using refs.

In this example, the input field's value is not controlled by React. Instead, a ref (inputRef) is used to directly access the input field from the DOM. The handleButtonClick function retrieves the input value using the ref. This approach is more imperative and less declarative compared to controlled components.

import React, { useRef } from 'react';

const UncontrolledComponent = () => {

  // Create a ref to store the input DOM element

  const inputRef = useRef(null);

  const handleButtonClick = () => {

    // Access the input value directly from the DOM using the ref

    alert(`Uncontrolled Input Value: ${inputRef.current.value}`);

  };

  return (

    <div>

      <label>

        Uncontrolled Input:

        <input

          type="text"

          ref={inputRef}

        />

      </label>

      <button onClick={handleButtonClick}>Get Input Value</button>

    </div>

  );

};

export default UncontrolledComponent;

**What Is 'State' in ReactJS?**

The state is a built-in React object that is used to contain data or information about the component. A component's state can change over time; whenever it changes, the component re-renders.

**React Props**

Props stands for properties.

Props are arguments passed into React components.

Props are passed to components via HTML attributes.

| **PROPS** | **STATE** |
| --- | --- |
| It is defined and managed by Parent component. The Data is passed from one component to another. | It is defined and managed within the component. The Data is passed within the component only. |
| It is Immutable (cannot be modified). | It is Mutable (can be modified). |
| External data, passed into components. | Internal data of a components. |
| Props are read-only. | The state is both read and write. |

**Classes**

ES6 introduced classes.

A class is a type of function, but instead of using the keyword function to initiate it, we use the keyword class, and the properties are assigned inside a constructor() method.

## **Method in Classes**

<script>

class Car {

constructor(name) {

this.brand = name;

}

present() {

return 'I have a ' + this.brand;

}

}

const mycar = new Car("Ford");

document.write(mycar.present());

</script>

## **Class Inheritance:**

<script>

class Car {

constructor(name) {

this.brand = name;

}

present() {

return 'I have a ' + this.brand;

}

}

class Model extends Car {

constructor(name, mod) {

super(name);

this.model = mod;

}

show() {

return this.present() + ', it is a ' + this.model

}

}

const mycar = new Model("Ford", "Mustang");

document.write(mycar.show());

</script>

## **What is unidirectional data flow in react?**

Unidirectional data flow is a pattern in React that ensures that data only flows in one direction within an application. This means that data changes can only be made in one place, and those changes are propagated down the component tree in a one-way direction.

In a React application with unidirectional data flow, the data flows from the parent component to the child components through props. When a parent component updates its state or props, React automatically re-renders the child components with the updated data.

### **Virtual DOM**

VDOM is the ***virtual representation of Real DOM***

* React update the state changes in Virtual DOM first and then it syncs with Real DOM
* Virtual DOM is just like a ***blueprint of a machine***, can do the changes in the blueprint but those changes will not directly apply to the machine.
* Virtual DOM is a programming concept where a *virtual representation of a UI* is kept in memory synced with “Real DOM” by a library such as ReactDOM and this process is called reconciliation
* Virtual DOM makes the performance faster, not because the processing itself is done in less time. The reason is the amount of changed information – rather than wasting time on updating the entire page, you can dissect it into small elements and interactions.

### **React Memo: (Memoization)**

React.memo is a higher-order component (HOC) in React that is used to optimize the rendering performance of functional components by memoizing them.

When a functional component is wrapped with React.memo, React will only re-render the component if its props have changed. If the props are the same, React will reuse the previously rendered component, which can significantly improve performance by avoiding unnecessary re-renders.

import React from 'react';

const MyComponent = React.memo((props) => {

return (

<div>

{/\* component code \*/}

</div>

);

});

export default MyComponent;

## **Pure Component:**

**Pure Component** is a class component that inherits from React.PureComponent instead of React.Component. Pure Components are similar to regular components, but they have a built-in optimization for performance. The key feature of a Pure Component is that it implements a shouldComponentUpdate method that automatically performs a shallow comparison of the component's props and state to determine whether the component should re-render.

## **Middlewares:**

Middleware is a **third-party extension** that intercepts and modifies data before it reaches the application. Middlewares are commonly used in conjunction with Redux, a popular state management library for React applications. Here are some examples of middlewares used in React:

**Redux Thunk** - is a middleware that allows you to write action creators that return a function instead of an action object. This function can then be used to perform asynchronous operations, such as API calls.

**Redux Saga** - is a middleware that allows you to write complex asynchronous logic, such as long-running processes and complex workflows, in a more readable and testable way.

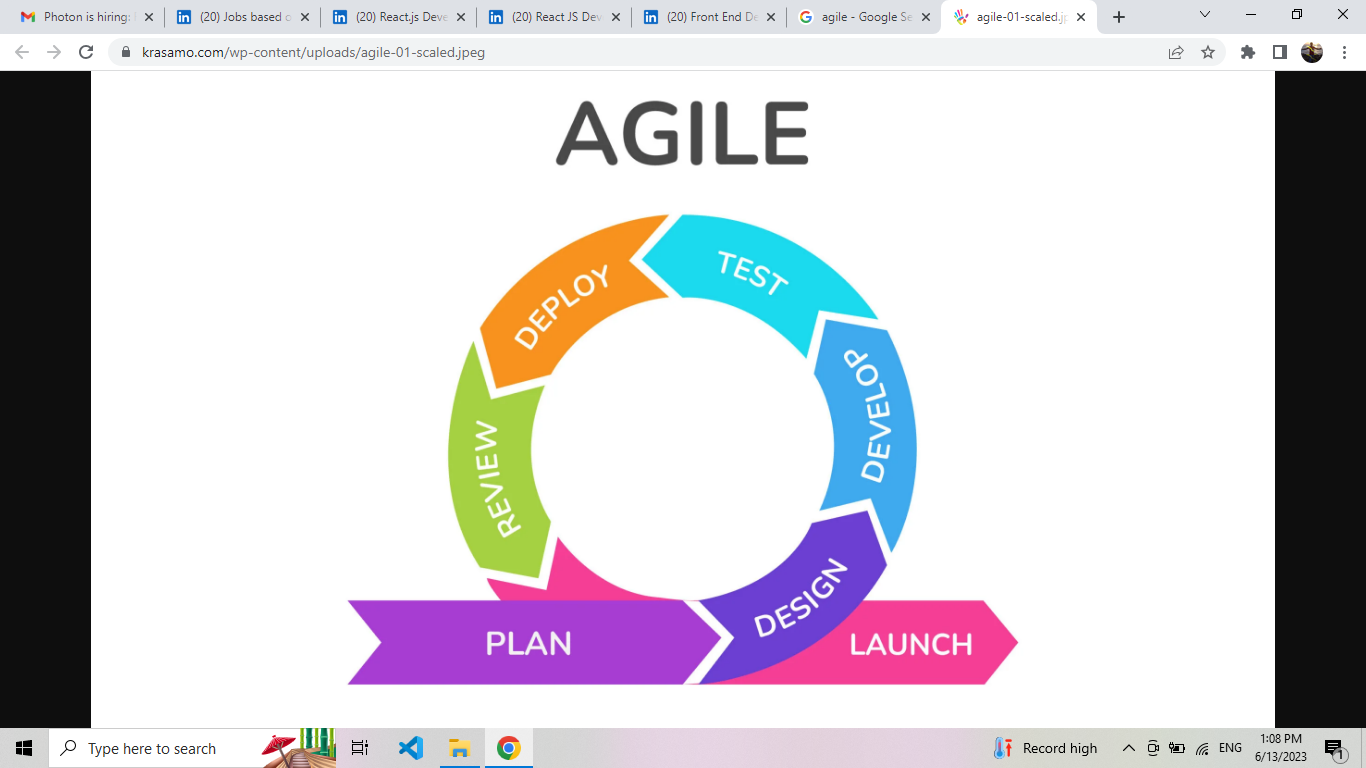
**Redux Logger** - is a middleware that logs Redux actions and state to the console, making it easier to debug your application.

**Redux Persist** - is a middleware that allows you to persist the Redux store to a storage engine, such as local storage or async storage.

**ESLint**

ESLint is a powerful tool for JavaScript and TypeScript developers to ensure code quality, maintain consistency, and improve collaboration within development teams.

**Agile Method:**



**Npm and Yarn:**

npm (Node Package Manager) and Yarn are both package managers for JavaScript and Node.js projects. They allow you to manage and install third-party libraries, frameworks, and tools, making it easier to build and maintain your projects. While they have similar purposes, they have different implementations and some varying features.

npm:

npm is the default package manager for Node.js and is bundled with it.

It has a vast registry of open-source packages, making it the largest package ecosystem for JavaScript.

npm uses a centralized approach, where packages are installed from the npm registry to your project's node\_modules directory.

It provides a command-line interface (CLI) that allows you to perform various package-related operations, such as installing packages, updating dependencies, and publishing your own packages.

npm supports a package.json file that lists your project's dependencies, scripts, and other metadata.

Yarn:

Yarn is a package manager developed by Facebook, designed to address some limitations of npm.

It also utilizes the npm registry, so it can install packages available on npm.

Yarn introduced a few enhancements over npm, such as parallel package downloads, deterministic dependency resolution, and offline mode for faster and more reliable installations.

Yarn has a lock file called yarn.lock, which ensures that the same versions of dependencies are installed across different development environments, making the builds more reproducible.

Yarn offers a CLI similar to npm, providing commands for managing packages and dependencies.

Uses of npm and Yarn:

**Dependency Management:** Both npm and Yarn are primarily used for managing project dependencies. They allow you to define and install external packages needed for your project, making it easier to include libraries and frameworks in your codebase.

**Package Installation:** You can use npm or Yarn to install packages from the npm registry or from other sources like Git repositories. They handle the installation process and manage the dependencies required by those packages.

**Dependency Resolution:** npm and Yarn analyze your project's dependencies and resolve version conflicts, ensuring that compatible versions of packages are installed. This helps prevent dependency issues and ensures a stable development environment.

**Script Execution:** Both package managers provide a way to define scripts in the package.json file. You can run these scripts using the CLI, performing tasks such as building the project, running tests, or starting a development server.

**Package Publishing:** npm and Yarn allow you to publish your own packages to the npm registry, making it easy to share your code and contribute to the JavaScript ecosystem.

While npm is the default package manager for Node.js and is widely used, Yarn gained popularity due to its performance improvements and enhanced features. However, the choice between npm and Yarn ultimately depends on your project's needs and preferences.

**Class components and functional components**

Class components and functional components are two primary ways to define components in React. Here are some key differences between the two:

**Syntax:**

**Class components** -> are defined as **JavaScript classes** that extend the React.Component class. They use the render() method to return the JSX markup.

**Functional components** -> are defined as **JavaScript functions** that return JSX directly.

**State Management:**

Class components have their own state, which can be initialized in the constructor using this.state. The state can be updated using this.setState().

Functional components, by themselves, don't have a built-in way to manage state. However, with the introduction of React Hooks, functional components can now manage state using hooks like useState.

**Lifecycle Methods:**

A component's lifecycle has three main phases: the Mounting Phase, the Updating Phase, and the Unmounting Phase.

The **Mounting Phase** begins when a component is first created and inserted into the DOM. The **Updating Phase** occurs when a component's state or props change. And the **Unmounting Phase** occurs when a component is removed from the DOM.

Class components have lifecycle methods like componentDidMount, componentDidUpdate, and componentWillUnmount, which allow developers to perform certain actions at specific points in a component's lifecycle.

Functional components, prior to React Hooks, didn't have lifecycle methods. With hooks, functional components can use the useEffect hook to achieve similar functionality.

**Babel:**

Babel is a JavaScript compiler that enables developers to write code in the latest version of JavaScript (ES6, ES7, ES8, etc.) and transpile it into an older version of JavaScript that can run in environments that may not yet support the latest features. This process is often referred to as transpiling, as opposed to compiling, because the output code is still in JavaScript.

In the context of React, Babel is commonly used to transpile JSX (JavaScript XML) syntax. JSX is a syntax extension for JavaScript recommended by React that allows you to write HTML elements and components in a syntax similar to XML or HTML. JSX is not natively understood by browsers, so it needs to be transpiled into standard JavaScript.

**Webpack:**

Webpack is a powerful and popular open-source JavaScript module bundler. It takes the various assets, such as JavaScript files, CSS stylesheets, images, and more, and bundles them together into smaller, optimized files that are suitable for deployment. In the context of React applications, Webpack is often used to manage and bundle the project's assets and dependencies.

Here are some key aspects of Webpack and its uses in React:

**Module Bundling:**

Webpack allows you to structure your React application using a modular approach, where each component or module is in a separate file. It then bundles these modules together, optimizing the code for deployment.

**Code Splitting:**

Webpack supports code splitting, which means that instead of loading the entire application in one go, you can split it into smaller chunks. This can lead to faster initial page loads and improved performance.

**Loaders:**

Webpack uses loaders to process different types of files. For example, it can use Babel as a loader to transpile React's JSX and ES6/ESNext code into standard JavaScript that can be understood by browsers.

**Hot Module Replacement (HMR):**

Webpack's HMR feature allows for real-time updates in the browser as you make changes to your React code. This speeds up development by avoiding full page reloads and maintaining the application's state.

**Asset Management:**

Webpack can handle various types of assets such as images, fonts, and CSS stylesheets. It allows you to import these assets directly into your JavaScript code, and it takes care of bundling and optimizing them.

**Environment Configuration:**

Webpack supports different configurations for development and production environments. For example, in a production build, it can minify and uglify the code to reduce file sizes and improve performance.

**Integration with Other Tools:**

Webpack is often used in conjunction with other tools and libraries in the React ecosystem, such as Babel for transpilation, ESLint for code linting, and various plugins for additional features.

**DevServer:**

Webpack DevServer is a development server that comes with Webpack. It provides a convenient way to preview your React application during development. Combined with HMR, it allows for a fast and efficient development workflow.

Here's a simplified example of a Webpack configuration file for a React project:

*// webpack.config.js*

const path = require('path');

module.exports = {

entry: './src/index.js',

output: {

path: path.resolve(\_\_dirname, 'dist'),

filename: 'bundle.js',

},

module: {

rules: [

{

test: /\.(js|jsx)$/,

exclude: /node\_modules/,

use: 'babel-loader',

},

{

test: /\.css$/,

use: ['style-loader', 'css-loader'],

},

{

test: /\.(png|svg|jpg|gif)$/,

use: ['file-loader'],

},

],

},

};

**Rest and Restful API:**

"**REST**" stands for Representational State Transfer, which is an architectural style for designing networked applications. When people refer to a "RESTful API," they are generally talking about an API that follows the principles and constraints of REST.

A **RESTful** API is a type of API that adheres to the principles of REST architecture.

**React.Fragment:**

import React from 'react';

class ExampleComponent extends React.Component {

  render() {

    return (

      <>

        <p>Paragraph 1</p>

        <p>Paragraph 2</p>

      </>

    );

  }

}

In this case, the <> and </> act as shorthand for <React.Fragment> and </React.Fragment>. Using fragments helps keep the HTML structure cleaner, especially when you don't want an additional wrapping element in the output.