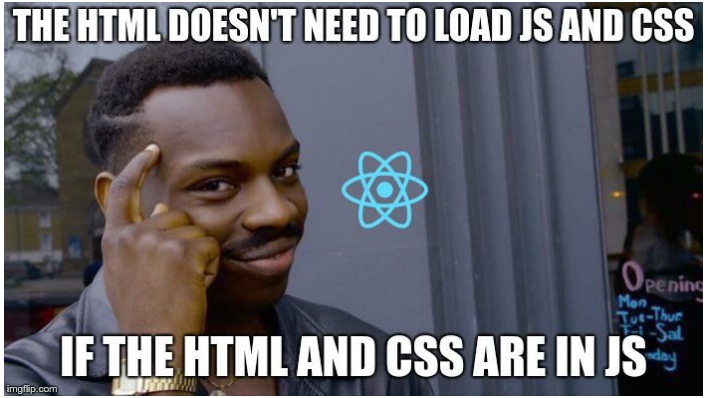
Web development basics

Introduction to React

What is React?

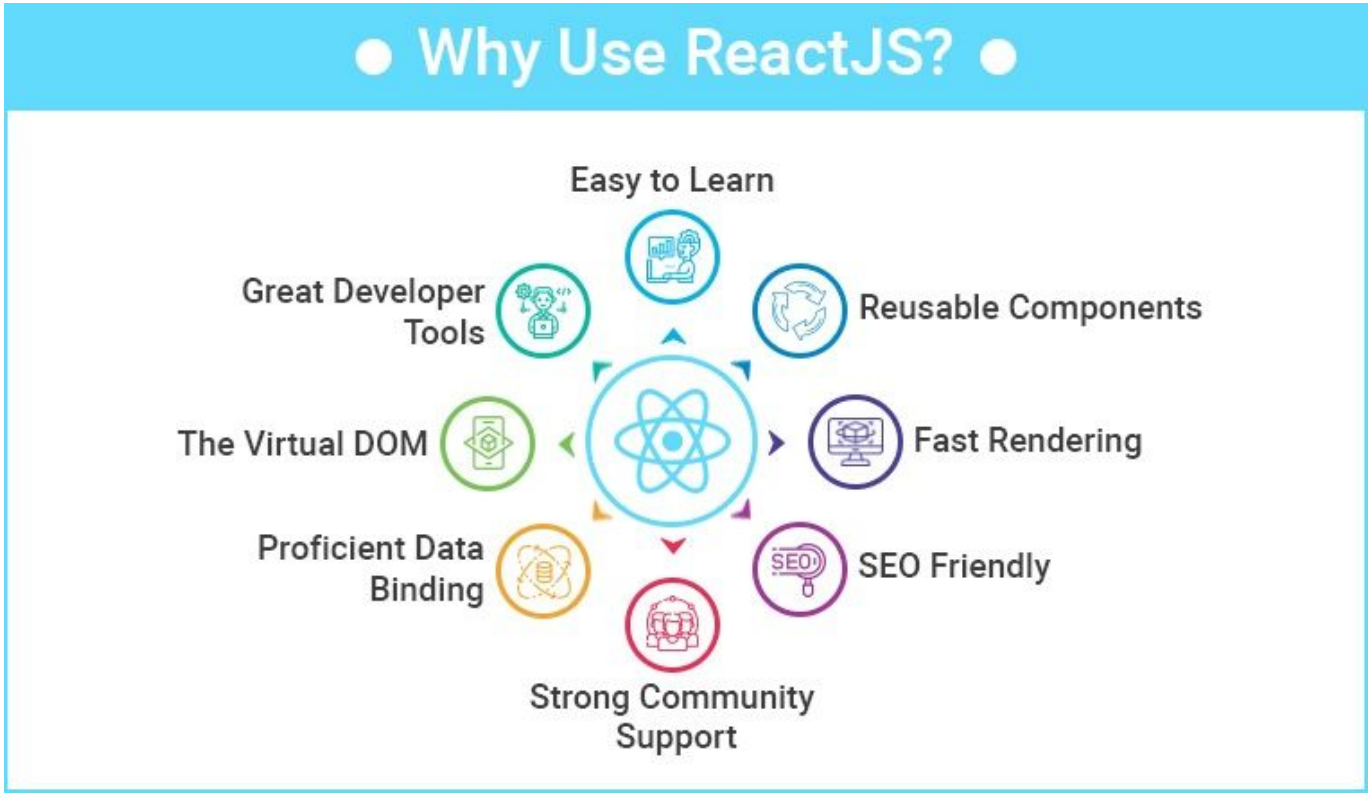
React is a **JavaScript library** for building user interfaces. It was developed and is maintained by Facebook, and it is used to build **single-page applications** and **mobile applications**. React allows developers to create **reusable UI components** and manage the **state** of their applications. The **components** in React are **self-contained** and manage their own state, making it easier to build complex applications with a clear structure. The library uses a **virtual DOM**, which allows it to efficiently update the view when the state of the application changes, making it a fast and performant choice for building modern web applications.



Why use React?

1) **Reusable Components**: React allows developers to create reusable UI components that can be easily composed to build complex applications.
2) **Virtual DOM**: React uses a virtual DOM, which makes it fast and efficient in rendering updates to the user interface.
3) **Declarative Programming**: React uses a declarative programming approach, making it easier to understand and debug code.
4) **Large Community**: React has a large and active community of developers, which provides a wealth of resources, including tutorials, packages, and libraries.
5) **SEO-friendly**: React applications can be easily optimized for search engines, making it a good choice for building applications that need to be discoverable.
6) **Support for Mobile Development**: React Native, a platform for building native mobile applications using React, allows developers to reuse code from their web applications to build high-performance mobile apps.

Why use React?



Create New React App

To create a new React app using create-react-app, you will need to have **Node.js** and **npm** (Node Package Manager) installed on your computer.

Follow these steps to create a new React app:

1) Open a command line terminal (such as **Terminal** on Mac or **Command Prompt** on Windows)
2) Type **npx create-react-app my-app** and hit enter, where "my-app" is the name of your new app.
3) Wait for the process to finish, which could take a few minutes.
4) Once the process is complete, navigate into the newly created app directory by typing cd my-app in the terminal.
5) Type **npm start** to start the development server and run the app in your browser.
6) Open your browser to **http://localhost:3000/** to see your new React app in action.

React application structure

A React application typically has the following structure:

1) **public** directory: contains the index.html file, which is the entry point for the application, and other static assets like images and icons.
2) **src** directory: contains the main source code for the application.
    a) **index.js**: the entry point for the React application, which renders the root component to the DOM.
    b) **App.js**: the root component of the application, which contains other nested components.
    c) **components** directory: contains the individual components that make up the UI of the application.
    d) **styles** directory: contains CSS files for styling the application.
    e) **assets** directory: contains other assets, such as images and icons.
3) **node_modules** directory: contains the dependencies and libraries used by the application, installed through npm.
4) **package.json**: lists the dependencies and scripts needed for the application.
5) **package-lock.json (or yarn.lock)**: a file that ensures that the exact version of dependencies is installed when the application is built.

React JSX

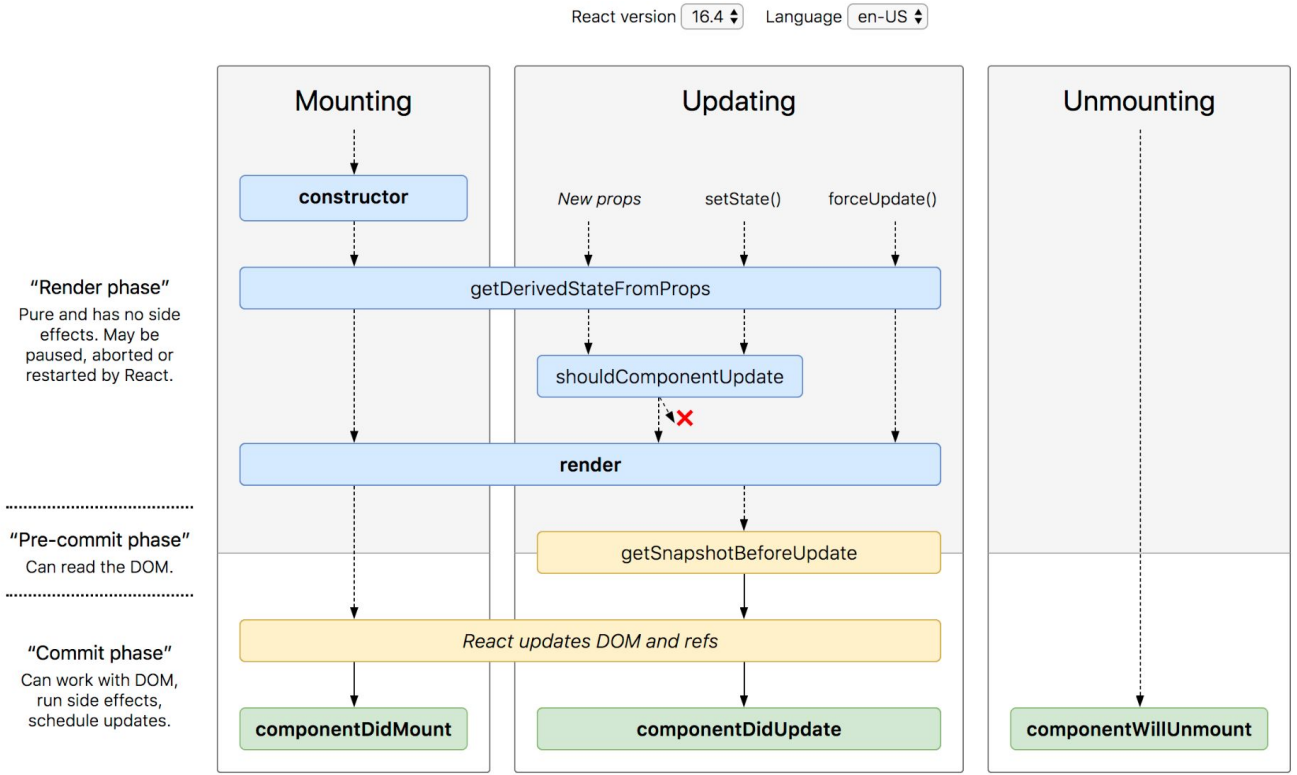
JSX is a **syntax extension** for JavaScript that allows for writing HTML-like code within JavaScript. It is used in React to define and render components. **JSX elements** are transformed into **React elements** that can be processed and rendered by the browser. The combination of **JSX** and **JavaScript** makes it easier for developers to write and understand the structure and behavior of a React component in a single place.

```jsx
import React from 'react';

const Hello = () => {
  return (
    <div>
      <h1>Hello, World!</h1>
      <p>This is a simple example of JSX</p>
    </div>
  );
};

export default Hello;
```

How does React render

1) The **React component tree** is constructed using the React.**createElement**() or **JSX syntax**, which describes the UI.
2) The **React DOM** (Document Object Model) takes the **component tree** and updates the actual **browser DOM** to match it.
3) When a **component state** or **props** change, React updates the **component tree**, and the React DOM updates the **browser DOM** accordingly.
4) React uses a **virtual DOM** to optimize updates to the browser DOM, making sure that only the **necessary** updates are made.
5) The React DOM then updates the browser DOM with the minimum number of changes needed to keep it in **sync** with the **component tree**.
6) This process continues, with the React DOM updating the browser DOM in response to changes in the component tree, creating a **dynamic**, **interactive UI**.

React Life Cycle Methods

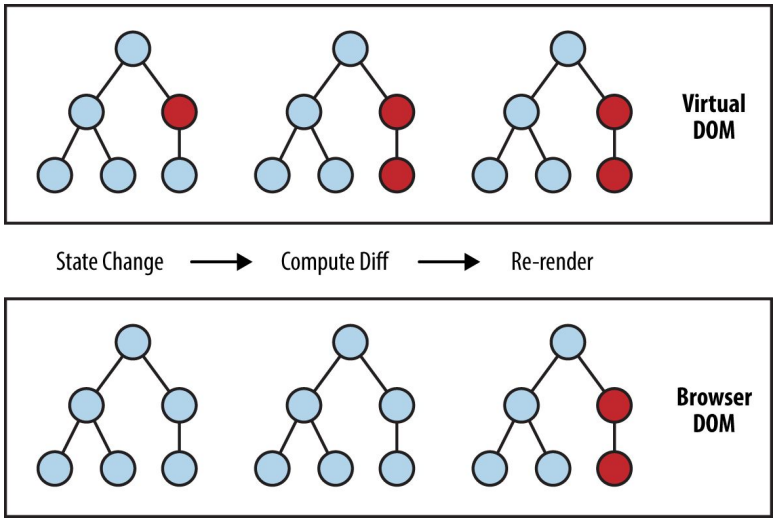


React Virtual DOM

The **Virtual DOM** is a virtual representation of the actual DOM in a web page. In React, it acts as an intermediary between the components and the actual DOM. When a component's state changes, React updates the virtual DOM and then calculates the most efficient way to update the real DOM.

This process is faster than directly manipulating the real DOM because the virtual DOM can be updated quickly in memory, whereas updating the real DOM is much slower due to the time it takes to update the actual page. The Virtual DOM only updates the parts of the real DOM that have changed, reducing the amount of work that needs to be done to update the page.

The React Virtual DOM is also designed to handle all the logic for rendering components and updating the UI, freeing up developers from having to manually manipulate the DOM. This makes it easier to develop complex applications, and it also ensures that the application remains fast and responsive, even as it grows in size.



React Components

React functional components are a simpler way to define components in React, as opposed to using class components. They are just JavaScript functions that receive props as an argument and return React elements to describe the component's UI.

Functional components are usually preferred for stateless components, which don't manage internal state, but only receive data from their parent component via props.

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

```
const Welcome = (props) => {
  return <h1>Hello, {props.name}</h1>;
};
```

```
<Welcome name="Sara" />
```

React Props

React **props** are the data that is passed from a **parent component** to its **child component**. Props allow components to receive and render data, and they are often used to create **reusable**, **composable** components. In functional components, props are passed **as an argument** to the function. In the previous slide you can see an example of a functional component receiving and using props. Here are more advanced examples with multiple props and props **destructurization** (es6)

```
<Product
  name="Shirt"
  price={29.99}
  image="https://example.com/shirt.jpg"
/>
```

```
const Product = ({ name, price, image }) => {
  return (
    <div>
      <h2>{name}</h2>
      <p>Price: ${price}</p>
      <img src={image} alt={name} />
    </div>
  );
};
```

Classwork - create components with props

Steps:

1) Create a new React project using npx create-react-app app-name (if you didn't do that already)
2) In the src directory, create a new file Person.js.
3) In Person.js, create a functional component Person that takes in multiple props: name, age, and location.
4) Use destructuring to extract the props from the props object and assign them to variables name, age, and location.
5) In the component return statement, use the variables to display information about the person.
6) In the component return statement, log the values of the destructured props using console.log to make sure they're being passed correctly.

Classwork - create components with props - solution

```jsx
import React from 'react';

const Person = ({ name, age, location }) => {
  console.log(name, age, location);

  return (
    <div>
      <h2>{name}</h2>
      <p>Age: {age}</p>
      <p>Location: {location}</p>
    </div>
  );
};

export default Person;
```

```jsx
import React from 'react';
import Person from './Person';

function App() {
  return (
    <div>
      <Person name="Jane Doe" age={32} location="San Francisco" />
    </div>
  );
}

export default App;
```

React State

React state is a way of keeping track of dynamic information within a component. Unlike props, state **can be changed within the component**, and it will trigger a **re-render** of the component when it **updates**. In functional components, state is declared as an object within the component, and can be accessed and updated using the **useState** hook.

In this example, the component is using useState to initialize the isToggled state to false. The component also receives a function setToggled that can be used to update the state. When the button is clicked, the component calls setToggled with the opposite of the current value of isToggled to toggle the state. The component then uses the value of isToggled to determine what text to display on the button.

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

const Toggle = () => {
  const [isToggled, setToggled] = useState(false);

  return (
    <div>
      <button onClick={() => setToggled(!isToggled)}>
        {isToggled ? 'On' : 'Off'}
      </button>
    </div>
  );
};

export default Toggle;
```

React State example 2

In this example, the component is using useState to initialize the person state to an object with properties for name, age, and location. The component also receives a function setPerson that can be used to update the state. The component uses the handleChange function to update the state whenever the user changes the value of one of the input fields. The component then uses the values of the person state to render the form.

```
const [person, setPerson] = useState({
  name: '',
  age: '',
  location: '',
});

const handleChange = (event) => {
  setPerson({
    ...person,
    [event.target.name]: event.target.value,
  });
};
```

```
return (
  <form>
    <label>
      Name:
      <input type="text" name="name" value={person.name} onChange={handleChange} />
    </label>
    <br />
    <label>
      Age:
      <input type="number" name="age" value={person.age} onChange={handleChange} />
    </label>
    <br />
    <label>
      Location:
      <input type="text" name="location" value={person.location} onChange={handleChange} />
    </label>
  </form>
);
```

Other Hooks - useEffect

useEffect: The useEffect hook is used to run side effects in a functional component. It allows you to perform operations that don't relate directly to rendering, such as fetching data or updating the title of a document. Here's an example of how you might use useEffect to log a message when a component is mounted:

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

const ComponentWithEffect = () => {
  useEffect(() => {
    console.log('Component has been mounted');
  }, []);

  return (
    <div>
      Component with effect
    </div>
  );
};

export default ComponentWithEffect;
```

Other Hooks - useRef

useRef: The useRef hook is used to access the value of a DOM element or a JavaScript object that doesn't change. It is often used to store a reference to a DOM element so that you can manipulate it directly in response to events. Here's an example of how you might use useRef to focus an input field when a component is mounted:

```
import React, { useRef, useEffect } from 'react';

const InputField = () => {
  const inputRef = useRef(null);

  useEffect(() => {
    inputRef.current.focus();
  }, []);

  return (
    <input type="text" ref={inputRef} />
  );
};

export default InputField;
```

Other Hooks - custom Hooks

Custom Hooks: Custom hooks are a way to extract logic from a component into a reusable function. They allow you to reuse stateful logic across many components without having to write the same code multiple times. Here's an example of how you might create a custom hook to manage the state of a toggle:

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

const useToggle = (initialValue) => {
  const [value, setValue] = useState(initialValue);

  const toggle = () => setValue(!value);

  return [value, toggle];
};

export default useToggle;
```

useState - update previous value

When updating the state, it is important to note that React batches updates and only updates the state after rendering has been completed. This means that if you need to update the state based on its previous value, you should pass a callback to the setter function instead of directly updating the state.

Here is how you update previous state value:

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

const Counter = () => {
  const [count, setCount] = useState(0);

  const handleClick = () => {
    setCount(prevCount => prevCount + 1);
  };

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={handleClick}>Increment</button>
    </div>
  );
};

export default Counter;
```

useEffect - how to cleanup

useEffect is a hook in React that allows you to perform side effects in your functional components. It runs after every render of your component, including the first render. In some cases, you may want to perform some cleanup when your component is unmounting, for example, to cancel a network request or remove an event listener.

In these cases, you should return a cleanup function from your useEffect callback.

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

const Example = () => {
  const [width, setWidth] = useState(window.innerWidth);

  useEffect(() => {
    const handleResize = () => setWidth(window.innerWidth);
    window.addEventListener('resize', handleResize);
    return () => {
      window.removeEventListener('resize', handleResize);
    };
  }, []);

  return (
    <div>
      <p>Window width: {width}</p>
    </div>
  );
};

export default Example;
```

Classwork about hooks

Task: Create a simple React component that displays the current time and updates it every second using the useState and useEffect hooks.

Instructions:

1) Create a new React project using npx create-react-app my-app.
2) Open the project in your preferred text editor.
3) In the src folder, create a new file called Clock.js.
4) In the Clock.js file, import the React and useState and useEffect hooks.
5) Create a new functional component called Clock that returns the current time.
6) In the Clock component, use the useState hook to store the current time.
7) In the Clock component, use the useEffect hook to update the time every second.
8) In the render function, display the current time using the state from the useState hook.
9) In the App.js file, import the Clock component.
10) In the App component, render the Clock component.

Classwork about hooks - solution

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

const Clock = () => {
  const [time, setTime] = useState(new Date().toLocaleTimeString());

  useEffect(() => {
    const intervalId = setInterval(() => {
      setTime(new Date().toLocaleTimeString());
    }, 1000);
    return () => clearInterval(intervalId);
  }, []);

  return (
    <div>
      <p>The current time is: {time}</p>
    </div>
  );
};

export default Clock;
```

Homework

Task: Create a simple React component that allows users to search for Github users and display their profile information.

1) Create a new React project using npx create-react-app my-app.
2) Open the project in your preferred text editor.
3) In the src folder, create a new file called GithubProfile.js.
4) In the GithubProfile.js file, import the React and useState and useEffect hooks.
5) Create a new functional component called GithubProfile that allows users to search for Github users and display their profile information.
6) In the GithubProfile component, use the useState hook to store the search input and profile information.
7) In the GithubProfile component, use the useEffect hook to fetch the Github user's profile information based on the search input.
8) In the render function, display a form with a text input for the user to search for Github users.
9) When the user submits the form, send a GET request to the Github API to retrieve the profile information for the given username. (https://api.github.com/users) **(optional - next lectures' topic)**
10) Display the profile information, including the user's avatar, name, username, and location, if available. **(optional)**
11) Handle errors if the API request fails or the user does not exist**. (optional)**