What is React Js?

React is a JavaScript library for building user interfaces (UIs) in web applications. It was developed by Facebook and released in 2013. React allows developers to build interactive and dynamic UIs by creating reusable components.

Key features and benefits of React include:

Component-Based Architecture: React encourages building UIs as reusable components, making it easier to manage and maintain code.

Virtual DOM: React uses a virtual representation of the actual DOM, which is a lightweight copy of the UI. This allows React to efficiently update and render changes to the UI, resulting in improved performance.

Declarative Syntax: React uses a declarative approach, where developers describe the desired UI state, and React takes care of updating the UI to match that state. This simplifies development and helps in code reusability.

JSX: React introduces JSX, a syntax extension that allows developers to write HTML-like code within JavaScript. JSX makes it easier to write and understand the structure of components.

React Components: React provides two types of components: functional components and class components. Functional components are JavaScript functions that return JSX, while class components are JavaScript classes that extend React.Component.

React Fragments: React Fragments allow grouping multiple elements without introducing an extra parent element. This helps to keep the DOM structure clean and organized.

React ecosystem: React has a vast ecosystem with tools and libraries that complement its functionality, such as React Router for routing, Redux for state management, and many more.

To learn more about React and get detailed documentation, you can visit the official React website at https://react.dev/learn. The current version of React as of my knowledge cutoff in September 2021 is version 18.

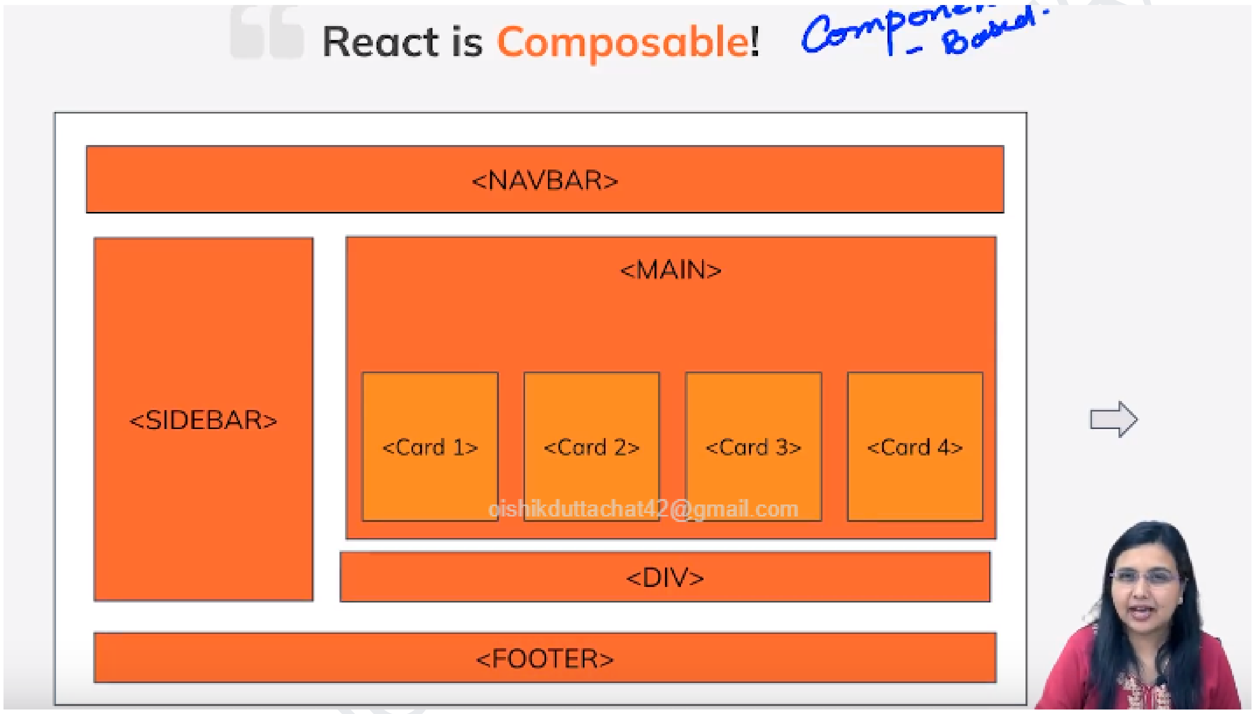
Regarding writing React code using JSX, you can use Babel (https://babeljs.io/) to transpile JSX code into regular JavaScript that the browser can understand.

For specific information on React Fragments, React components, and functional components, you can refer to the official React documentation available on the React website.

React is a javascript library for building web interfaces.

More Better while Creating the UX/UL. The library makes your life easier due to code Reusalblity.

The link is been given https://react.dev/learn .

1. The current version is V18
2. Initial launch in 2013
3. Handle by Meta
4. 

5. He helped us to Code Reusability.
6. He bullied the Single page Application with the Single Html Document making this a very Dynamic and Components based wise Design.

SPA AND MPA == Single Page Application and Multi-page Application overview.

Certainly! Here's an overview of Single Page Applications (SPAs) and Multi-page Applications (MPAs):

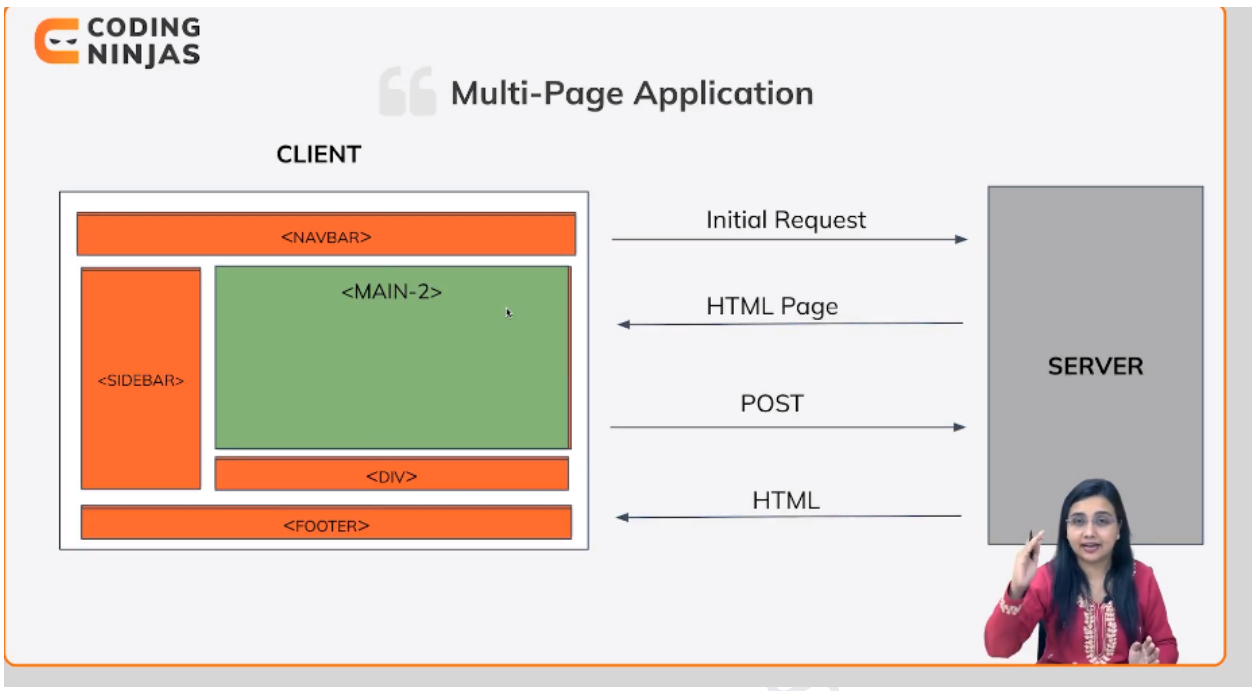
Single Page Applications (SPAs):
A Single Page Application is a web application that loads a single HTML page and dynamically updates its content using JavaScript. The initial HTML, CSS, and JavaScript resources are loaded once, and subsequent interactions with the application are handled by JavaScript, typically using AJAX (Asynchronous JavaScript and XML) to retrieve data from the server. SPAs provide a more fluid and seamless user experience as they avoid full-page reloads and provide real-time updates. Examples of SPAs include Gmail, Facebook, and Twitter.

Multi-page Applications (MPAs):
A Multi-page Application is a traditional web application where each interaction or navigation typically leads to a new HTML page being loaded from the server. MPAs follow a more traditional model, where each page represents a separate request/response cycle. When users navigate between different sections or pages, the server delivers the relevant HTML, CSS, and JavaScript for each page. Examples of MPAs include e-commerce websites like Amazon and news websites like BBC.

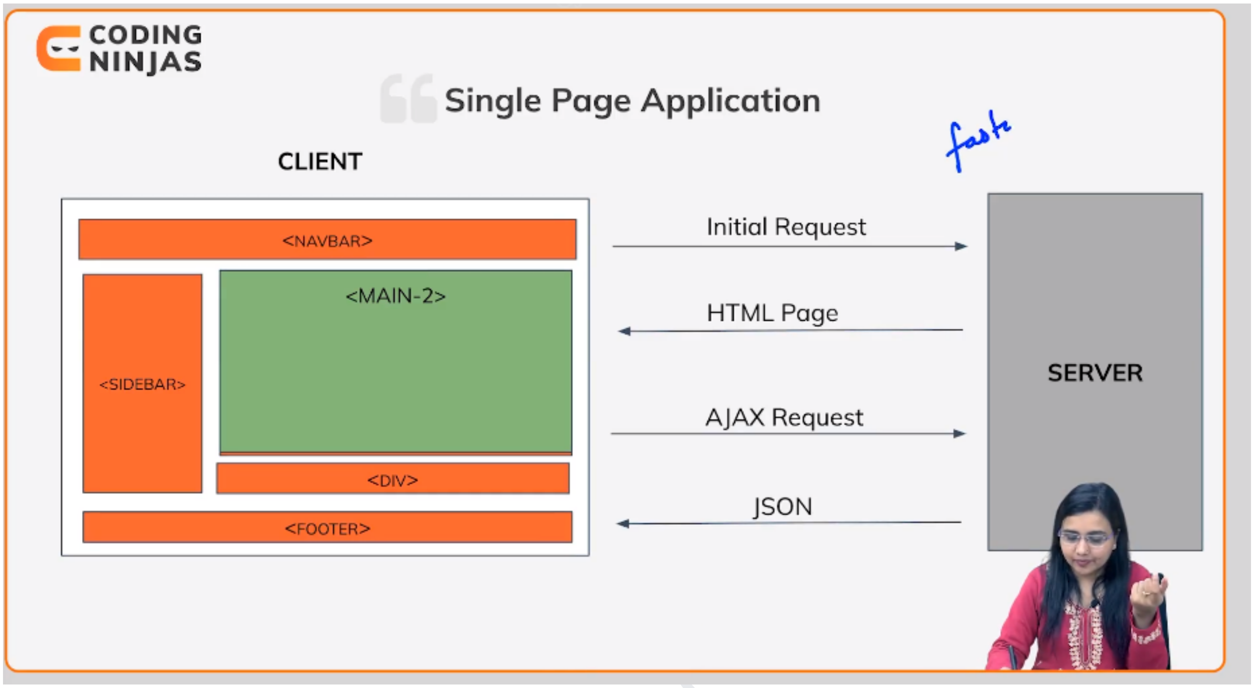
SPAs and MPAs differ in their approach to handling user interactions and content updates. SPAs offer a more interactive and dynamic experience, with faster content updates and reduced server load. They excel at providing a seamless user interface and are well-suited for complex web applications that require real-time updates. On the other hand, MPAs are suitable for applications that primarily focus on delivering content and do not require extensive client-side interactivity.

It's important to note that both SPAs and MPAs have their advantages and disadvantages, and the choice between them depends on the specific requirements of the application and the desired user experience.

Multi-Page-application:


CODING NINJAS
Multi-Page Application
CLIENT
<NAVBAR>
<MAIN-2>
<SIDEBAR>
<DIV>
<FOOTER>
Initial Request
HTML Page
POST
HTML
SERVER

Single Page Applications:
Single page applications are used to load on a single Html Document Which is been Extracted by javaScript via AJEX Apis.

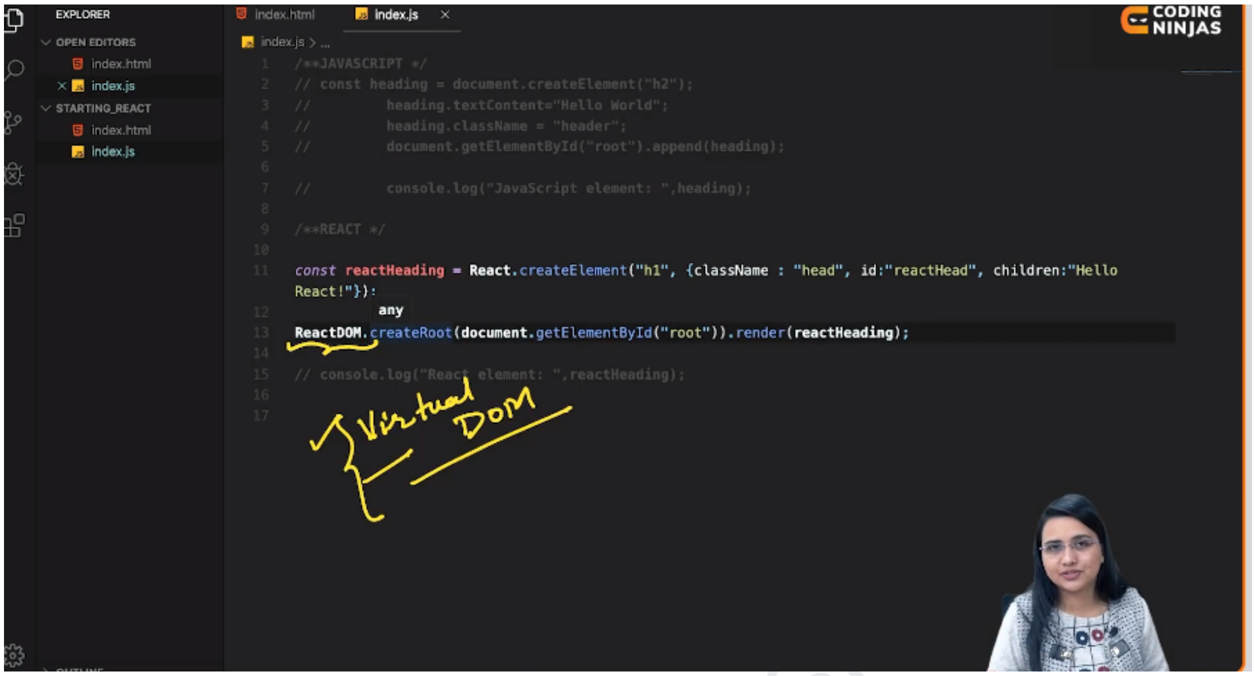


React Is Declarative

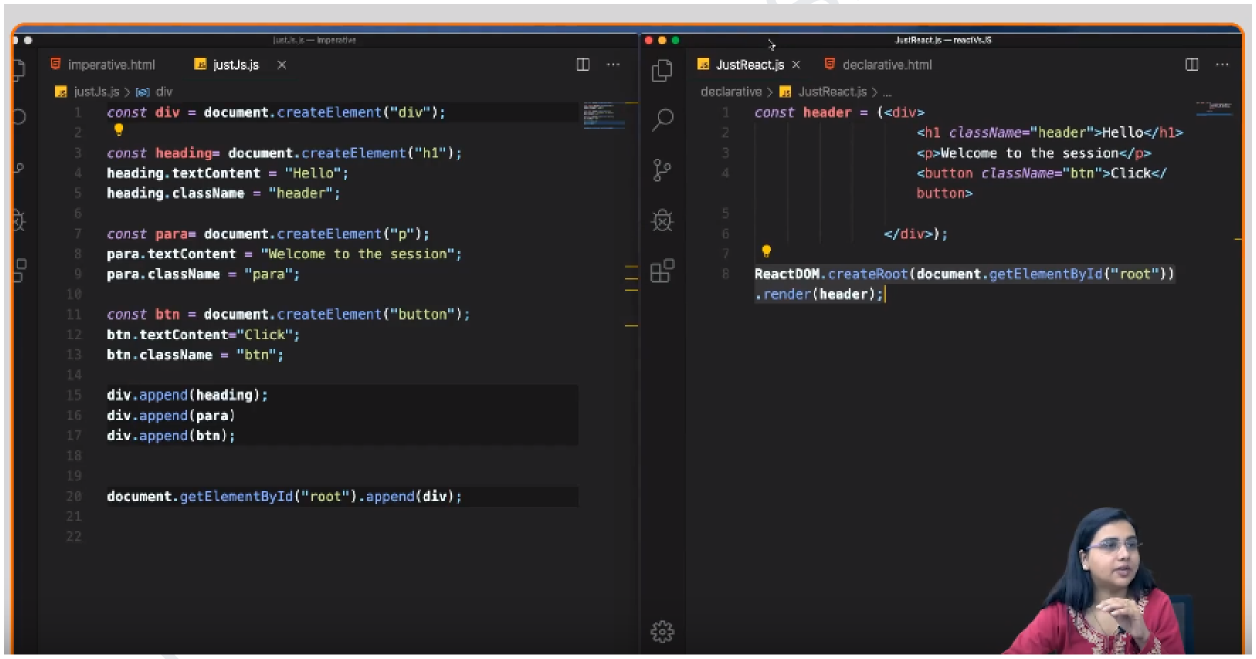
The issue is occurring when it's used to create the javascript Declaration using the javascript and building the UI/UX components in the Browser. Sometimes, the Synamics to build the Sctuture using the Valina javascript can be code reusability. react helps us to build the same thing in a very efficient way to do the same adjustment.

Import React Via CDN == Content Delivery Network

https://legacy.reactjs.org/docs/cdn-links.html



```
/**JAVASCRIPT */
// const heading = document.createElement("h2");
//          heading.textContent="Hello World";
//          heading.className = "header";
//          document.getElementById("root").append(heading);

//          console.log("JavaScript element: ",heading);

/**REACT */

const reactHeading = React.createElement("h1", {className : "head", id:"reactHead", children:"Hello
React!"});

ReactDOM.createRoot(document.getElementById("root")).render(reactHeading);

// console.log("React element: ",reactHeading);
```



```
const div = document.createElement("div");

const heading= document.createElement("h1");
heading.textContent = "Hello";
heading.className = "header";

const para= document.createElement("p");
para.textContent = "Welcome to the session";
para.className = "para";

const btn = document.createElement("button");
btn.textContent="Click";
btn.className = "btn";

div.append(heading);
div.append(para)
div.append(btn);


document.getElementById("root").append(div);
```

```
const header = (<div>
                    <h1 className="header">Hello</h1>
                    <p>Welcome to the session</p>
                    <button className="btn">Click</
                    button>

                </div>);

ReactDOM.createRoot(document.getElementById("root"))
.render(header);
```

What Is a Virtual Dom ??

The Virtual DOM (Document Object Model) is a concept and technique used in React and other JavaScript frameworks/libraries. It is a lightweight copy or representation of the actual DOM, which is a tree-like structure representing the HTML elements of a web page.

In React, when you create components and update the UI, you don't directly manipulate the actual DOM. Instead, React creates and maintains its own Virtual DOM. When changes are made to the state or props of a component, React updates the Virtual DOM to reflect these changes.

The Virtual DOM is a more efficient way to update the UI compared to directly manipulating the real DOM. Here's how it works:

Render: When a React component renders, it creates a Virtual DOM representation of the component's UI structure.

Diffing: When a component's state or props change, React creates a new Virtual DOM representation of the updated UI structure.

Diffing Algorithm: React performs a process called "diffing" to compare the previous Virtual DOM with the new Virtual DOM. It identifies the differences between the two representations.

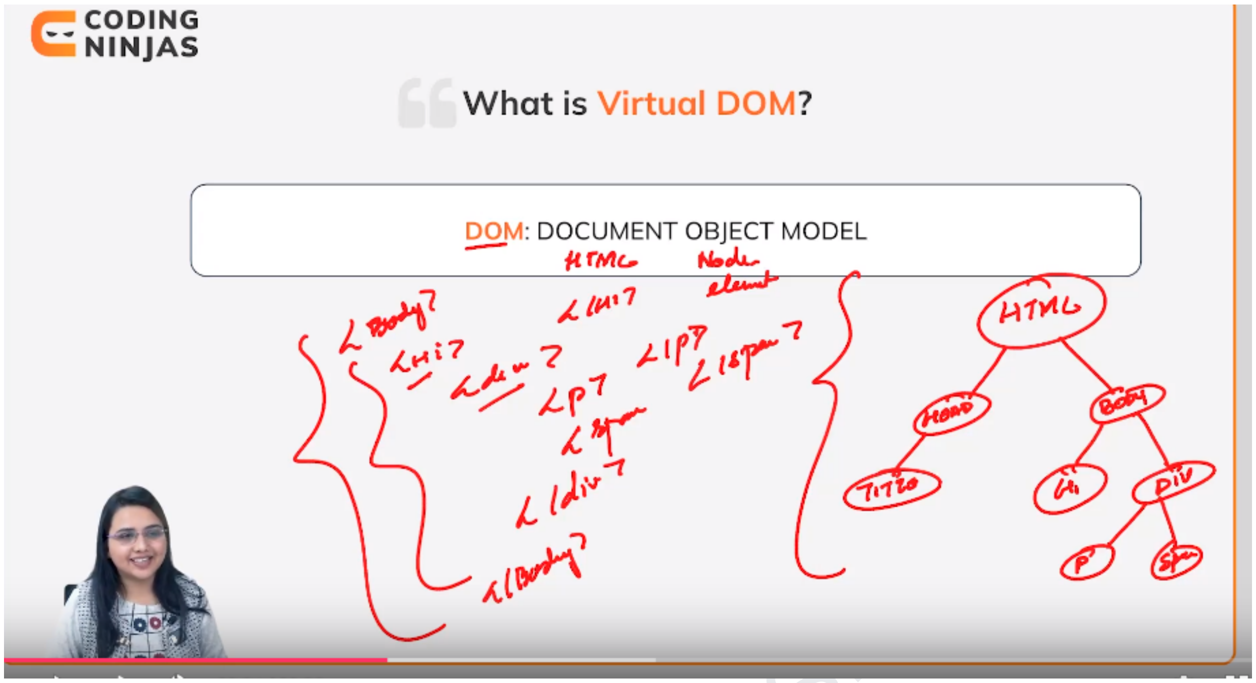
Update: React determines the minimal set of changes needed to update the actual DOM to match the new Virtual DOM.

Reconciliation: React efficiently applies these changes to the real DOM, updating only the necessary elements and minimizing the overall impact on performance.

By using the Virtual DOM and performing efficient diffing and update operations, React reduces the number of actual DOM manipulations and improves the overall performance of the application. It allows for efficient rendering of complex UI structures and provides a smoother user experience.

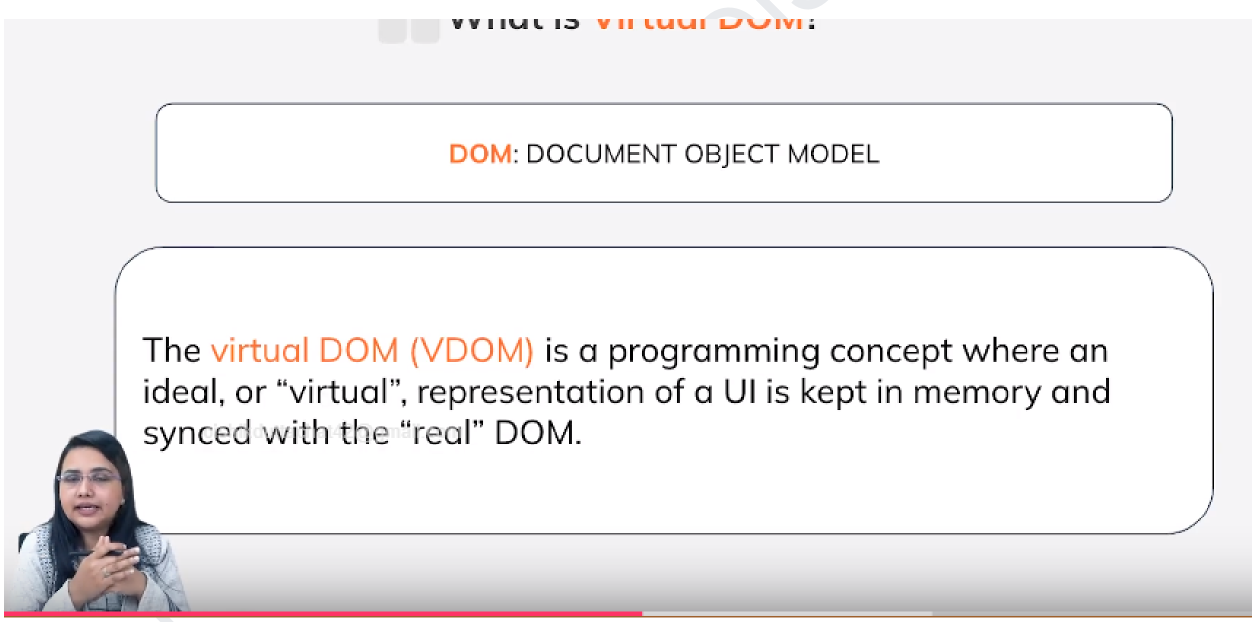
It's important to note that the concept of a Virtual DOM is not exclusive to React. Other frameworks and libraries, such as Vue.js, also utilize a similar technique to optimize UI updates.

Dom = Document Object Models
Actual Documentation of an HTML Page



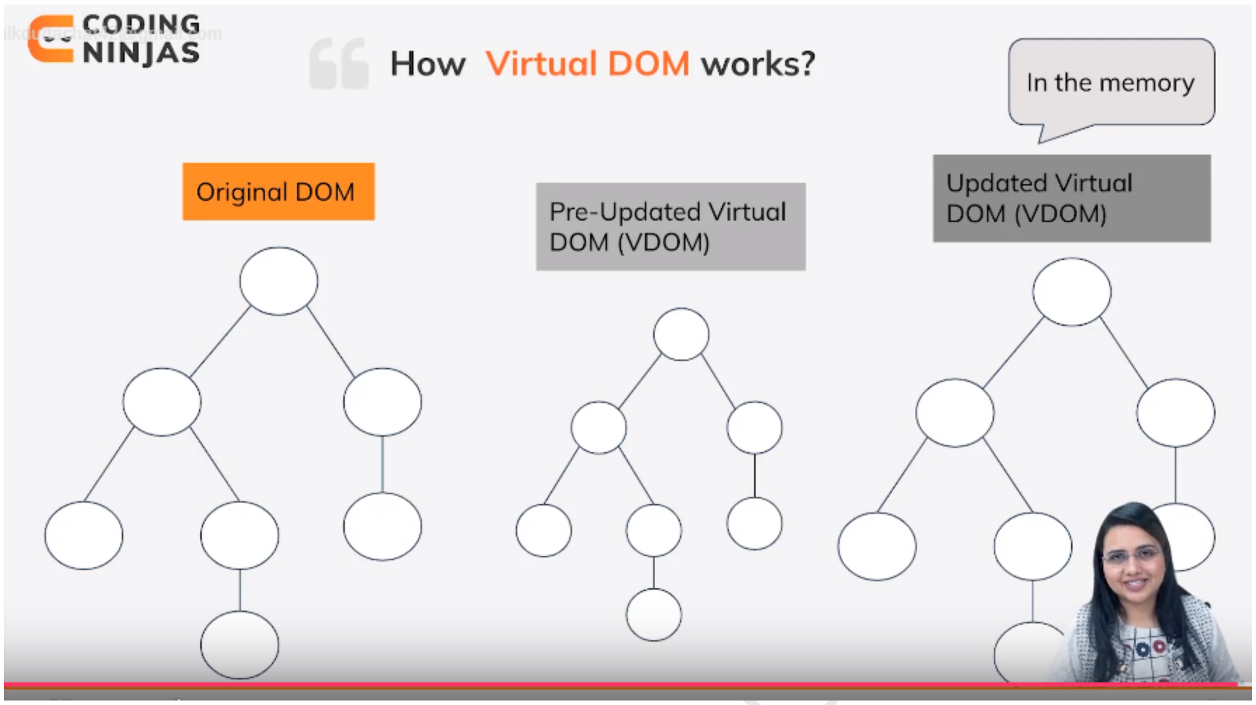
What is Virtual DOM?

DOM: DOCUMENT OBJECT MODEL



DOM: DOCUMENT OBJECT MODEL

The virtual DOM (VDOM) is a programming concept where an ideal, or “virtual”, representation of a UI is kept in memory and synced with the “real” DOM.



Writing code in the JSX

1. Understanfing babeel. https://babeljs.io/docs/

React Fragments:

React Fragments are a feature in React that allow you to group multiple elements together without introducing an additional parent element. Fragments are useful when you want to return multiple elements from a component, but you don't want to add an extra wrapper element to the DOM.

Before React Fragments were introduced, if you wanted to return multiple elements from a component, you had to wrap them in a single parent element. For example:

~~~
```jsx
render() {
  return (
    <div>
      <h1>Title</h1>
      <p>Content</p>
    </div>
  );
~~~

```
}
```

In the above example, a `<div>` is used as the wrapper element for the `<h1>` and `<p>` elements. However, sometimes you may not want to introduce an additional `<div>` or any other element just for the sake of wrapping.

With React Fragments, you can achieve the same result without introducing a wrapper element. Here's how you can use React Fragments:

```jsx
render() {
  return (
    <>
      <h1>Title</h1>
      <p>Content</p>
    </>
  );
}
```

In the updated example, the `<>` (empty angle brackets) are used to represent the React Fragment. You can place multiple elements inside the Fragment without introducing an extra DOM node. The Fragment itself won't be rendered in the DOM. It acts as a wrapper in the virtual DOM during the rendering process but doesn't create an actual element in the resulting HTML structure.

React Fragments help to keep the DOM structure cleaner and more concise when returning multiple elements from a component. They are especially useful in scenarios where adding an extra wrapper element would interfere with CSS styles or cause unnecessary nesting.

In addition to using the `<>` syntax, you can also use the `<React.Fragment>` syntax to achieve the same result.

```jsx
render() {
  return (
    <React.Fragment>
      <h1>Title</h1>
      <p>Content</p>
    </React.Fragment>
  );
}
```

Both syntax variations of React Fragments are functionally equivalent, and you can choose the one that best fits your coding style or project requirements.

React Components

In React, components are the building blocks of user interfaces. They are reusable, self-contained units of code that encapsulate a specific set of functionality and UI elements. React components allow you to break down your user interface into smaller, modular pieces, making your code more organized, maintainable, and reusable.

There are two main types of components in React:

1. Functional Components:
Functional components are JavaScript functions that accept inputs called "props" (short for properties) and return React elements to describe what should be rendered on the screen. Functional components are also referred to as "stateless components" because they don't have their own internal state. They are simpler and easier to understand and test compared to class components.

Here's an example of a functional component:

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

In the above example, `MyComponent` is a functional component that accepts a `name` prop and returns an `<h1>` element displaying a greeting.

2. Class Components:
Class components are ES6 classes that extend the `React.Component` class. They have their own state and can contain lifecycle methods, event handlers, and other functionalities. Class components are used when you need to manage state or require more complex logic.

Here's an example of a class component:

```jsx
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0,
    };
  }

  render() {
    return (
      <div>
        <h1>Count: {this.state.count}</h1>
        <button onClick={() => this.setState({ count: this.state.count + 1 })}>
          Increment
        </button>
      </div>
    );
  }
}
```

In the above example, `MyComponent` is a class component that manages its own state using the `state` property. It renders a `<div>` containing a count value and a button that increments the count when clicked.

Both functional components and class components can be used interchangeably in React. However, with the introduction of React Hooks in React 16.8, functional components became more powerful and gained the ability to manage state and use lifecycle methods, making them the preferred choice in many cases.

When creating components, it's important to consider reusability, maintainability, and separation of concerns. By breaking down your UI into small, reusable components, you can create a modular and flexible codebase that is easier to understand and maintain.

Functional Components Using the Arrow Approach

Functional components can also be defined using arrow function syntax. This is known as the "arrow approach" or "arrow function components". Arrow function components provide a more concise and expressive way to define functional components in React.

Here's an example of a functional component using the arrow approach:

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

In the above example, `MyComponent` is defined as an arrow function that accepts `props` as its parameter. It returns an `<h1>` element displaying a greeting with the `name` prop.

Arrow function components are especially useful for simple components that don't require complex logic or state management. They have a more compact syntax and can make your code look cleaner and more concise.

You can also make the arrow function component even more concise by utilizing implicit return. If the component body consists of a single expression, you can omit the curly braces `{}` and the `return` keyword, and the expression will be automatically returned. Here's an example:

```jsx
const MyComponent = (props) => <h1>Hello, {props.name}!</h1>;
```

In this updated example, the arrow function component is defined in a single line, and the `<h1>` element is automatically returned without the need for explicit `return` statement.

Arrow function components have become increasingly popular due to their simplicity and readability. They are widely used in React applications, especially when creating smaller, functional components.

It's important to note that arrow function components do not have their own `this` context, so you cannot access `this.props` or use lifecycle methods like in class components. However, you can still access the `props` object directly as a parameter in the arrow function.

Overall, the arrow approach is a convenient and concise way to define functional components in React, providing a more modern and streamlined syntax for component creation.