

Introduction to React

React

React is a JavaScript library for building user interfaces. JavaScript libraries like React are collections of prewritten code snippets that can be used and reused to perform common JavaScript functions, helps in faster development with fewer vulnerability to have errors. UI(User Interface) is built from small units like buttons, text, and images. Everything on the screen can be broken down into components, from websites to phone apps. React lets you combine them into reusable, nestable components.

History of React

- React was originally created by Jordan Walke, a software engineer at Facebook. But today, it is maintained by Meta(formerly Facebook) and a community of over a thousand open-source developers.
- It was first deployed on Facebook's News Feed in 2011 and later on Instagram in 2012. It was open-sourced at JSConf US in May 2013.
- Some of the major companies that currently use React include Netflix, Facebook, Instagram, Airbnb, Reddit, Dropbox, and Postmates.
- Current(Latest) version of React is v18.

Why React?

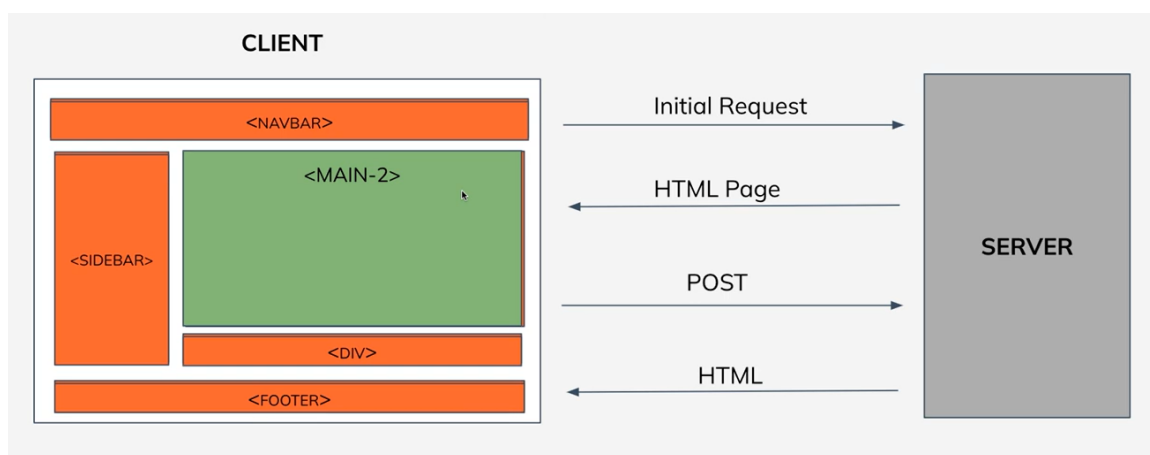
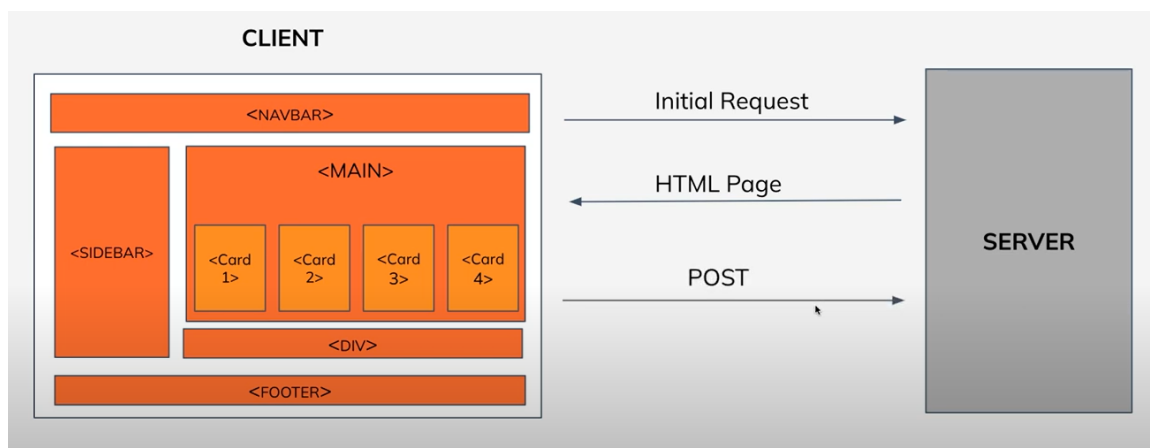
- **React is Composable:** Components are the building blocks of any React application, and a single app usually consists of multiple components. These components have their logic and controls, and they can be reused throughout the application, which in turn dramatically reduces the application's development time.
- **Faster performance:** React uses Virtual DOM, thereby creating web applications faster. Virtual DOM compares the components' previous states

and updates only the items in the Real DOM that were changed, instead of updating all of the components again, as conventional web applications do.

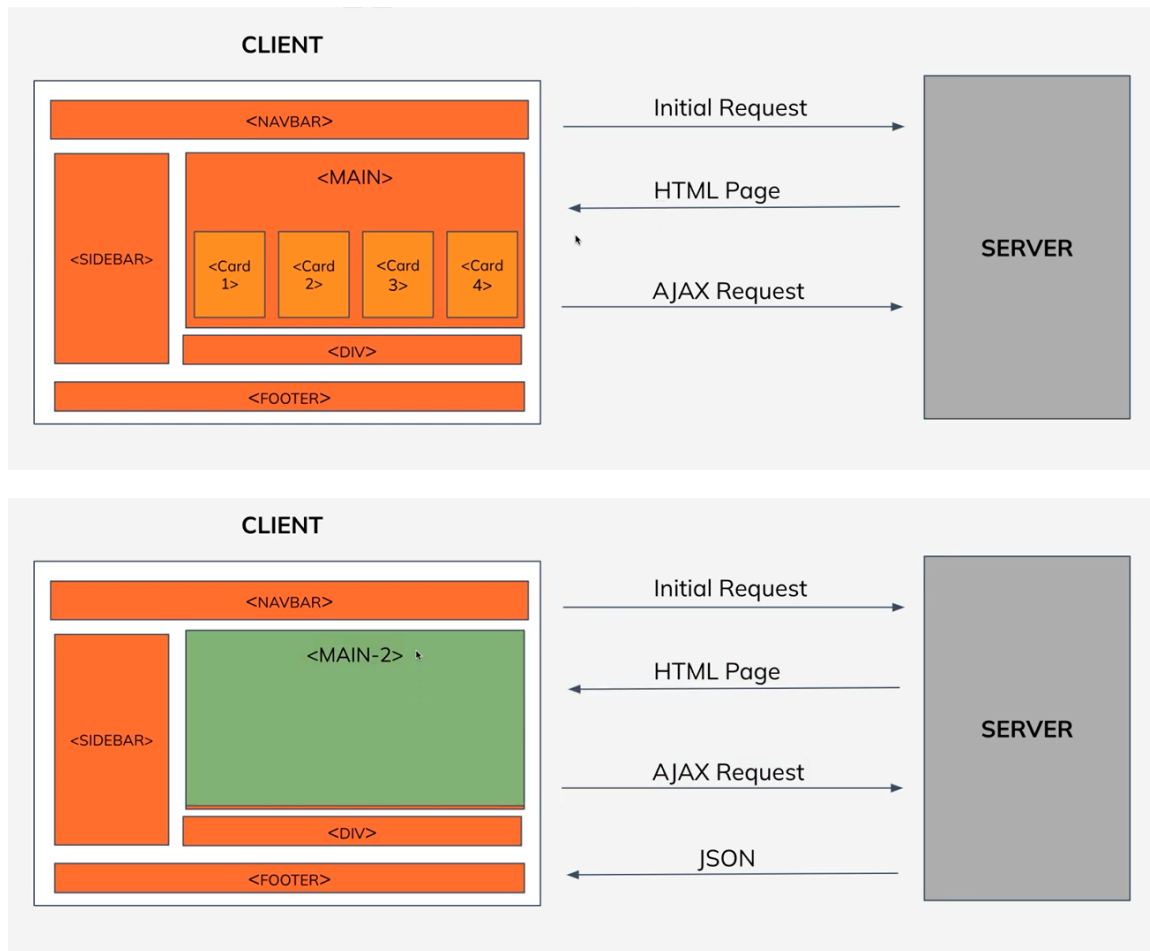
- **React is Declarative:** React is easy to learn, mostly combining basic HTML and JavaScript concepts with some beneficial additions. Still, as is the case with other tools and frameworks, you have to spend some time to get a proper understanding of React's library.
- **Dedicated tools for easy debugging:** Facebook has released a Chrome extension that can be used to debug React applications. This makes the process of debugging React web applications faster and easier.

Multi-Page Applications vs Single-Page Applications

Multi-Page Application (MPA) is a traditional implementation of a web application that reloads the whole page when a user interacts with the app.



Single-Page Application (SPA) is a web application that loads a single document(HTML) and updates the parts of the document using APIs(AJAX).



Difference between MPA and SPA

| | Multi-Page Application | Single-Page Application |
|----|--|---|
| 1. | In MPAs, content is constantly loaded, which increases the load on your server. This can adversely affect web page speed and overall system performance. | SPAs provide increased content load speed because they do not have many pages and load content at once. |
| 2. | Multi-page applications have more features than single-page applications. Therefore, more effort and resources are required to make | Single-page app development is easy because you need to create fewer pages, create less functionality, and test and display less content. |

| | | |
|----|---|---|
| | them. Development time increases in proportion to the number of pages created and the activity to be executed. | |
| 3. | Multi-page applications are more SEO-friendly than single-page applications. Their content is constantly updated. In addition, they have many pages for adding various keywords, images, and meta tags. | Single-page app developers have trouble indexing a website properly. Multi-page applications are more SEO-friendly than single-page and achieve high search rankings. |
| 4. | It is difficult to maintain and is not budget-friendly. | It is easy to maintain at a low cost. |
| 5. | It always requires an internet connection as it does not load all the data at once. | It has the ability to work offline if there are some problems with the internet connection, as it loads all the data at once. |

How to include Javascript in HTML?

You can include JavaScript in your HTML in two ways:

- Embedding code in your HTML file using `<script>` tag
- Including it as a link to an external file

Embedding Code

You can add JavaScript code in an HTML document by employing the dedicated HTML tag `<script>` that wraps around JavaScript code. The `<script>` tag can be placed in the `<head>` section of your HTML or in the `<body>` section, depending on when you want the JavaScript to load.

```
<!DOCTYPE html>
<html>
```

```
<head>
  <title>Document</title>
  <script>
    document.write("Welcome to the session");
  </script>
</head>

<body>
  <h1>Hello</h1>
</body>

</html>
```

External File

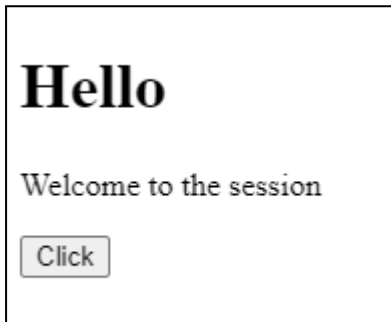
To include an external JavaScript file, we can use the script tag with the attribute src. The value for the src attribute should be the path to your JavaScript file. This script tag should be included between your HTML document's <head> tags. When JavaScript files are cached, pages load more quickly.

```
<script type="text/javascript" src="path-to-javascript-file.js"></script>
```

How React is declarative?

Declarative programming is when you say what you want to do, and describe the final state of the desired UI. Imperative programming is when you say how to get what you want and provide step-by-step DOM mutations until you reach the desired UI. Javascript is an imperative Language whereas React is a declarative language.

For eg: For the following output, You need to add an element to the DOM imperatively using JavaScript. As your app gets bigger, with more DOM elements you being created, this can become hard to maintain. But, React it performs all of the JavaScript/DOM steps as per the declared code to get us to our desired result. It abstracts away all the nuts and bolts of how the DOM renders these elements. In your code you tell your page "Look like this" and you'll get that result. Declarative programming is much easier to read and figure out what is going on in your code. That makes it easier to debug and easier for other devs to work on.



Javascript CODE

index.html

```
<html>
  <head>
    <title> HTML </title>
  </head>
  <body>
    <div id="root"> </div>
    <script src="script.js" type="javascript"> </script>
  </body>
</html>
```

script.js

```
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);
```

React CODE

index.html

```
<html>
  <head>
    <title> Document </title>
    <script crossorigin
src="https://unpkg.com/react@18/umd/react.development.js"></script>
    <script crossorigin
src="https://unpkg.com/react-dom@18/umd/react-dom.development.js"></script>
    <script src =
"https://unpkg.com/@babel/standalone/babel.min.js"></script>
  </head>

  <body>
    <div id="root"> </div>
    <script src="scripts.js" type="text/babel"> </script>
  </body>

</html>
```

script.js

```
const heading = React.createElement("h1", null, "Hello");
const para = React.createElement("p", {className:"para"}, "Welcome to the
session");
const btn = React.createElement("button", {className:"btn"}, "Click");

const div = React.createElement("div", {className:"App", children :
[heading,para,btn]});

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

script.js (Using JSX)

```
const header = (
  <div>
```

```
    <h1 className="header">Hello</h1>
    <p className="para">Welcome to the session</p>
    <button className="btn">Click</button>
  </div>
);

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

Creating First React Element

To include React in a simple web page, CDN(Content Delivery Network) can be used. You need to create a new HTML file and include the CDN links of the following:

React: React-script-tag is an npm package that provides a React `<script>` tag which supports universal rendering. With this library, we can create react components, that is, a plain javascript object with some properties.

```
<script crossorigin
src="https://unpkg.com/react@18/umd/react.development.js"></script>
```

React-DOM: React-DOM basically converts the javascript object returned by React script tag to HTML nodes that can be rendered in the browser.

```
<script crossorigin
src="https://unpkg.com/react-dom@18/umd/react-dom.development.js"></script>
```

Babel: JSX files are not understandable by the browser. It is a tool that converts JSX files to simple javascript code that the browser understands. Moreover, it also converts ES6 and ES5 code to javascript code.

```
<script src = "https://unpkg.com/@babel/standalone/babel.min.js"></script>
```

Now, we are ready to use React library in our webpage. So, introduce a div tag with an id "root" in the body. We call this a "root" DOM node because everything inside it will be managed by React DOM.

```
<div id="root" type="text/babel"> </div>
```


Now, create a script section at the end of the document. Then, pass the DOM element to `ReactDOM.createRoot()`, and then to `root.render()` to render an HTML element dynamically.

```
<script language="JavaScript">
  heading = React.createElement("h1", null, "Hello");
  ReactDOM.createRoot(document.getElementById("root")).render(heading);
</script>
```

After that, use the **live server** extension of VS Code to serve the webpage and see the output.

React.createElement

A React element describes what the real Document Object Model (DOM) element should look like. React.js uses virtual DOM to design the UI and interact with the browser. It is made up of react elements that seem similar to HTML elements but are JavaScript objects. In simple words, react elements are the instructions for how the browser DOM should be created. We can create the react elements using the below syntax by embedding HTML elements in JavaScript to display the content on the screen.

```
React.createElement(type,{props},children);
```

It takes three arguments. They are:

- **type**: specifies the type of the HTML element (h1, p, button).
- **props**: specifies properties of the object ({style:{size:10px}} or event handlers, classNames,etc).
- **children**: anything that needs to be displayed on the screen.

React.createRoot

It creates a React root for the supplied container and returns the root. The root can be used to render a React element into the DOM with render:

```
const root = createRoot(container);
root.render(element);
```

Root.render

React elements are immutable. Once you create an element, you can't change its children or attributes. The only way to update the UI is to create a new element, and pass it to `root.render()`.

```
root.render(element, container element);
```

It takes two arguments:

- **element:** The element that needs to be rendered in the DOM.
- **container element:** It specifies where to render the element in the DOM.

Note: For running your React.js project, there are two modes available – development and Reactjs build production. During the **development phase**, we will be running our code locally using the development mode where React provides us with many helpful warnings and tools for easily detecting and fixing problems in our application code and eliminating potential bugs. But in **production mode**, the warning messages and other features present in development mode for debugging are suppressed. It minifies your code, optimizes assets, and produces lighter-weight source maps. As a result, the bundle size is drastically reduced, improving page load time. React recommends utilizing the production mode while deploying the application.

Understanding Real DOM and Virtual DOM

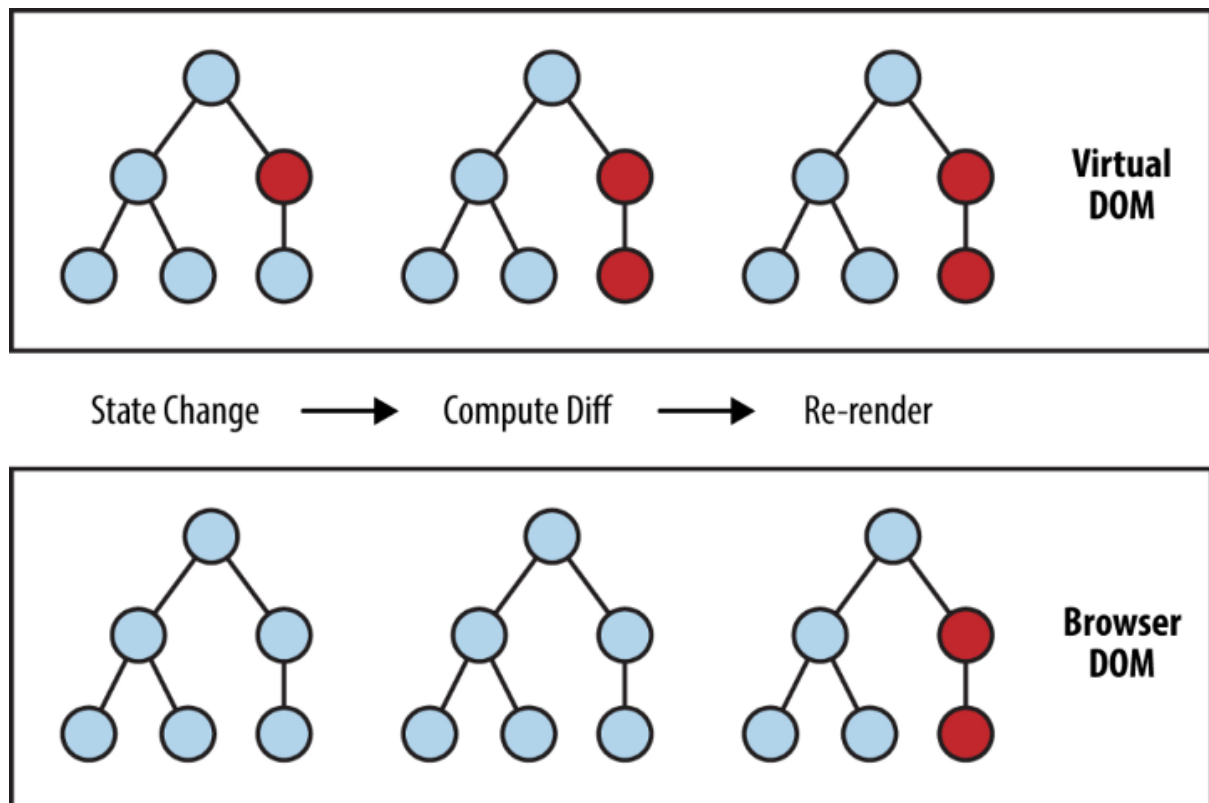
Real DOM

DOM stands for “Document Object Model”. The DOM in simple words represents the UI of your application. Every time there is a change in the state of your application UI, the updated element and its children have to be re-rendered to represent that change. But frequently manipulating the DOM affects performance, making it slow. Therefore, the more UI components you have, the more expensive the DOM updates could be, since they need to be re-rendered for every DOM update.

Virtual DOM

The virtual DOM is only a virtual representation of the DOM. Every time the state of our application changes, the virtual DOM gets updated instead of the real DOM. If

the state of any of these elements changes, a new virtual DOM tree is created. This tree is then compared or “diffed” from the previous virtual DOM tree. Once this is done, the virtual DOM calculates the best possible method to make these changes to the real DOM. This ensures that there are minimal operations on the real DOM. Hence, reducing the performance cost of updating the real DOM.



React compares the Virtual DOM with Real DOM. It finds out the changed nodes and updates only the changed nodes in Real DOM leaving the rest nodes as it is. This process is called **Reconciliation**. Diffing algorithm is a technique of reconciliation that is used by React.

JSX

JSX, or JavaScript XML, is an extension to the JavaScript language syntax. Similar in appearance to HTML, JSX provides a way to structure component rendering using syntax familiar to many developers. React components are typically written using JSX, although they do not have to be (components may also be written in pure

JavaScript). JSX is similar to another extension syntax created by Facebook for PHP called XHP.

- **Why is a class not used as an attribute in JSX?** We cannot use class attributes in script tags. Instead of this, we use it because the class is a reserved keyword in javascript.
- **Using javascript variables in JSX:** We can use variable names instead of static text by creating variables. We can add them in a JSX file using {variable_name}.

For Example

```
const name="Joy";
const header = () => (
  <>
    <h1 className="header">Hello {name}</h1>
    <p className="para">Welcome to the session</p>
    <button className="btn">Click</button>
  </>
);
```

Here, a constant variable is created named “name” and in the h1 tag, it is used in curly braces to display the name saved in the “name” variable.

Babel

React uses JSX syntax and JSX files are not understandable by the browser. Babel is a transpiler i.e. it converts the JSX to vanilla JavaScript. It can also convert the latest version of JavaScript code into the one that the browser understands.

| | |
|--|--|
| <pre>1 const header = () => (2 <div> 3 <h1 className="header">Hello</h1> 4 <p className="para">Welcome to the session</p> 5 <button className="btn">Click</button> 6 </div> 7); 8 9 ReactDOM.createRoot(document.getElementById("root")).render(header);</pre> | <pre>1 const header = () => /*#__PURE__*/React.createElement("div", null, /*#__PURE__*/React.createElement("h1", { 2 className: "header" 3 }, "Hello"), /*#__PURE__*/React.createElement("p", { 4 className: "para" 5 }, "Welcome to the session"), /*#__PURE__*/React.createElement("button", { 6 className: "btn" 7 }, "Click")); 8 ReactDOM.createRoot(document.getElementById("root")).render(header);</pre> |
|--|--|

React Fragments

In React, when a component returns multiple elements, we must wrap them in a container element like a `div` for the code to work. While this is fine, it may however cause unintended issues in our components. React fragments serve as a cleaner alternative to using unnecessary `divs` in our code. Fragments let you group a list of children without adding extra nodes to the DOM.

Here is a code snippet for your reference:

```
const header = () => (  
  <React.Fragment>  
    <h1 className="header">Hello</h1>  
    <p className="para">Welcome to the session</p>  
    <button className="btn">Click</button>  
  </React.Fragment>  
)
```

You can create a React fragment using `<React.Fragment></React.Fragment>`.

You can also use the shorthand syntax to wrap components using an empty HTML element like syntax, `<></>`.

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

Components in React

Components are independent and reusable codes. They work for the same purpose as JavaScript functions but work independently and restore HTML with the `render()` function. In simple words, react is like the lego game, and here components are bricks of lego that are used to build different applications. Components are of two types:

- **Class components:** The class component name must start with a capital letter. This component should contain `React.Component` statement, which creates the inheritance for `React.Component` and gives your component

access to the functions of `React.Component`. The component also requires a `render()` method, which provides HTML.

- **Function components:** A functional component is just a plain JavaScript function that accepts props (arguments passed into React components) as an argument and returns a React element. There is no render method used in functional components.

Arrow Function in React

An arrow function expression is a JavaScript expression that has a shorter syntax than the function keyword. It is designed for situations where you want to create a one-line anonymous function expression, like in event handlers.

The arrow function expression syntax for the above function is as follows –

```
let handleClick = (parameter) => { // code };
```

An arrow function expression always has a single parameter, following the `=>` token, and then an expression or statement within parentheses that follow the return value using the parameter.

There are several benefits to using arrow functions in ReactJS.

- They are much simpler to write and understand than traditional function expressions. This can make your code more readable and easier to debug.
- Arrow functions do not create a new scope, so they can be used in ReactJS without polluting the global scope.
- Arrow functions can be used as arguments to other functions, which can make your code more flexible and expressive.

Summarising it

Let's summarise what we have learned in this module:

- Learned about React, its history, and its features.
- Learned about Multi-Page and Single Page Applications.
- Learned about Declarative and Imperative Languages.
- Learned how to create elements in React.

- Learned about Real DOM and Virtual DOM.
- Learned about JSX.
- Learned about Babel.
- Learned about React Fragments
- Learned about types of components.
- Learned how to create components using arrow functions.

Some Additional Resources: To explore more

- React Official Documentation: [link](#)
- MPA vs SPA: [link](#)
- Rendering Elements: [link](#)
- Reconciliation: [link](#)
- Babel Documentation: [link](#)
- Babel Try it out: [link](#)
- React Fragments: [link](#)
- Understanding Fragments: [link](#)

Learning JSX

What is JSX?

JSX stands for JavaScript XML. It is syntactic sugar for creating React Elements. It is a syntax extension to JavaScript. It is used with React to describe what the UI should look like. JSX may remind you of a template language, but it comes with the full power of JavaScript. It produces React “elements.” Example:

```
const element = <h1>Hello world!</h1>;
```

Why JSX

JSX allows us to write HTML elements in JavaScript and place them in the DOM without any `createElement()` and `appendChild()` methods. JSX converts HTML tags into react elements. You are not required to use JSX, but JSX makes it easier to write React applications. Example:

Without JSX

```
const myElement = React.createElement('h1', {}, 'I do not use JSX!');
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(myElement);
```

With JSX

```
const myElement = <h1>I Love JSX!</h1>;
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(myElement);
```


React embraces the fact that rendering logic should be coupled with UI logic. It is helpful as a visual aid when working with UI inside the javascript. React separates concerns instead of separating technologies. It allows React to show more valuable errors and warning messages.

Babel is a JavaScript Compiler that allows us to use future JavaScript in today's browsers. Simply put, it can convert the latest version of JavaScript code into the one the browser understands.

Babel can convert JSX syntax. It is therefore used to convert JSX expressions into JavaScript code browsers can understand.

You can try it yourself. [Here is the link.](#)

React Fragments

It is a common pattern in React that a component returns multiple elements. Fragments let you group a list of children without adding extra nodes to the DOM. We know that we use the render method inside a component whenever we want to render something to the screen. We may generate single or multiple elements. However, rendering multiple elements will require a 'div' tag around the content as the render method will only render a single root node inside it at a time. Example:

```
function App() {  
  return (  
    <div>  
      <h2>Hello</h2>  
      <p>How are you doing?</p>  
    </div>  
  );  
}
```

Reason to use Fragments: As we saw in the above code, when we are trying to render more than one root element, we have to put the entire content inside the 'div' tag, which is not a good approach because no one wants to include an extra div element if that is not required in the code. Hence

Fragments were introduced, and we use them instead of the extraneous 'div' tag. Example:

```
function App() {  
  return (  
    <React.Fragment>  
      <h2>Hello</h2>  
      <p>How are you doing?</p>  
    </React.Fragment>  
  );  
}
```

Shorthand Fragment: The output of the first code and the code above is the same, but the main reason for using it is that it is a tiny bit faster when compared to the one with the 'div' tag inside it, as we didn't create any DOM nodes. Also, it takes less memory. Another shorthand also exists for the above method in which we use '<>' and '</>' instead of the 'React.Fragment'.

Example:

```
function App() {  
  return (  
    <>  
      <h2>Hello</h2>  
      <p>How are you doing?</p>  
    </>  
  );  
}
```

JSX Expression

With JSX, you can write expressions inside curly braces { }. JSX Expressions, written inside curly brackets, allow only things that evaluate some value like string, number, array map method, etc. The expression can be a React variable, property, or any other valid JavaScript expression. JSX will execute the expression and return the result.

Example:

```
function App() {  
  var name = "John Doe";  
  let age = 4;  
  const header = <h2>This is Header</h2>;  
  
  return (  
    <div>  
      {header}  
      <p>My name is {name}</p>  
      <p>My age is {age}</p>  
    </div>  
  );  
}
```

Returning two items at a time is invalid in Javascript. Since JSX is still JavaScript, its return can only handle one expression therefore, to return more than one element, we need to wrap it in another (an outer element), which most commonly will be a `<div></div>`.

For eg, The code below will throw an error because JSX expressions must have only one parent element.

```
function App() {  
  return (  
    <div>Hello</div>  
    <div>World</div>  
  )  
}
```

Comments in JSX

To add JavaScript code inside JSX, we need to write it in curly brackets. To add a comment for that code, then you have to wrap that code in JSX expression syntax inside the `/*` and `*/` comment symbols like this:

```
{/* <p>This is some text</p> */}
```

Null/Undefined/Boolean in JSX

JSX ignores null, undefined, and Booleans (false, true). They don't render as JSX is syntactic sugar for `React.createElement(component, props, ...children)`.

For Eg: These JSX expressions will all render the same thing:

```
<div />
<div></div>
<div>{false}</div>
<div>{null}</div>
<div>{undefined}</div>
<div>{true}</div>
```

If you want a value like false, true, null, or undefined to appear in the output, you have to convert it to a string first:

```
<div>
  My JavaScript variable is {String(myVariable)}.
</div>
```

Functions in JSX

A function can be defined and used by the expression. The component takes the function's output to produce content. We can simply define a function inside our component, and we can call it inside the `return()` method. We can invoke the function by adding parentheses `()` at the end.

For Eg:

```
const App = () => {
  const a = 4;
  const b = 6;
  const sum = (a, b) => a + b;
  return (
    <h2>Sum of {a} and {b} is: {sum(a, b)}</h2>
  );
};
```

Arrays in JSX

Arrays can be rendered inside the JSX Expressions easily using curly braces similar to any variable. For example:

```
const arr = [1, 2, 3, 4, 5];
const App = () => {
  return ( <h2>Array is: {arr}</h2> );
};
```

Output: Array is: 12345;

Iterating over the array

We usually use the map function to iterate through the array in React. The map is a JavaScript function that can be called on any array. With the map function, we map every element of the array to the custom components in a single line of code. That means there is no need to call components and their props array elements repeatedly.

The `.map()` method allows you to run a function on each item in the array, **returning a new array**. In React, map() can be used to generate lists and render a list of data to the DOM. To use the map() function, we attach it to an array we want to iterate over.

Example:

```
const myArray = ['apple', 'banana', 'orange'];
const myList =
myArray.map((item, index) => <p key={index}>{item}</p>)
```

Note: Keys allow React to keep track of elements. This way, if an item is updated or removed, only that item will be re-rendered instead of the entire list. Keys need to be unique to each sibling.

Generally, the key should be a unique ID assigned to each item. As a last resort, you can use the array index as a key.

Filter()

The `filter()` method is an iterative method. It calls a provided `callbackFn` function once for each element in an array and constructs a new array of all the values for which `callbackFn` returns a truthy value. Array elements that do not pass the `callbackFn` test are not included in the new array.

The JavaScript Array `filter()` Method is used to create a new array from a given array consisting of only those elements from the given array which satisfy a condition set by the argument method.

Example:

```
function isEven(value) {  
    return value % 2 == 0;  
}  
  
var originalArr = [11, 98, 31, 23, 944];  
var newArr = originalArr.filter(isEven);  
console.log(newArr);
```

Output: [98,944]

The function passed in the filter function checks whether each element is even or odd. The filter function then returns only the truthy element; hence, the new array returned consists of only 2 elements, 98 and 944.

Objects in JSX

Objects can be defined as an unordered collection of data in the form of “key: value” pairs. JSX can’t render objects. React has no way to tell what to render when provided with an object, thus the Invariant Violation error pops up when attempting so. The error “Objects are not valid as a React child” is standard, and the solution is to manipulate the object so that it becomes a valid element.

And this is excellent because it now remains in the developer's hand to decide how to present the data in the object in its application.

Example:

```
const App = () =>
const myVariable = {
  productName: "Watermelon",
  price: 12
};
return (
  <div>
    {myVariable.productName} : ${myVariable.price}
  </div>
);
}
```

And the above code would render a div with the content: Watermelon: \$12

Tables in JSX

The tables are created using the `<table>` tag in which the `<tr>` tag is used to create table rows, and the `<td>` tag is used to create data cells. The elements under `<td>` are regular and left aligned by default. Here, the `border` is an attribute of `<table>` tag, and it is used to put a border across all the cells. If you do not need a border, then you can use `border = "0"`.

Table Heading

Table heading can be defined using `<th>` tag. This tag will be put to replace `<td>` tag, which is used to represent actual data cells. Normally you will put your top row as a table heading as shown below; otherwise, you can use `<th>` element in any row. Headings, which are defined in `<th>` tag are centred and bold by default.

Colspan and Rowspan Attributes

You can use the `colspan` attribute if you want to merge two or more columns into a single column. Similarly, you can use `rowspan` if you want to merge two or more rows. Example:

```
<table border = "1">
  <tr>
    <th>Column 1</th>
    <th>Column 2</th>
    <th>Column 3</th>
  </tr>
  <tr>
    <td rowspan = "2">Row 1 Cell 1</td>
    <td>Row 1 Cell 2</td>
    <td>Row 1 Cell 3</td>
  </tr>
  <tr>
    <td>Row 2 Cell 2</td>
    <td>Row 2 Cell 3</td>
  </tr>
  <tr>
    <td colspan = "3">Row 3 Cell 1</td>
  </tr>
</table>
```

| Column 1 | Column 2 | Column 3 |
|--------------|--------------|--------------|
| Row 1 Cell 1 | Row 1 Cell 2 | Row 1 Cell 3 |
| | Row 2 Cell 2 | Row 2 Cell 3 |
| Row 3 Cell 1 | | |

Table Caption

The caption tag will serve as a title or explanation for the table and will show up at the top. It can be just before the table's 1st `<tr>` element.

Example:


```
<table border="1">
  <caption>This is the caption</caption>
  <tr>
    <td>row 1, column 1</td>
    <td>row 1, column 2</td>
  </tr>
  <tr>
    <td>row 2, column 1</td>
    <td>row 2, column 2</td>
  </tr>
</table>
```

| This is the caption | |
|---------------------|-----------------|
| row 1, column 1 | row 1, column 2 |
| row 2, column 1 | row 2, column 2 |

<thead>, <tbody>, <tfoot>

The **<thead>** tag is used to group header content in a table. The **<tbody>** tag is used to group the body content in a table. The **<tfoot>** tag is used to group footer content in a table. These are the semantic tags that not only provide meaning to the elements but also have some other useful functions as well. Browsers can use these elements to enable scrolling of the table body independently of the header and footer. Also, when printing a large table that spans multiple pages, these elements can enable the table header and footer to be printed at the top and bottom of each page.

Note: You can use one table inside another table. Not only tables, but you can also use almost all the tags inside table data tag **<td>**. This is called a nested table.

Conditional Rendering

Conditional rendering in React works the same way conditions work in JavaScript. We can use JavaScript operators like if - else or the conditional operator (ternary operator) or AND operator or OR operator in JSX.

Conditional rendering with if else statement

We can use the if-else statements to render a JSX expression on the basis of some conditions. Note that if-else statements can't return additional JSX elements apart from the elements which are inside the if-else statements. This is because the return keyword inside the App component will return the elements which are after the return statement. So, if any condition inside the if-else statement is true, then just the elements inside that condition gets rendered and it doesn't even check the rest of the conditions or elements which are put outside that truthy condition. Example:

```
const App = () => {  
  const email = "demo@codingninjas.com";  
  const password = "demo";  
  if (email == "demo@codingninjas.com") {  
    if (password == "demo") {  
      return <h1>User is an employee.</h1>;  
    } else {  
      return <h1>Incorrect password</h1>;  
    }  
  } else {  
    return <h1>User is a student.</h1>;  
  }  
};
```

Conditional rendering with the ternary operator

The conditional (ternary) operator is the only JavaScript operator that takes three operands: a condition followed by a question mark (?), then an expression to execute if the condition is truthy followed by a colon (:), and

finally, the expression to execute if the condition is falsy. This operator is frequently used as an alternative to an if...else statement.

Example:

```
const App = () => {
  const email = "demo@codingninjas.com";
  const password = "demo";
  return (
    <>
      {
        (email == "demo@codingninjas.com")
          ? (password == "demo")
            ? <h1>User is an employee.</h1>
            : <h1>Incorrect password.</h1>
          : <h1>User is a student.</h1>
      }
    </>
  )
};
```

Conditional rendering with AND operator

Another way to conditionally render a React component is by using the && operator. It returns the first falsy and last truthy value.

Example:

```
const App = () => {
  const email = "demo@codingninjas.com";
  const password = "demo";
  return (
    <>
      {
        (email == "demo@codingninjas.com" && password == "demo")
        && <h1>User is an employee.</h1>
      }
      {
        (email == "demo@codingninjas.com" && password != "demo")
        && <h1>Incorrect password.</h1>
      }
    </>
  )
};
```

```
    }  
    {email !== "demo@codingninjas.com"  
    && <h1>User is a student.</h1>}  
  </>  
);  
};
```

Conditional rendering with OR operator

We can also render a React component by using the `||` operator. It returns last falsy and first truthy value.

Example:

```
const App = () => {  
  const email = "demo@codingninjas.com";  
  return (  
    <>  
      {( email == "demo@codingninjas.com" ||  
        email == "demo2@codingninjas.com") && (  
        <h1>User is an employee.</h1>  
      )}  
    </>  
  );  
};
```

Summarising it

Let's summarise what we have learned in this module:

- Learned about the JSX.
- Learned about React Fragments and its shortcut.
- Learned about JSX Expressions.
- Learned about how arrays and lists are rendered in React.
- Learned about how to use objects in JSX.
- Learned about how to use tables in JSX.
- Learned about different types of conditional rendering in JSX.

Additional References (if you want to explore more):

- More information JSX Expressions: [Link](#)
- To apply React Developer tools: [Link](#)
- To read more about the array iteration methods: [Link](#)
- More information on Tables: [Link](#)
- Advanced information on Tables: [Link](#)
- You can read about nullish coalescing operator: [Link](#)
- Logical operators: [Link](#)

Mini-Project(SCORE- KEEPER)

Events in JSX

Handling events with React elements is very similar to handling events on DOM elements. There are some syntax differences:

- React events are named using camelCase, rather than lowercase.
- With JSX you pass a function as the event handler, rather than a string.

For example, the HTML:

```
<button onclick="activateLasers()">
  Activate Lasers
</button>
```

is slightly different in React:

```
<button onClick={activateLasers}>
  Activate Lasers
</button>
```

Another difference is that you cannot return false to prevent default behavior in React. You must call `preventDefault` explicitly. For example, with plain HTML, to prevent the default form behavior of submitting, you can write:

```
<form onsubmit="console.log('You clicked submit.');" return false">
  <button type="submit">Submit</button>
</form>
```

In React, this could instead be:

```
function Form() {
  function handleSubmit(e) {
    e.preventDefault();
    console.log('You clicked submit.');
```

```
}  
return (  
  <form onSubmit={handleSubmit}>  
    <button type="submit">Submit</button>  
  </form>  
)  
);  
}
```

Here, `e` is a synthetic event. React defines these synthetic events, so you don't need to worry about cross-browser compatibility. React events do not work the same as native events. See the [SyntheticEvent](#) reference guide to learn more.

When using React, you generally don't need to call `addEventListener` to add listeners to a DOM element after it is created. Instead, just provide a listener when the element is initially rendered.

Virtual DOM under the hood

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 by a library such as ReactDOM. This process is called reconciliation.

In reality, the virtual DOM is just an organized collection of React elements — plain objects, and it mimics the browser DOM in a way that is easier to maintain and update.

Let's take a step back and check out an example of a React element.

```
const title = <h1>Hello, world!</h1>
```

JSX does the heavy lifting to convert the familiar HTML syntax into a React element. Without JSX, this is just:

```
const title = React.createElement('h1', null, 'Hello, world!')
```

And below is the created React element under the hood.

```
{
  type: "h1",
  props: {
    children: "Hello, world!"
  }
}
```

More on Event-Handling

When you define a component using an ES6, a common pattern is for an event handler like 'handleClick' to be a method on the components. For example, the Toggle component returns a button that lets the user toggle between "ON" and "OFF" states:

```
var state = {isToggleOn: true};
handleClick() {
  state.isToggleOn = !isToggleOn;
  // rerender the App component
}
function Toggle{
  return (
    <button onClick={handleClick}>
      {state.isToggleOn ? 'ON' : 'OFF'}
    </button>
  );
}
```

The alternate way to pass arguments to event handlers is using inline functions-

```
<button onClick={() => state.isToggleOn = !isToggleOn; }>
  {state.isToggleOn ? 'ON' : 'OFF'}
</button>
```


You have to be careful about the meaning of this in JSX callbacks. In JavaScript, class methods are not bound by default. If you forget to bind `this.handleClick` and pass it to `onClick`, this will be `undefined` when the function is called.

But this is not the case in functional components and that's why we are using public fields syntax to correctly bind callbacks.

Forms in JSX

HTML form elements work a bit differently from other DOM elements in React because form elements naturally keep some internal state. For example, this form in plain HTML accepts a single name:

```
<form>
  <label>
    Name:
    <input type="text" name="name" />
  </label>
  <input type="submit" value="Submit" />
</form>
```

This form has the default HTML form behavior of browsing to a new page when the user submits the form. If you want this behavior in React, it just works. But in most cases, it's convenient to have a JavaScript function that handles the submission of the form and has access to the data that the user entered. The standard way to achieve this is with a technique called “**controlled components**”.

```
//variable to store form's values.
var state = {value: ''};

//methods used on handling Events in JSX
handleChange(event) {
  state = {value: event.target.value};
}
handleSubmit(event) {
  alert('A name was submitted: ' + state.value);
}
```

```
    event.preventDefault();
  }

  // created form in JSX
  const Form = () =>{
    return (
      <form onSubmit={handleSubmit}>
        <label>
          Name:
          <input type="text" value={state.value}
            onChange={handleChange} />
        </label>
        <input type="submit" value="Submit" />
      </form>
    );
  }
```

In a controlled component, form data is handled by a React component. The alternative is uncontrolled components, where form data is handled by the DOM itself. To write an uncontrolled component, instead of writing an event handler for every state update, you can **use a ref** to get form values from the DOM.

Iterate over Arrays in JSX

Given the code below, we use the `map()` function to take an array of numbers and double their values. We assign the new array returned by `map()` to the variable `doubled` and log it: For example:

```
const numbers = [1, 2, 3, 4, 5];

const doubled = numbers.map((number) => number * 2);

console.log(doubled);
```

This code logs `[2, 4, 6, 8, 10]` to the console.

You can build collections of elements and **include them in JSX** using curly braces `{}`.

Note- The `map()` method is used to transform the elements of an array, whereas the `forEach()` method is used to loop through the elements of an array. The `map()` method can be used with other array methods, such as the `filter()` method, whereas the `forEach()` method cannot be used with other array methods.

Few Important Concepts

Creating Refs

Refs are created using `React.createRef()` and attached to React elements via the `ref` attribute. Refs are commonly assigned to an instance property when a component is constructed so they can be referenced throughout the component.

Accessing Refs

When a ref is passed to an element in render, a reference to the `inputRef` becomes accessible at the current attribute of the ref.

```
let inputRef = React.createRef();
```

The value of the `ref` differs depending on the type of the node:

- When the `ref` attribute is used on an HTML element, the ref is created in the constructor with `React.createRef()` receives the underlying DOM element as its current property.
- When the `ref` attribute is used on a custom class component, the `ref` object receives the mounted instance of the component as its **current**.

Adding a Ref to a DOM Element

This code uses a ref to store a reference to a DOM node:

```
const Form = () =>{  
  <form onSubmit={handleSubmit}>  
    <input ref = {inputRef} placeholder="Name"/>  
    <button> Submit </button>  
  </form>  
}
```

- Pass it as `<input ref={inputRef}>`. This tells React to put this `<input>`'s DOM node into `inputRef.current`.
- In the handleClick function, read the input DOM node from `inputRef.current` and call `focus()` on it with `inputRef.current.focus()`.
- Pass the handleClick event handler to `<button>` with `onClick`.

While DOM manipulation is the most common use case for refs, the `createRef` can be used for storing other things outside React. Similarly to the state, refs remain between renders. Refs are like state variables that don't trigger re-renders when you set them.

SyntheticEvent

Your event handlers will be passed instances of `SyntheticEvent`, a cross-browser wrapper around the browser's native event. It has the same interface as the browser's native event, including `stopPropagation()` and `preventDefault()`, except the events work identically across all browsers.

React normalizes events so that they have consistent properties across different browsers.

To explore more, you can click here - [Click](#)

Summarising it

Let's summarise what we have learned in this Lecture:

- Learned about the Events in JSX.
- Learned about Form in JSX.
- Learned about Virtual DOM under the hood.
- Learned about how to store elements in an array and populate it.

Some References:

- More information JSX Events: [Link](#)
- More information on Ref and the DOM: [Link](#)
- To read more about Form: [Link](#)

Git Bash for Windows

What is Git Bash?

Git Bash is an application that provides Git command line experience on the Windows operating system. It is a command-line shell for enabling git with the command line in the system. Git Bash allows you to run Linux commands (eg. ls, pwd, mv) directly on your Windows machine.

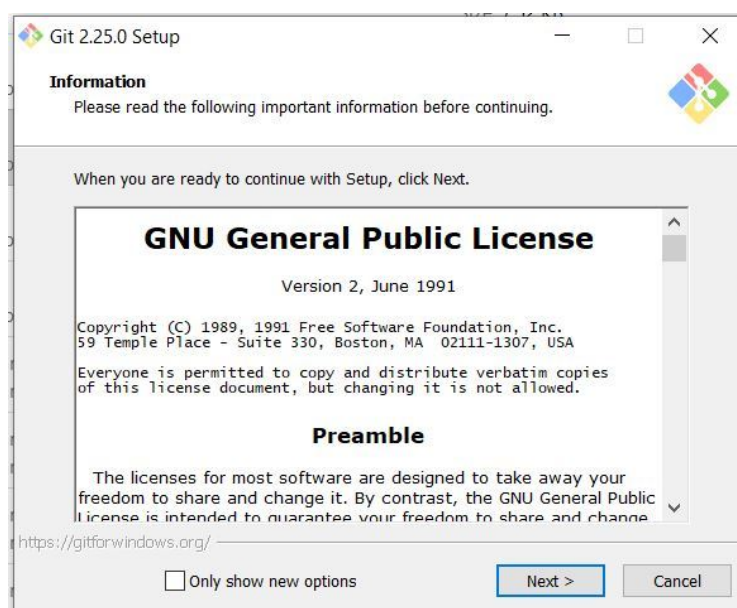
A shell is a terminal application used to interface with an operating system through written commands.

Git Bash is a package that installs Bash, some common bash utilities, and Git on a Windows operating system. It is a popular default shell on Linux and macOS.

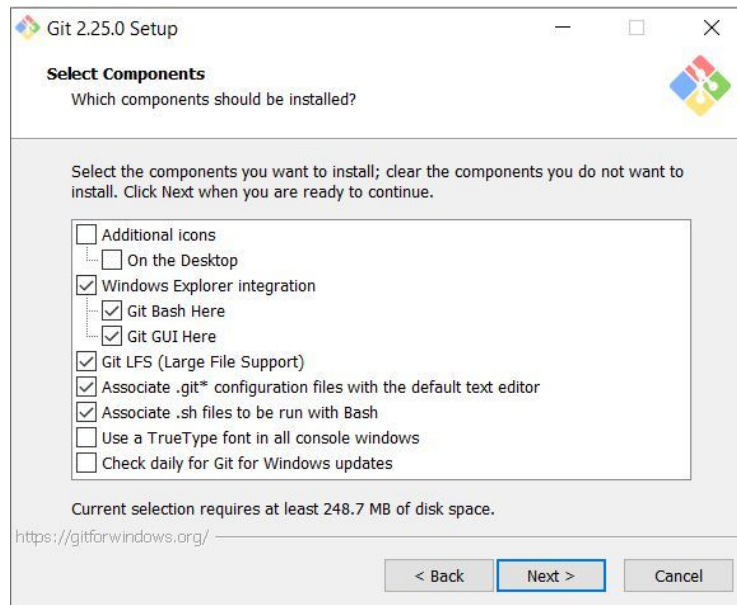
How to install Git Bash?

Follow the steps given below to install Git Bash on Windows:

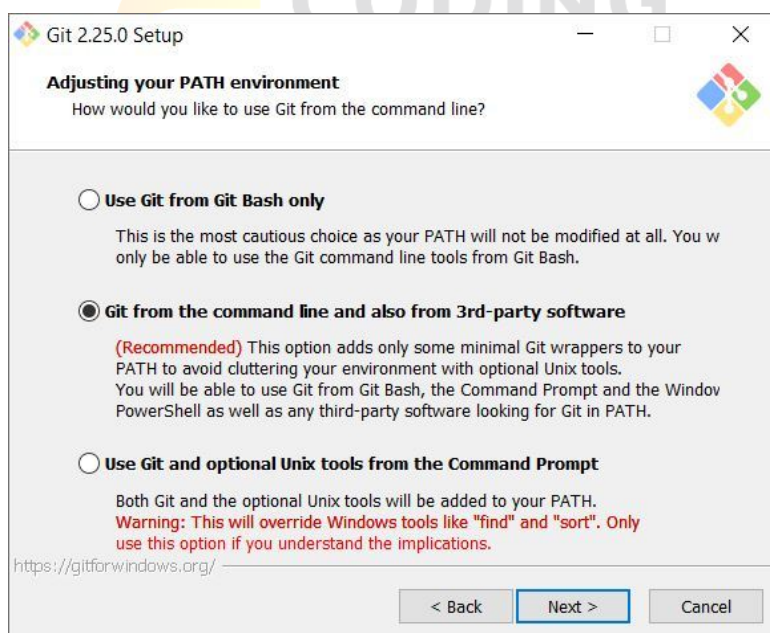
Step 1: The .exe file installer for Git Bash can be downloaded from here: [link](#). Once downloaded execute that installer.



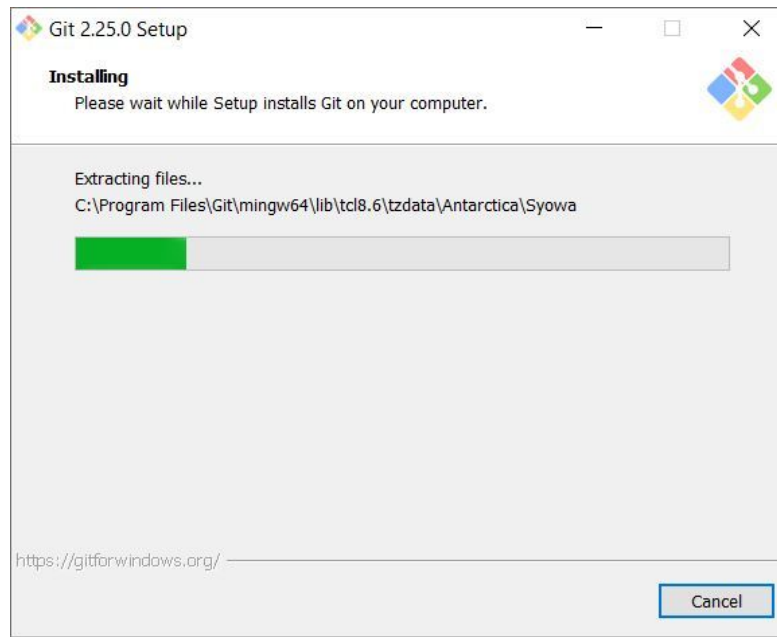
Step 2: Select the components that you need to install and click on the Next button.



Step 3: Select how to use the Git from command-line and click on Next to begin the installation process.

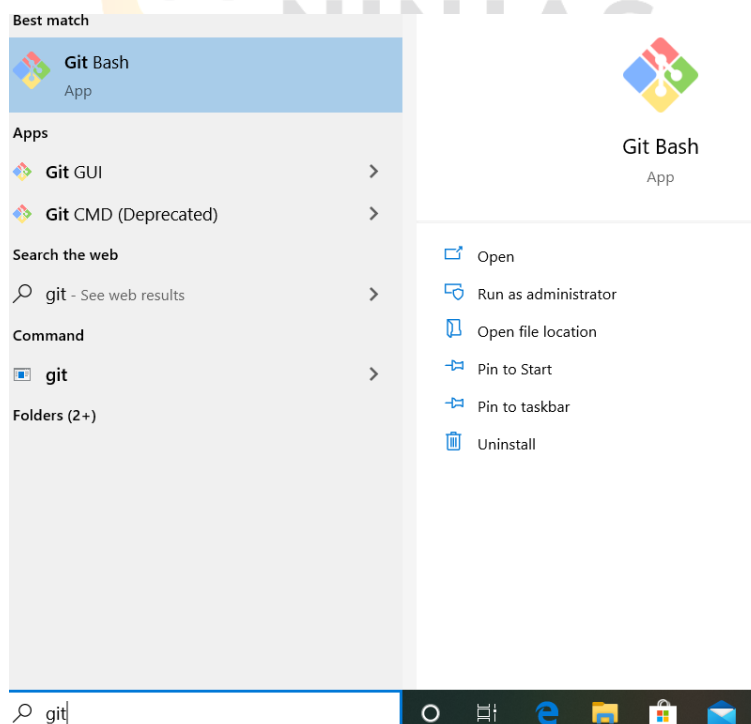


Step 4: Let the installation process finish to begin using Git Bash.



How to use Git Bash?

To open Git Bash shell navigate to the folder where you have installed the git otherwise simply search in your OS for git bash.



Now you can execute any Linux commands from this shell. Check the reference links to learn more about commonly used Linux commands.

Some References:

- ❖ Download Git Bash: [link](#)
- ❖ Git Bash for Windows: [link](#)
- ❖ Git Bash commands: [link](#)



Create-React-App

CDN (Content Delivery Network)

A CDN is a network of servers that distributes content from an "origin" server to end users around the globe by caching content nearby each user's point of internet access. They initially request content from the origin server, which is copied and stored elsewhere as needed. Both React and ReactDOM are available over a CDN.

Drawbacks of using CDNs

- An essential prerequisite is an internet connection.
- If there is a server issue, packages might not load.
- Since one script file may be dependent on another, the order of CDN links is important. Browsers are not capable of linking files together. It's possible that script files won't link correctly.
- You may forget to list all of the CDN connections required for an application.

How to overcome the drawbacks?

We can overcome drawbacks by installing packages locally and using webpack which helps in file management and linking.

Webpack

Webpack is a module bundler. Its main purpose is to bundle JavaScript files for usage in a browser. For eg: if there are 2-3 files like - index.js, app.js, form.js, button.js. A module bundler will create a bundle of these files with their dependencies. Webpack goes through your package and creates a **dependency graph** which consists of various modules which your web app would require to function as expected. Then, depending on this graph, it creates a new package which consists of the minimum number of files

required, often just a single bundle.js file which can be plugged into the html file easily and used for the application. CRA comes pre configured with a webpack.

Webpack Configuration

Webpack can be configured by adding a **webpack.config.js** file in the root of our application structure. It requires the following:

- **Entry property**, which is used to specify which file webpack should start with to get the internal dependency graph created. A path is passed while creating the entry point (for eg: src/index.js).
- **Loaders** check all the imports if any file needs any transformations or not. So, it internally transforms the files which need transformations.
- **Output property** specifying where the bundled file should be generated. Generally kept in the “dist” or “build” folder.

Tools Installation

Nodejs

Node.js is a **run-time environment** that comes with everything you need to run a JavaScript programme. It is used for running scripts on the server to render content before it is delivered to a web browser. **Node Package Manager, or NPM**, is a tool and repository for developing and sharing JavaScript code. You can Download Node.js Installer from [here](#). NPM is installed automatically when node js is installed.

Google Chrome Browser

Chrome, an Internet browser released by Google, Inc. It is an open source program for accessing the World Wide Web and running Web-based applications. **Chrome DevTools** is a amazing set of web developer tools built directly into the Google Chrome browser. You can go [here](#) to check the instructions for downloading Google chrome.

VS Code

Visual Studio Code is a lightweight but powerful source code editor which runs on your desktop and is available for Windows, macOS and Linux.. Visit the official website of the [Visual Studio Code](https://code.visualstudio.com/) using any web browser like Google Chrome, Microsoft Edge, etc to install the VS Code according to your operating system.

Create React App

Create-React-App is a tool given by Facebook that provides us with **boilerplate code** and helps us to create our own react app. It comes pre configured with webpack and babel.

Instructions to create react app:

1. Go to the desktop using `cd Desktop` or your project root directory where you want to create the project.
2. Use command `npx create-react-app <app_name>`, to create the react app.
3. Use the command `cd <app_name>` and go to the app.
4. Use `ls` to display the list of files in the current directory.
5. And now open the file in VS Code.
6. Use `npm start` to start your first react project.

Create-React-App can also be installed globally so that you can create a react project anywhere inside your system. `npm install -g create-react-app` is used to install CRA globally. But it is not recommended as versions may change, and you may have two projects going that use two different versions. It's not even needed to install create-react-app as you can do `npx create-react-app <app_name>` and always use the latest version without polluting your system.

Folder and File Structure

The React application automatically creates required folders, as shown below.

- **node_modules:** This folder will contain all the third party libraries and other react js dependencies.
- **index.html:** It is the html file which gets loaded on the browser. It contains html tags.
- **manifest.json:** It contains information about your app like name, description, icon, etc.
- **src folder:** src is one of the main folders in react project. You can delete or modify any file of this folder except index.js as it is the entry point for webpack.
- **index.js:** index.js is the file that will be called once we will run the project.
- **app.js:** App.js is a component that will get loaded under index.js file.
- **.gitignore:** This file is used by source control tool to identify which files and folders should be included or ignored during code commit
- **package.json:** This file contains dependencies and scripts required for the project.
- **package.lock.json:** It is created for locking the dependency with the installed version. It will install the exact latest version of that package in your application and save it in the package.

Imports/Exports

If you declare some value/function in some file, and you try to access that in another file, you won't be able to do so. As, each individual has its own local scope. To make all these available in another file, we can use export and import.

The **export** and **import** are the keywords to utilize the code of one file to other files.

Export

Export keyword is used to provide code to other files. There are two types of exports:

- **Named Exports:** This syntax allows you to individually import values that have been exported by their name. It can be done in two ways:

- Export Individually

```
export var a=10;

export let obj = {name:"Alexa"};

export function greet() {
  console.log("Hello");
}
```

- Export all at once at the bottom

```
var a=10;

let obj = {name:"Alexa"};

function greet() {
  console.log("Hello");
}

export {a, obj, greet};
```

- **Default Exports:** You can export multiple named exports and imported them individually or as one object with each export as a method on the object. But, files can also contain a default export, using the **default** keyword. A default export will not be imported with curly brackets, but will be directly imported into a named identifier. It can be done in two ways:

- Export Individually

```
export default function greet() {  
  console.log("Hello");  
}
```

- Export at the bottom

```
export function greet() {  
  console.log("Hello");  
}  
  
export default greet;
```

Import

Import keyword is used to read code exported from another files. The **as** keyword is used to create an alias to import under different names. Import can be done in three ways:

- Importing named exports:

```
import {x, ...} from "file"
```

- Importing the default export:

```
import x from "file"
```

- Import all:

```
import * as obj from "file"
```

Additional Notes:

File Structure and Conventions

Some of the common approaches that are followed while creating the folder structure of the application are as follows:

- **Grouping by features or routes:** structuring project by locating CSS, JS, and tests together inside folders grouped by feature or route.

- **Grouping by file type:** structuring project by grouping similar files together. For eg: separate folder for CSS files, Test files etc. This is the most common approach which is followed.

The top level directory structure of src folder can be as follows:

- **assets** - for keeping global static assets such as images, svgs, company logo, etc.
- **components** - for keeping global shared/reusable components, such as layout (wrappers, navigation), form components, buttons
- **services** - this folder can have JavaScript modules
- **store** - Global Redux store
- **utils** - Utilities, helpers, constants, etc. can be kept in this folder
- **views** - Can also be called "pages", the majority of the app would be contained here

Note: Try to minimise nesting of folders as much as possible as it becomes harder to write relative imports between them.

Summarising it

Let's summarise what we have learnt in this module:

- Learned about drawbacks of CDNs and how to overcome them.
- Learned about Webpack and configuration.
- Learned about tools required to install and use create-react-app.
- Learned how to install create-react-app.
- Learned about files and folder structure of a react app.

Some Additional Resources : To explore more

- What Is a CDN and How Does It Work?: [link](#)
- CDN links for React and React DOM: [link](#)
- Babel and Webpack in 2 minutes: [link](#)

- Webpack link: [link](#)
- Babel link: [link](#)
- Visual studio download link: [link](#)
- Nodejs download link: [link](#)
- Documentation for create-react-app: [link](#)
- Folder and File structure: [link](#)

React Components-I

Components in React

A component is a small, reusable chunk of code. It lets you split the UI into independent, reusable pieces, and think about each piece in isolation.

We can create components with JavaScript classes or functions. To use React's properties and methods in our class components we must subclass the **Component** class from React. This way we can use the code from the React library without having to write it over and over again.

A function is a valid React component if it accepts a single props object argument with data and returns a React element. We call these functions as functional components because they are simple JavaScript functions.

Functional Component Snippet

```
const Navbar = () => {  
  return (  
    <div>  
      <span>Navbar component</span>  
    </div>  
  );  
};  
export default Navbar;
```

Class component Snippet

```
import { Component } from "react";
```

```
class Navbar extends Component {  
  render() {  
    return (  
      <div>  
        <span>Navbar component</span>  
      </div>  
    );  
  }  
}  
  
export default Navbar;
```

Function v/s Class Components

| Functional Components | Class Components |
|--|--|
| Functional components cannot extend from any class | Class components must extend from the React.Component class |
| Create and Maintain state information with hooks | Create and Maintain state information with lifecycle methods |
| Do not support a constructor | Require a constructor to store state |
| Do not require the render function | Require a render function that returns an HTML element |

Why use Functional Components?

- Make code more reusable and readable
- Are easy to test and debug
- Yield better performance
- Low coupling and cross dependency in code
- Easy to separate code into container and presentational components

Why use Class Components?

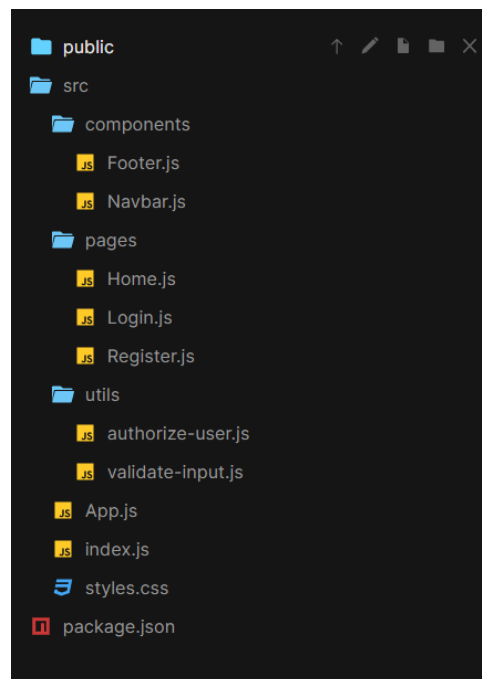
- If you prefer working with classes
- Still used in some legacy projects

File Structure Conventions for Components

React doesn't have opinions on how you put files into folders. That said there are a few common approaches popular in the ecosystem you may want to consider.

The typical folder structure in a React project can follow the following conventions to make the code more organised.

Example Snapshot -



Components

This folder consists of a collection of reusable UI components like buttons, modals, inputs, loader, etc. They can be used across various files in the project.

Pages

The files in the pages folder indicate the route of the react application. Each file in this folder corresponds to a standalone page of the project.

Utils

Utils folder consists of some repeatedly used functions that are commonly used in the project.

State in React

State is a built-in object in React that is used to contain dynamic information about a component. Unlike props that are passed from the outside, a component maintains its own state.

A component's state is mutable and it can change over time. Whenever it changes, the component re-renders.

Adding an initial state

To add an initial state to a component instance we give that component a state property. This property should be declared inside of the class constructor and should be set to an object with key and value pairs. We must also call super with props inside of the constructor to access common properties of the built-in Component class.

Super

The super keyword calls the constructor of the parent class. In our case the call to super passes the props argument to the constructor of React.Component class and saves the return value for the derived class component.

Updating state with setState

The components state can be updated with **this.setState** built-in method. It takes an object and merges it with the component's current state. If there are properties in the current state that are not a part of that object, those properties remain unchanged.

Anytime that we call this.setState it automatically calls the render method as soon as the state changes which rerenders the component with the updated state value.

Accessing previous state values

The **setState** method can take a callback function as an argument which receives the previous state as a default parameter. This is useful in cases where we need access to previous state values to update the current state.

State is Asynchronous

The **setState** method works in an asynchronous way. That means after calling **setState** the **this.state** variable is not immediately changed.

So If we want to perform an action after the state value is updated we can pass a callback function as a second parameter to the **setState** method.

Example snippet -

```
import { Component } from "react";
export default class Navbar extends Component {
  constructor(props) {
    super(props);
    // initialising state
    this.state = { count: 0 };
  }
  // access previous state with a callback
  updateState = () => {
    this.setState((prev) => ({ count: prev.count + 1 }));
  };
  render() {
    return (
      <div>
        <h1>Count is {this.state.count}</h1>
        <button onClick={this.updateState}>Click Me</button>
      </div>
    );
  }
}
```

Some References:

- ❖ Function and Class Components: [link](#)
- ❖ File Structure conventions: [link](#)
- ❖ State structure patterns: [link](#)

React Components-II

Props in React

A component can pass information to other components. Information that gets passed from one component to another is known as props short for properties. A component's props is an object which holds information about that component.

Props are passed down from parent to child components as a key and value pair. If we want to pass information that is not string we have to wrap it with curly braces. This information will be stored inside of the props object of the child component.

The most common use of props is to pass data and event handlers down to the child components.

Rules of Props

There is only one strict rule in regard to props in React. Props are read-only. A component should never try to mutate or change the value of its props.

Default Props

Default props can be used to define any props that you want to be set for a component. They can be used to ensure that props will have a value if it was not specified by the parent component.

We can set default values for the props by assigning to the special **defaultProps** property on the component class.

Additional Information: Type Checking in Props

Note - Props type checking can be optionally used to ensure that the passed value is of the correct data. This can help prevent errors in rendering and force correct usage of components.

Props type-checking can be used to validate props that are passed down from the parent for missing or incorrect data type values.

Type checking of props can also help document the code to make it easier to understand and debug the component class.

We can add type checking to our props by specifying it on the **propTypes** static property on the components class after it has been defined. The value of this property is an object that has multiple key and value pairs. Each key corresponds to a prop of that our component expects and the value should be the expected data type for that prop.

We need to import the PropTypes object from 'prop-types' and use it to specify the expected data type for each prop.

Example snippet -

```
import { Component } from "react";
import PropTypes from "prop-types";

export default class Navbar extends Component {
  render() {
    const { username, avatar } = this.props;
    return (
      <div>
        <span>{username}</span>
        <img alt={username} src={avatar} />
      </div>
    );
  }
}

// setting default props
Navbar.defaultProps = {
  username: "Pranav",
  avatar: "/image.png"
};

// type checking props
Navbar.propTypes = {
  username: PropTypes.string,
```

```
avatar: PropTypes.string  
};
```

State v/s Props

Props and state are both plain JavaScript objects. While both hold information that influences the output of render, one important difference between the two is that Props get passed to the component whereas state is managed within the component.

| State | Props |
|--|---|
| State can be changed (mutable) | Props are read-only and cannot be changed (immutable) |
| State changes can be asynchronous | Props cannot be changed |
| State is managed within the component | Props gets passed to the component |
| State can used to display changes with the component | Props are used to pass information between components |

Some References:

- ❖ Props vs State: [link](#)
- ❖ Presentational and Container components: [link](#)

Styling in React

Styling in React

Styling is one of the most important aspects of the React application. There are various ways to follow when planning to style React components. Some of the most popular and modern styling strategies are:

- CSS Stylesheets
- Inline Styling
- Styled Components
- CSS Modules

CSS Style sheets

This is the conventional way of styling websites. In this method, we separate the CSS part into an external file with a .css extension which is simply imported into the React component. After that, we can give className and id to point which styles should point to which element.

Note: **class** attribute is used in HTML, whereas **className** is used in React. This is because class is a reserved keyword in JavaScript and since React uses JSX, which is a syntax extension to JavaScript, we must use className instead of the class attribute.

Example: Styling the Navbar

Basic Structure of the Navbar Component



Navbar.js

```
import './Navbar.css';
```

```
const Navbar = () => {
  return (
    <div className="navbar">
      <span>Title of Navbar</span>
      <span>
        Cart Icon<sup>count</sup>
      </span>
    </div>
  );
};

export default Navbar;
```

Navbar.css

```
.navbar {
  display: flex;
  justify-content: space-between;
}
```

Output:



Advantages:

- Styles of numerous documents can be organized from one single file.
- Good performance as it is easy for the browser to optimize and cache the files locally for repeated visits.
- You can very easily rip out the entire stylesheet and create a new one to refresh the look and feel of your app.

Disadvantages:

- If not properly structured, It can become long and difficult to navigate through as the application becomes complex.

- CSS Stylesheets have global scopes and can cause conflicts in styles if the same selector names are used in the codebase

Inline Styling

Inline CSS is the widely preferred but less recommended way to style your website. In React, you will write your style using the style attribute followed by = and then CSS properties enclosed by double curly braces {{ }} instead of quotes “. ”. React uses JSX, In JSX for evaluation of any variable, state object , expression etc has to be enclosed in {}. The style attribute in React only accepts a JavaScript object with camelCased properties and values enclosed with quotes rather than a CSS string. This is the reason there are two pairs of curly braces.

Note: Inline styles have got more priority, and they will overwrite any other styles given to them in any manner.

Example: Styling the Navbar

Navbar.js - Method 1 (inline styling)

```
const Navbar = () => {
  return (
    <div style={{display:"flex", justifyContent:"space-between"}}>
      <span>Title of Navbar</span>
      <span>
        Cart Icon<sup>count</sup>
      </span>
    </div>
  );
};

export default Navbar;
```

Navbar.js - Method 2 (internal style object)

```
const Navbar = () => {
  return (
    <div style={styles}>
```

```


    <span>Title of Navbar</span>
    <span>
      Cart Icon<sup>count</sup>
    </span>
  </div>
);
};

let styles = {
  display: "flex",
  justifyContent: "space-between"
};

export default Navbar;

```

Output:



Advantages:

- Inline CSS is best suited for learners and when you are testing a particular style.

Disadvantages:

- It cannot be reused, i.e., you must write the same CSS code repeatedly for the same styles.
- It does not provide browser cache advantages.
- Some useful CSS properties like pseudo-codes, pseudo-classes, media queries, etc., cannot be used in inline styles.

Styled Components

styled-components is a library for React that allows you to use component-level styles in your application that are written with a mixture of

JavaScript and CSS using a technique called CSS-in-JS. This is done using the tagged template literal syntax. Follow the following steps to implement styling using styles-components:

1. First, we need to install the styled-components library by running `npm install styled-components`.
2. We then need to import the styled component library into our component by writing `import styled from 'styled-components'`.
3. Now we can create a variable by selecting a particular HTML element where we store our style keys.
4. Then we use the name of the variable we created as a wrapper around our JSX elements.

Example: Styling the Navbar

Navbar.js

```
import styled from "styled-components";

const Nav = styled.div`
  display: flex;
  justify-content: space-between;
`;

const Navbar = () => {
  return (
    <Nav>
      <span>Title of Navbar</span>
      <span>
        Cart Icon<sup>count</sup>
      </span>
    </Nav>
  );
};

export default Navbar;
```

Output:



Example: Dynamic Styling with props

One of the advantages of styled-components is that the components themselves are functional, as in you can use props within the CSS. You can also use conditional statements to change styles based on a state or prop.

Navbar.js

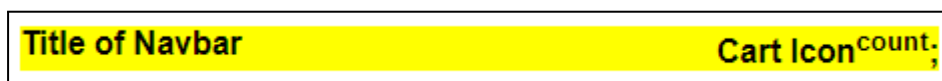
```
import styled from "styled-components";

const Nav = styled.div`
  display: flex;
  justify-content: space-between;
  background-color: ${(props) => props.color};
  font-weight: ${(props) => (props.bold ? "normal" : "bolder")};
`;

const Navbar = () => {
  return (
    <Nav color="yellow" bold="">
      <span>Title of Navbar</span>
      <span>
        Cart Icon<sup>count</sup>
      </span>
    </Nav>
  );
};

export default Navbar;
```

Output



Advantages:

- Styled components eliminate specificity problems as it encapsulates CSS inside a component.
- styled-components allow you to combine CSS and JS in the same file.
- You can make use of props to dynamically change the styles in any way.

Disadvantages:

- Writing CSS in JS means separating the two in the future will be difficult, which is terrible for maintainability.
- Differentiating between styled and React components can be difficult
- For applications that use styled components, the browser downloads the CSS and parses it using JavaScript before injecting them into the page. This causes performance issues because the user must download a lot of JavaScript in the initial load.

CSS Modules

A CSS Module is a CSS file with a `.module.css` extension in which all class names and animation names are scoped locally by default. One huge advantage of the CSS modules is that it is locally scoped to the component which prevents conflicting styles because of using the same selector names.

The CSS properties are hashed into unique class names during compilation. You can use CSS Modules by creating a file with extension `.module.css` file and import it into the specific React Component file.

Example: Styling the Navbar

Navbar.js

```
import styles from './Navbar.module.css';

const Navbar = () => {
```

```
return (
  <div className={styles.navbar}>
    <span>Title of Navbar</span>
    <span>
      Cart Icon<sup>count</sup>
    </span>
  </div>
);
};
export default Navbar;
```

Navbar.module.css

```
.navbar {
  display: flex;
  justify-content: space-between;
}
```

Output:



Note: When you check it in the browser. On inspecting, The class name is `__src_Navbar_module__navbar` which is further transformed into a Unique Identifier. This will remove any chances of name collision in the React App.

```
<div id="root">
  <div class="App">
    <div class="__src_Navbar_module__navbar">...</div>
  </div>
</div>
```

```
.__src_Navbar_module__navbar {
  display: flex;
  justify-content: space-between;
}
```

Advantages:

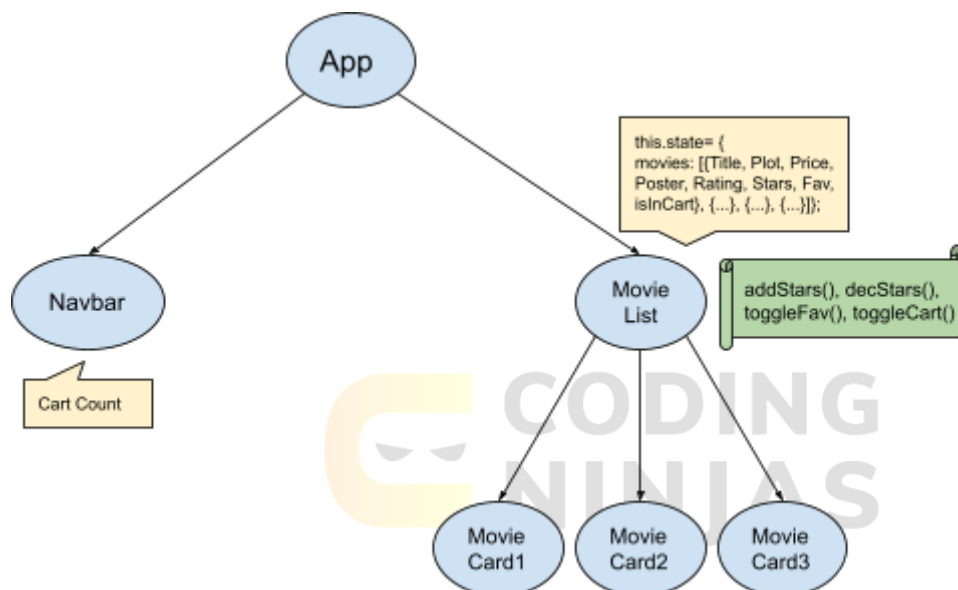
- Modular and reusable CSS
- No more styling conflicts, So, you can use the same CSS class in multiple CSS files

Disadvantages:

- Using the styles object whenever constructing a className is compulsory.
- Only allows usage of camelCase CSS class names.

Lifting up the State

Lifting state up is a common pattern that is essential for React developers to know. It helps you avoid more complex (and often unnecessary) patterns for managing your state. The app structure of the project is shown below:

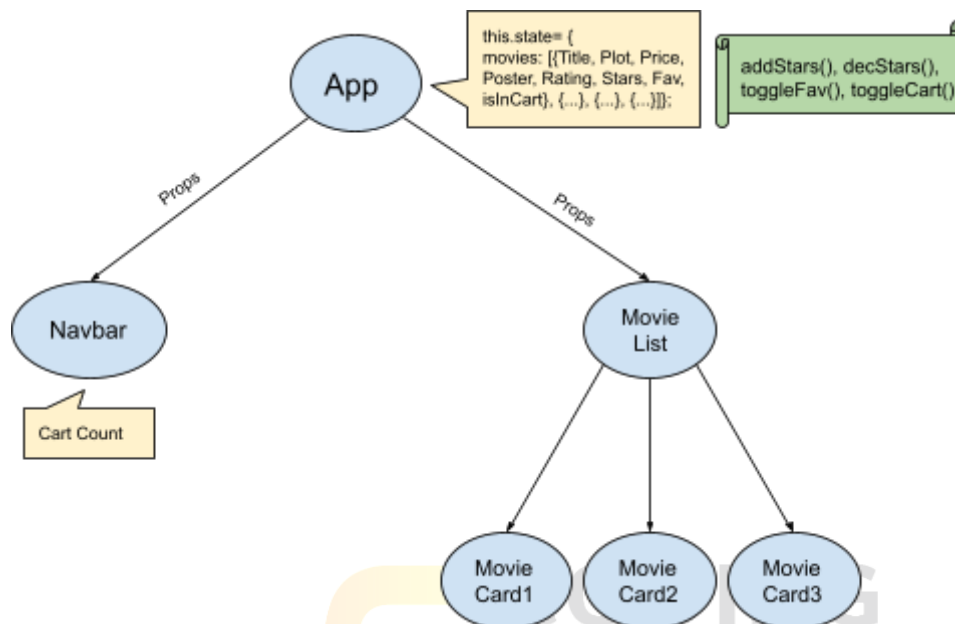


In `<Navbar>`, we have a **cart count** which we want to increase/ decrease as the add to cart/ remove from cart button is pressed. These buttons are present under `<MovieCards>` but the logic/ event handlers for add to cart and remove from cart are written inside `<MovieList>`. We want to access `cartCount` in `<Navbar>` component which is a sibling of this component, but we cannot pass the data among siblings as per the rule we can only pass data from parent to child.

```

this.state = {
  movies: movies,
  cartItems :[],
  cartCount:0
}
  
```

The solution is **Lifting up the state to App component** and passing the **cartCount** from **<App>** component to **<Navbar>**. So with states we can take all the handlers to the **<App>** component. And then we can pass everything as a **prop** to another component.



```

render(){
  const {movies, cartItems, cartCount} = this.state;
  return(
    <>
    <Navbar cartItems = {cartItems} cartCount={cartCount}/>
    <MovieListS movies={movies}
      decStars = {this.decStars}
      addStars = {this.addStars}
      toggleCart = {this.toggleCart}
      toggleFav = {this.toggleFav} />
    </>
  )
}

```

In the Navbar component we can now access CartCount and in MovieList, we can access all the handlers through props.

Presentational Components

A component that has to have a state, make calculations based on props or manage any other complex app logic is called a container component

A component whose only job is to contain some JSX and render it in UI is called a presentational component. A presentational component must be exported and will never render anything on its own. It will always be rendered by a container component.

All presentation components can be changed from class-based to stateless functional components as shown in the following example.

Navbar.js (class-based)

```
import { Component } from "react";
import styles from "../navbar.module.css";

class Navbar extends Component {
  render() {
    return (
      <div className={styles.navbar}>
        <span>Title of Navbar</span>
        <span>
          Cart Icon<sup>count</sup>
        </span>
      </div>
    );
  }
}

export default Navbar;
```

Navbar.js (stateless functional)

```
import styles from "../navbar.module.css";

function Navbar() {
  return (
    <div className={styles.navbar}>
      <span>Title of Navbar</span>
      <span>
```

```
      Cart Icon<sup>count</sup>
    </span>
  </div>
);
}
export default Navbar;
```

Note: Functional components can create and maintain their internal state using React hooks. They can also be a container component.

Summarising it

Let's summarise what we have learned in this Lecture:

- Learned about the styling strategies in React.
- Learned about styling React components using CSS Stylesheets.
- Learned about styling React components using Inline Styling.
- Learned about styling React components using styled components.
- Learned about styling React components using CSS Modules.
- Learned about lifting up the state.
- Learned about presentational components.

Some References:

- Styling and CSS in React: [link](#)
- Inline CSS vs Raw CSS: [link](#)
- Styled Components: [link](#)
- Dynamic Styling with styled-components: [link](#)
- CSS Modules: [link](#)
- Lifting state up in React: [link](#)
- Container vs Presentational components: [link](#)

Component Lifecycle Methods

Introduction

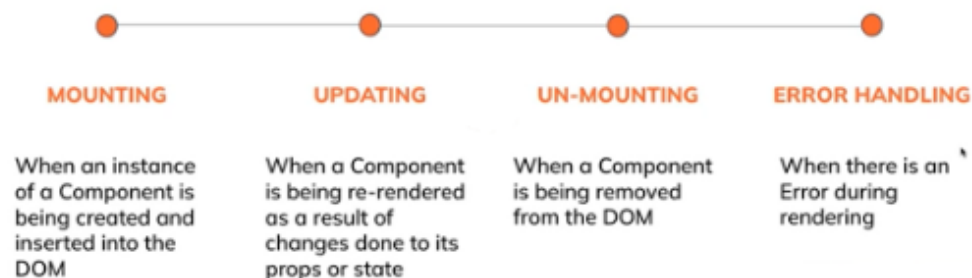
Lifecycle

Lifecycle is the series of stages through which a component passes from the beginning of its life until its death. The life of the React component starts when it is born (Created/Mounted) and ends when it is destroyed (Unmounted).

Different Phases of a Lifecycle

Different Phases of a component lifecycle are:

- **Mounting:** When a component is being created and inserted into the DOM.
- **Updating:** When a Component is being re-rendered due to any updates made to its state or props.
- **Unmounting:** When it is destroyed/ removed from the DOM.
- **Error Handling:** When there is an error during rendering.



During the lifecycle of a component, certain methods are called at each phase where we can execute some logic or perform a side-effect.

Side effects are actions that are not predictable because they are actions that are performed with the "outside world."

For example: Using Browser APIs like `localStorage`, using the native DOM methods instead of the `ReactDOM`, fetching the data from an API, and setting timeouts and intervals.

Mounting Phase

These methods are called in the following sequence when an instance of a component is being created:

- `constructor()`
- `static getDerivedStateFromProps()`
- `render()`
- `componentDidMount()`

constructor

- A special function that will get called whenever a new component is created.
- It can be used to initialize the state and bind the event handlers.
- This is the only place where the state can be modified directly. Everywhere else state should be updated using the `setState` function (used to update the state of a component).
- Avoid introducing any side effects/subscriptions in the constructor.

Example:

```
class Counter extends Component {
  constructor(props) {
    super(props);
    this.state = {
      name: "Counter",
      count: 0
    }
    console.log("Counter Constructor")
  }
}
```

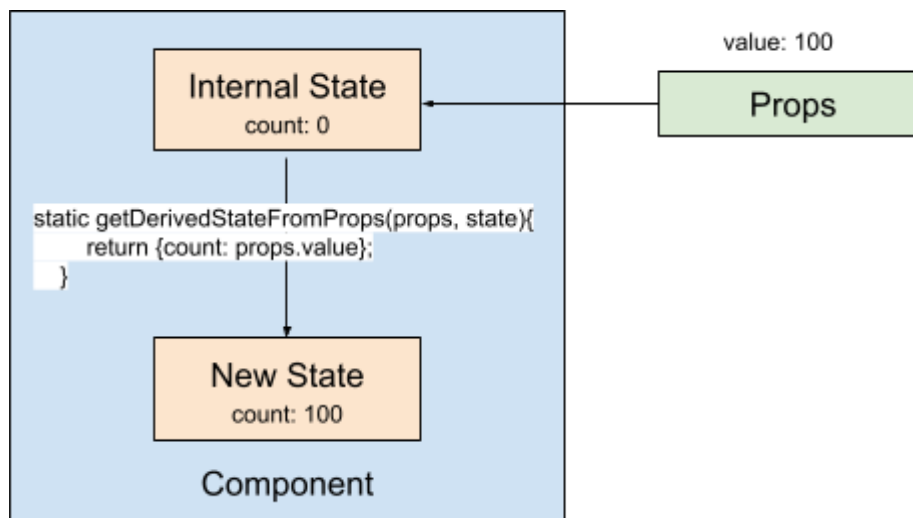
static getDerivedStateFromProps

- It is invoked right before the render function. This method is invoked in both the mounting and updating phases.

- This method exists for rare use cases where the state depends on changes in the props over time. If there is no change in state, then this method returns the null value.
- It is a static method that does not have any access to this keyword.

Example:

It will return an object to update the state. if the props.value is equal to state.counter then it will return a null value.



```

static getDerivedStateFromProps(props, state){
  console.log("Counter getDerivedStateFromProps");
  return {count: props.value};
}

```

render()

- This is the only required method in the class component. `render()` executes during both the mounting and updating phase of the component's lifecycle.
- It is used to render elements to the DOM by returning some JSX.
- The `render()` method must be a pure function, meaning it should not modify the component's state, as when the state gets updated, the render method gets automatically called, which could lead to infinite looping.
- `render()` will not be invoked if `shouldComponentUpdate()` returns false.

Example:

It returns JSX to render your UI and returns null value if there is nothing to render inside the component.

```
render() {  
  console.log("Counter Render")  
  return (  
    <h1>{this.state.count}</h1>  
  );  
}
```

componentDidMount

- It is invoked after a component is mounted. (initially renders on the screen).
- This method is a good place to handle side effects like setting up subscriptions and loading data from a remote endpoint.
- You can also use the `setState` function in this method to update the state.

Example:

When the state gets updated inside this method, it causes another rendering just before the browser updates the UI.

```
componentDidMount() {  
  console.log("Counter componentDidMount");  
  setTimeout(() => {  
    this.setState({count: 50});  
  }, 1000);  
}
```

Updating Phase

Changes to props or state can cause an update. These methods are called in the following order when a component is being re-rendered:

- `static getDerivedStateFromProps()`
- `shouldComponentUpdate()`
- `render()`
- `getSnapshotBeforeUpdate()`
- `componentDidUpdate()`

shouldComponentUpdate()

- `shouldComponentUpdate()` is called as soon as `static getDerivedStateFromProps()` the method is invoked.
- In the `shouldComponentUpdate()` method, you can return a Boolean value that controls whether the component gets rerendered upon a change in state/props. It defaults to true.
- This method only exists as a performance optimization. Do not rely on it to “prevent” a rendering, as this can lead to bugs.

Example:

It returns the boolean value true or false if you want to re-render or not.

```
shouldComponentUpdate(nextProps, nextState) {  
  console.log("Counter shouldComponentUpdate");  
  if(this.state.count == nextState.count) {  
    return false;  
  }  
  return true;  
}
```

`getSnapshotBeforeUpdate()`

- `getSnapshotBeforeUpdate()` is invoked just after the `render()` method.
- It stores the previous values of the state after the DOM is updated, meaning that even after the update, you can check what the values were before the update. Any value returned by this lifecycle method will be passed as a parameter to `componentDidUpdate()`.
- Most likely, you'll rarely reach for this lifecycle method. But it comes in handy when you need to grab information from the DOM (and potentially change it) just after an update is made, like a chat thread that needs to handle the scroll position.
- A snapshot value (or null) should be returned.

Example:

```
getSnapshotBeforeUpdate(prevProps, prevState) {  
  console.log("Counter getSnapshotBeforeUpdate");  
  return prevState.time || null;  
}
```

componentDidUpdate()

- `componentDidUpdate()` is invoked immediately after updating occurs.
- It can operate on the DOM when the component has been updated.
- This is also a good place to do network requests as long as you compare the current props to previous props (e.g., a network request may not be necessary if the props have not changed).
- It is similar to `componentDidMount()` as you can use `setState()` or fetch API call but you have to mention a condition to check if the previous state or props has changed or not.

Example:

```
getSnapshotBeforeUpdate(prevProps, prevState) {
  console.log("Counter getSnapshotBeforeUpdate");
  return prevState.count || null;
}

componentDidUpdate(prevProps, prevState, snapshot) {
  console.log("Counter componentDidUpdate");
  if (snapshot !== null) {
    this.setState({ count: 20 });
  }
}
```

Unmounting Phase

This method is called when a component is being removed from the DOM:

- `componentWillUnmount()`

componentWillUnmount()

- It is invoked immediately before a component is unmounted and destroyed.
- Perform any necessary cleanup in this method, such as invalidating timers, canceling network requests, or cleaning up any subscriptions that were created in `componentDidMount()`.
- You should not call `setState()` in `componentWillUnmount()` because the component will never be re-rendered. Once a component instance is unmounted, it will never be mounted again.

Example:

```
componentWillUnmount() {  
  if (this.count) {  
    clearInterval(this.timer);  
  }  
}
```

Error Handling Phase

These methods are called when an error occurs during rendering, in a lifecycle method, or the constructor of any child component.

- `static getDerivedStateFromError()`
- `componentDidCatch()`

`static getDerivedStateFromError()`

- This lifecycle is invoked after a descendant component has thrown an error.
- It receives the error thrown as a parameter and should return a value to update the state.
- `getDerivedStateFromError()` is called during the “render” phase, so side effects are not permitted. For those use cases, use `componentDidCatch()` instead.

Example:

whenever an error is thrown in a descendant component, the error will be logged to the console, `console.error(error)`, and an object is returned from the `getDerivedStateFromError` method. This will be used to update the state of the `ErrorBoundary` component i.e., with `hasError: true`.

```
static getDerivedStateFromError(error){  
  console.log("Error:", error);  
  return{  
    hasError: true,  
    error: error  
  }  
}
```

`componentDidCatch()`

- This lifecycle is invoked after a descendant component has thrown an error. It receives two parameters:
 - **error** - The error that was thrown.
 - **info** - An object with a `componentStack` key containing information about which component threw the error.
- `componentDidCatch()` is called during the “commit” phase, so side effects are permitted. It should be used for things like logging errors.

Example:

```
componentDidCatch(error, info){  
  console.log("Error:", error);  
  console.log("Error Info: ", info);  
}
```

Error boundaries

Error boundaries are React components that catch JavaScript errors anywhere in their child component tree, log those errors, and display a fallback UI instead of the component tree that crashed.

Example:

```
import { Component } from "react";  
class ErrorBoundary extends Component {  
  constructor() {  
    super();  
    this.state = {  
      hasError: false,  
      error: ""  
    }  
  }  
}  
  
static getDerivedStateFromError(error){  
  return{  
    hasError: true,  
    error: error  
  }  
}  
  
componentDidCatch(error, info){
```

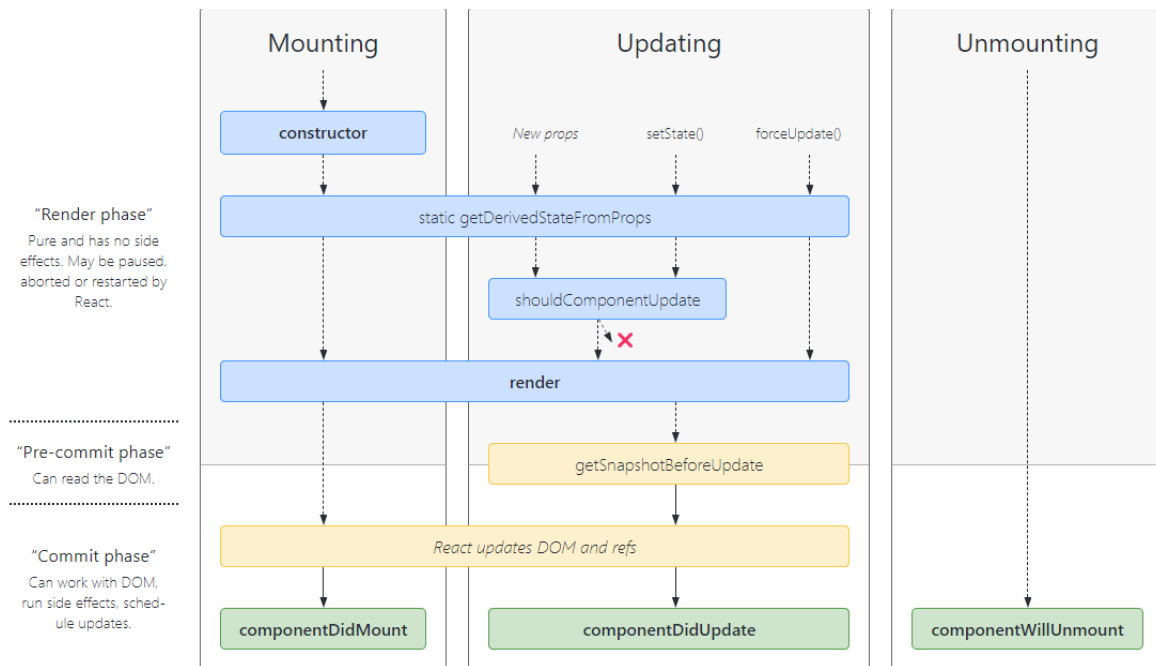
```
        console.log("Error:", error);
        console.log("Error Info: ", info);
    }

    render() {
        if(this.state.hasError){
            return (
                <h1>Something Went Wrong with {this.props.children.type.name}!
<br/> Please contact the admin</h1>
            );
        }
        return this.props.children;
    }
}

export default ErrorBoundary;
```

Summary

React follows a proper path of calling all lifecycle methods, which are called in different phases of a component's lifetime. Starting before the component is created, when the component is mounted, updated, or unmounted, and finally, during error handling. They all have roles, but some are used more often than others. The only required method in the whole lifecycle is the `render()` method.



Summarising it

Let's summarise what we have learned in this Lecture:

- Learned about the component lifecycle.
- Learned about different phases of a lifecycle.
- Learned about lifecycle methods in different phases.
- Learned about the order in which lifecycle methods are called during execution.
- Learned about how and where to perform side effects.

Some References:

- Component Lifecycle: [link](#)
- React Lifecycle Methods: [link](#)

React Hooks

Introduction

What are hooks?

Hooks are a new addition in React 16.8. They are functions that let you manage the internal state of component and handle post rendering side effects.

Note: Before we continue, note you can try Hooks in a few components without rewriting any existing code and hooks are 100% backwards-compatible. They don't contain any breaking changes.

Motivation to use hooks

Hooks can be used to address the following problems caused by using class based components:

1. Hard to reuse stateful logic between components

Hooks provide a way to separate stateful logic from a component, which allows for independent testing and reusability. With Hooks, you can reuse this logic in multiple components without restructuring the component hierarchy. This flexibility makes it easy to share Hooks with others and promote community collaboration.

2. Complex components become difficult to understand

In class-based components, related code often gets mixed with unrelated code, which can lead to bugs and inconsistencies. Hooks address this issue by allowing you to break down a component into smaller functions that handle related pieces of logic, like setting up a subscription or fetching data. This approach is more intuitive than forcing a split based on lifecycle methods.

3. Classes can be confusing at times

Classes in React can be a significant hurdle for learning the framework. Not only do you have to understand how "this" works in JavaScript, which can be quite different from other languages, but they can also complicate code reuse and organization. Hooks provide a simpler way to use React's features without relying on classes. By embracing functions, Hooks allow for a more intuitive and functional programming style that is closer to the conceptual nature of React components.

Rules of Hooks

The two main rules of Hooks in React are:

1. Only call Hooks at the top level:

This rule states that Hooks should only be called at the top level of a function component or another custom Hook. They should not be called inside loops, conditions, or nested functions, as this can lead to unexpected behavior and bugs.

Example 1: Incorrectly using hooks inside a function

```
const showAlert = () => {  
  
    useEffect(() => {  
  
        alert("Dependency has changed");  
  
    }, [dependency]);  
  
}
```

Example 2: Correctly using hooks at the top level of the component

```
useEffect(() => {  
  
    alert("Dependency has changed");  
  
}, [dependency]);
```

2. Only call Hooks from React function components:

This rule states that Hooks can only be used in React function components or other custom Hooks. They should not be used in class components or regular JavaScript functions, as this can cause errors or crashes.

Example 1: Incorrectly using hooks inside a class based component

```
class Example extends Component{  
  
    useEffect(() => {  
  
        alert("Dependency has changed");  
  
    }, [dependency]);  
  
}
```

Example 2: Correctly using hooks inside a functional component

```
const Example = () => {  
  
    const [dependency, setDependency] = useState("");  
  
    useEffect(() => {  
  
        alert("Dependency has changed");  
  
    }, [dependency]);  
  
};
```

Adhering to these rules ensures that Hooks work as intended and can help to prevent common issues and errors when working with React.

Order of Hooks

In React, hooks are executed in the order in which they are written. The order of Hooks is important. The order in which Hooks are called within a component must always be the same, so that React can correctly associate state and props with each Hook.

State in Function & Class based components

State management in functional and class-based React components is essentially the same, but the syntax and implementation details differ.

Syntax:

Class-based components have lifecycle methods, such as **componentDidMount** and **componentDidUpdate**, which allow for fine-grained control over the state and behavior of the component. Functional components have **hooks** that let you hook into the state and lifecycle features of the component but the syntax and implementation are different.

Updates and Side effects:

In class-based components, state is updated via the **setState** method. In functional components with hooks, state is updated via the update function returned by the **useState** hook. Side effects in functional components are managed using the **useEffect** hook which is a replacement for the lifecycle methods used in class-based components, such as **componentDidMount** and **componentDidUpdate**.

Boilerplate:

Class-based components require more boilerplate code to manage state and lifecycle methods, which can make the code more verbose and harder to read. Functional components with hooks have less boilerplate code, making them more concise and easier to read.

The **useState** hook

useState is a React Hook that lets you add a state variable to your component.

Parameters

1. **Initial State:** The **useState** hook in React takes an initial state as a parameter.

The initial state can be a value of any type. If a function is passed as the initial state, it will be treated as an initializer function. The initializer function should be a pure function, taking no arguments, and returning a value of any type. React will call the initializer function when initializing the component and store the return value as the initial state.

The initial state is only used during the first render and subsequent calls to **useState** with a new initial state will override the previous initial state.

Returns

The **useState** hook returns an array with exactly two values:

1. The current state. During the first render, it will match the **initialState** you have passed.
2. The set function that lets you update the state to a different value and trigger a re-render.

Example: Usage **useState** hook

```
const [count, setCount] = useState(0);
```

This code snippet uses the **useState** hook to define a state variable named **count** with an initial value of 0, and a function named **setCount** that can be used to update the state.

The set function returned by **useState** lets you update the state to a different value and trigger a re-render. You can pass the next state directly, or a function that calculates it from the previous state:

Example 1: Basic usage of state setter function

```
const [count, setCount] = useState(0);  
setCount(1)
```

Example 2: Passing a callback function to set the state

```
const [count, setCount] = useState(0);  
setCount((prevCount) => prevCount + 1);
```

Note: Using the state setter function with a callback allows you to access the latest state value at the time of the update and perform calculations or updates based on that value. This ensures that the state is always updated correctly and consistently, regardless of the timing of the updates.

The `useEffect` hook

useEffect is a React Hook that lets you synchronize a component with an external system.

Parameters

- 1. Setup:** The **useEffect** hook in React takes a function as its first argument, which contains the logic of your effect. This function may also return a cleanup function. When your component is initially added to the DOM, React will execute the setup function you provided in the `useEffect` hook. On subsequent re-renders React will first call the cleanup function with the previous values. After that, React will run your setup function with the new values.
Finally, when your component is removed from the DOM, React will execute your cleanup function one last time. This ensures that your component's side effects are properly added and removed throughout its lifecycle.
- 2. Options (dependencies):** The **useEffect** hook in React takes a dependency array as its second argument, which contains all the reactive values referenced inside the setup code. These values include props, state, and all variables and functions declared directly in the component body. The list of

dependencies must have a constant number of items and be written inline in the form of **[dep1, dep2, dep3]**.

React compares each dependency with its previous value using a comparison algorithm. If you don't provide the dependency array, your effect will run after every re-render of the component.

Returns

The **useEffect** hook does not return a value.

Example 1: Usage of useEffect hook

```
useEffect(() => {  
  setInterval(() => {  
    setTimer((prev) => prev++);  
  }, 1000);  
}, []);
```

This code snippet uses the **useEffect** hook to set an interval that increments the state value of timer after every second. The effect runs only once on mount due to an empty dependency array.

Example 2: Usage of useEffect hook (with a cleanup function)

```
useEffect(() => {  
  const interval = setInterval(() => {  
    setTimer((prev) => prev++);  
  }, 1000);  
  
  return () => clearInterval(interval)  
}, []);
```

This code snippet uses the **useEffect** hook to create a timer that updates every second. It sets up an interval to update the timer and clears it on unmount using the cleanup function. The effect runs only once on mount due to an empty dependency array.

The `useReducer` hook

`useReducer` is a React Hook that lets you add a reducer to your component. It is typically used when you have complex state transitions that involve multiple sub-values or when the next state depends on the previous state.

It is a more powerful alternative to the `useState` hook and is particularly useful when managing state for large or deeply nested objects. The `useReducer` hook provides a simple API for dispatching actions and updating state in a predictable way.

Parameters

1. **reducer:** In React, the `useReducer` hook takes a pure reducer function as its first argument, which defines how the state gets updated. The reducer function should take in the current state and an action as arguments and return the new state. The state and action can be of any type.
2. **initialState:** The value that represents the initial state of the component. This can be any value, including an object or an array.

Returns

`useReducer` returns an array with exactly two values:

1. The current state. During the first render, it's set to the `initialState`.
2. The dispatch function that lets you update the state to a different value and trigger a re-render.

Example: Usage of `useReducer` hook

```
const [state, dispatch] = useReducer(reducer, initialState);
```

This code snippet uses the `useReducer` hook to define a state variable named `state` with an initial value of `initialState`, and a function named `dispatch` that can be used to dispatch updates to the state.

The dispatch function

The **dispatch** function returned by **useReducer** lets you update the state to a different value and trigger a re-render. You need to pass the **action** as the only argument to the dispatch function

Example:

```
const [timer, dispatch] = useReducer(reducer, initialState)

const handleIncrement = () => {
  dispatch({ type: "INCREMENT_COUNT" });
};
```

This code snippet uses the **dispatch** function from the **useReducer** hook and passes an **action** object of type "INCREMENT_COUNT". The reducer function then checks this action type to update the state of the timer.

Writing the reducer function

The **reducer** function used in useReducer hook of React is a pure function that takes the current state and an action as arguments, and returns the new state.

The reducer function evaluates the type of the action and updates the state based on the type of action.

Example:

```
const reducer = (state, action) => {
  switch (action.type) {
    case "incremented_age": {
      return {
        name: state.name,
        age: state.age + 1,
      };
    }
  }
}
```

```
case "changed_name": {
  return {
    name: action.nextName,
    age: state.age,
  };
}
default:
  return state;
}
```

Custom hooks

Custom hooks are functions in React that allow you to reuse stateful logic across multiple components. They follow the naming convention of starting with the word "use" and can be defined and used in the same way as the built-in hooks provided by React.

Custom hooks are a way to abstract and share logic that is not tied to a specific component, which makes your code more modular, easier to read and maintain. They can encapsulate complex stateful logic and make it easy to use in multiple components, without having to repeat the same code in each component.

Custom hooks can use other built-in hooks and can also be composed with other custom hooks, which makes it easy to create complex and reusable logic that can be used across different parts of your application.

Example: The **useLocalStorage** custom hook

```
import { useState, useEffect } from "react";

const useLocalStorage = (key, initialValue) => {
  const [value, setValue] = useState(() => {
    const storedValue = localStorage.getItem(key);
    return storedValue !== null ? JSON.parse(storedValue) : initialValue;
  });
```

```
useEffect(() => {  
  localStorage.setItem(key, JSON.stringify(value));  
}, [key, value]);  
  
return [value, setValue];  
};
```

This custom hook allows you to store and retrieve data in the browser's `localStorage`. It takes a key and initial value as arguments and returns a state **value** and a **setValue** function to update it.

Summary

We have discussed the advantages of using hooks in functional components instead of relying on class-based lifecycle methods. The three main hooks covered were **useState**, **useEffect**, and **useReducer**, with `useState` used for state management, `useEffect` for handling side effects, and `useReducer` for more complex state management. These hooks are a newer addition to React and make managing stateful logic in functional components easier.

Summarising it

Let's summarise what we have learned in this Lecture:

- What are hooks?
- Motivation to use hooks.
- Rules of hooks.
- Order of hooks.
- State in function and class based components.
- The `useState` hook and usage
- The `useEffect` hook and usage
- The `useReducer` hook and usage
- Custom hooks in React

Some References:

- Lifecycle methods vs Hooks in React: [link](#)
- Hooks in React: [link](#)
- Custom hooks: [link](#)

Introduction to Firebase

Storing Data

Why does data get lost after refresh?

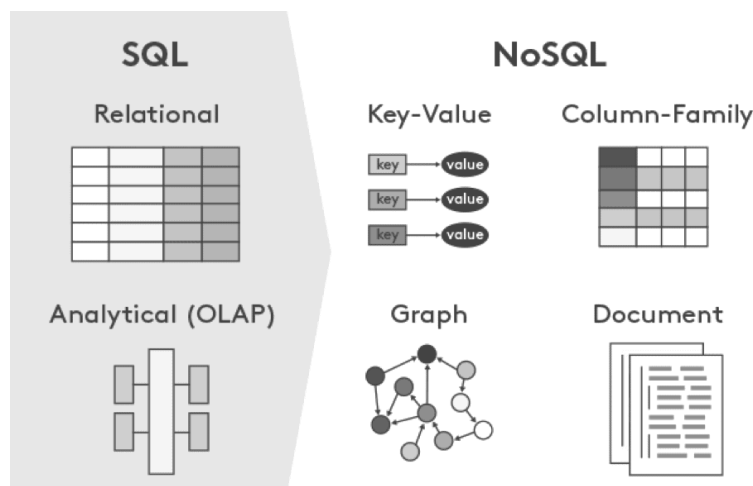
Here, we are storing the blog's data inside of the state locally in the form of an array. When you add a new blog, it gets added to the blogs array as well. But, when the page is reloaded, the App gets rerendered, and this array gets re-initialized to the empty array. So, This acts as temporary storage where data is not saved after refresh.

```
const [formData, setformData] = useState({title:"", content:""})
const [blogs, setBlogs] = useState([]);
```

Using Databases

A database is an organized collection of data for easy access, management and updating. To save stored data even after the refresh, you need to connect your React App with some external database. Databases can be classified into two categories:

- SQL Databases or Relational Databases
- No SQL Databases

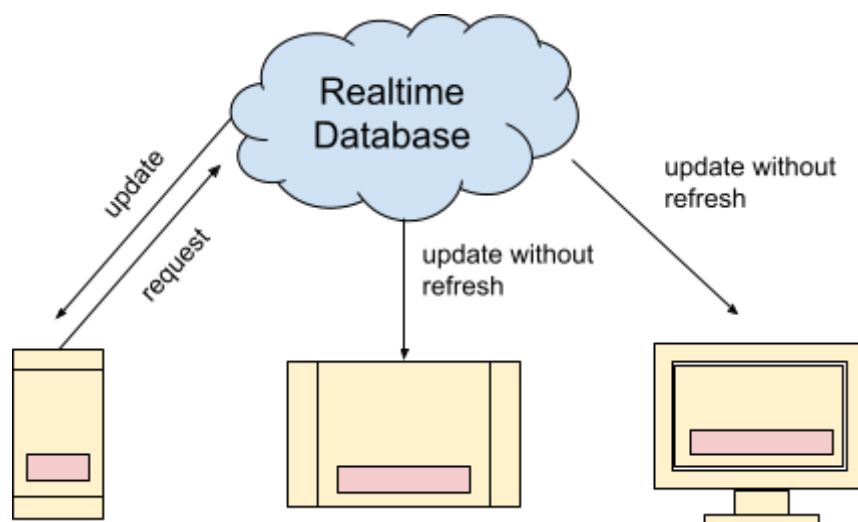


| Properties | SQL Databases | NoSQL Databases |
|----------------------------|---|---|
| Data Storage Model | Tables with fixed rows and columns | Document: JSON documents, Key-value: key-value pairs, Wide-column: tables with rows and dynamic columns, Graph: nodes and edges |
| Development History | Developed in the 1970s with a focus on reducing data duplication | Developed in the late 2000s with a focus on scaling and allowing for rapid application change driven by agile and DevOps practices. |
| Primary Purpose | General purpose and best suitable for structured, semi-structured, and unstructured data. | best suitable for structured data. Document: general purpose, Key-value: large amounts of data with simple lookup queries, Wide-column: large amounts of data with predictable query patterns, Graph: analyzing and traversing relationships between connected data |
| Schemas | Rigid | Flexible |
| Scaling | Vertical (scale-up with a larger server) | Horizontal (scale-out across commodity servers) |
| Examples | Oracle, MySQL, Microsoft SQL | Document: MongoDB and CouchDB, Key-value: Redis and DynamoDB, |

| | | |
|--|------------------------|---|
| | Server, and PostgreSQL | Wide-column: Cassandra and HBase, Graph: Neo4j and Amazon Neptune |
|--|------------------------|---|

Realtime Database

The Realtime database helps our users collaborate. It ships with mobile and web SDKs, allowing us to build our app without needing servers. When our users go offline, the Real-time Database SDKs use a local cache on the device for serving and storing changes. The local data is automatically synchronized when the device comes online.



Firebase

The Firebase Realtime Database is a cloud-hosted database in which data is stored as JSON. The data is synchronized in real-time to every connected client. Clients share one Realtime Database instance and automatically receive updates with the newest data when we build cross-platform applications with iOS and JavaScript SDKs. Firebase offers two cloud-based, client-accessible database solutions that support real-time data syncing:

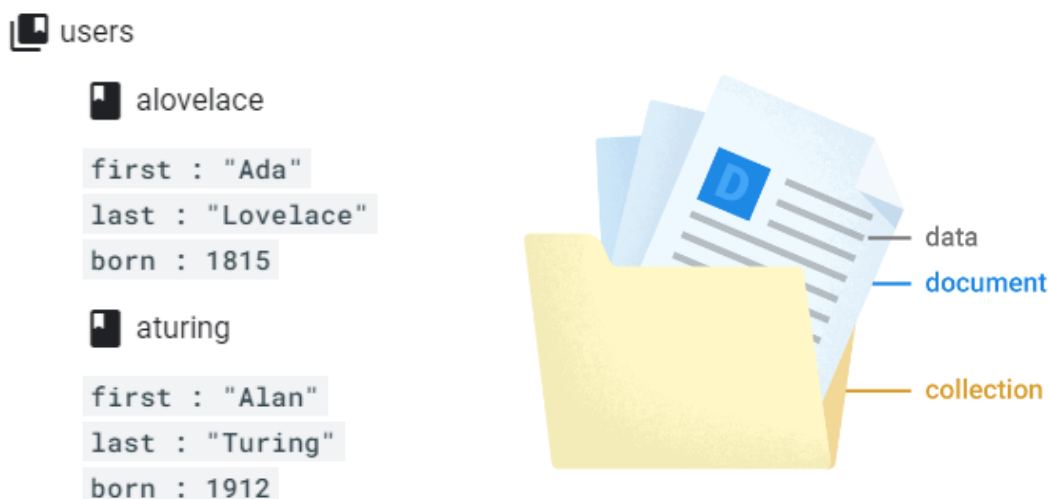
- **Cloud Firestore** is Firebase's newest database for mobile app development. It builds on the successes of the Realtime Database with a new, more intuitive data model. Cloud Firestore also features richer, faster queries and scales further than the Realtime Database. Data is stored in document format.

- **Realtime Database** is Firebase's original database. It's an efficient, low-latency solution for mobile apps requiring real-time synced states across clients. Data is stored in JSON format.

Cloud Firestore

In Cloud Firestore, the unit of storage is the document. A document is a lightweight record containing fields that map to values. Each document is identified by a name. Each document includes a set of key-value pairs. Cloud Firestore is optimized for storing extensive collections of small documents. Documents live in collections, which are simply containers for documents.

For example, you could have a users collection to contain your various users, each represented by a document:



Data types that Cloud Firestore supports are Array, Boolean, Bytes, Date and time, Floating-point number, Geographical point, Integer, Map, Null, Reference and a Text string.

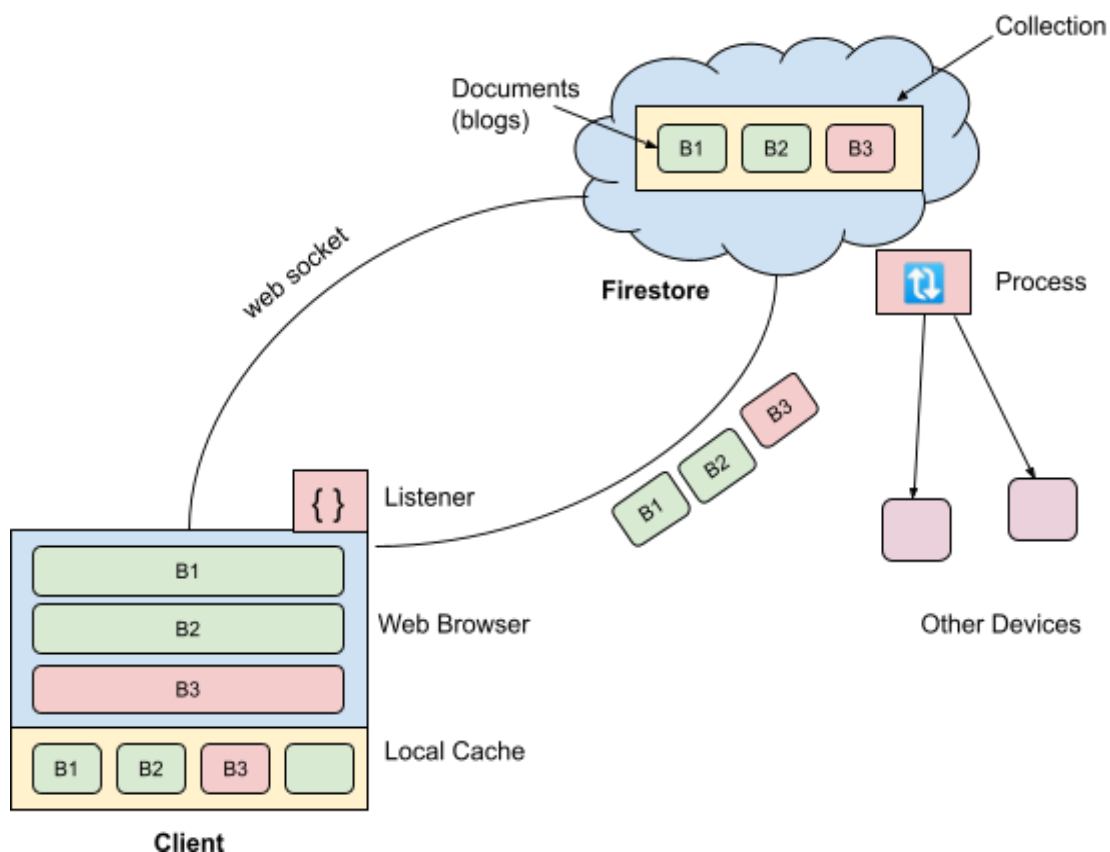
Understanding the working

Cloud Firestore caches data that your app is actively using, so the app can write, read, listen to, and query data even if the device is offline. When the device returns online, Cloud Firestore synchronizes any local changes back to Cloud Firestore. To keep data in your apps current without retrieving your entire database each time an update happens, **real-time listeners** are added. Adding real-time listeners to your

app notifies you with a data snapshot whenever the data your client apps are listening to changes, retrieving only the new changes.

For Example, Your firebase has a collection of blogs with blogs B1 and B2 as documents. As soon as the client opens the app, a persistent connection will be established between the firestore and the client via web socket. On the client side, the listeners installed, which are nothing but a call-back function, listen to any changes happening to the client. Similarly, there is a process inside cloud firestore, which listens to any changes happening in the database. These listeners are used to notify changes in the apps.

When we open the app for the first time, it is not directly updated to the UI. First, the data gets stored inside the local cache of the device. Here B1 and B2 will be stored already in the local cache. When a new document B3, is added, it will be added to the local cache, and then the listener will be notified. The Listener will send all the data from the local cache to the firebase, including the changes. Now, the Process present in firebase gets notified. The Process will notify all the other devices about the changes, and changes will get updated for all the devices. Only the new data or changes get updated.



Using Firestore in your Application

For more detailed steps, you can check this [link](#).

Create a Cloud Firestore Database

1. In the [Firebase console](#), click **Add project**, then follow the on-screen instructions to create a Firebase project. Enter a project name, then click Continue. Select your Firebase account from the dropdown or click Create a new account if you don't already have one. Click Continue once the process completes.
2. Next, click the Web icon (</>) towards the top-left of the following page to set up Firebase for the web. Enter a nickname for your app in the provided field. Then click the Register app.
3. Copy the generated code and keep it for the following step (discussed in the following section). Click Continue to the console.
4. Navigate to the **Cloud Firestore** section of the [Firebase console](#). Now, Follow the database creation workflow. Select a starting mode for your Cloud Firestore Security Rules:
 - **Test mode**
Good for getting started with the mobile and web client libraries, but allows anyone to read and overwrite your data.
 - **Locked mode**
Denies all reads and writes from mobile and web clients. Your authenticated application servers (C#, Go, Java, Node.js, PHP, Python, or Ruby) can still access your database.
5. Select a [location](#) for your database.
6. Click **Done**.

Initialize Firebase in Your React App

1. Install Firebase using npm:

```
npm install firebase
```

2. Create a firebaseinit.js file and paste the code generated earlier into this file.
You can also find this code in Project Overview > Project Settings.
3. Replace the TODOs with your app's Firebase project configuration
4. Export the firebase db object from the file and import this object into the files where it is needed.

Adding Data to Firebase

Add a document

When you use `set()` to create a document, you must specify an ID for the document to create. For example:

```
import { doc, setDoc } from "firebase/firestore";

await setDoc(doc(db, "cities", "new-city-id"), data);
```

But sometimes there isn't a meaningful ID for the document, and it's more convenient to let Cloud Firestore auto-generate an ID for you. You can do this by calling the following language-specific `add()` methods:

```
import { collection, addDoc } from "firebase/firestore";

// Add a new document with a generated id.
const docRef = await addDoc(collection(db, "cities"), {
  name: "Tokyo",
  country: "Japan"
});
console.log("Document written with ID: ", docRef.id);
```

For Blogs app the syntax will be:

```
const docRef = collection(db, "blogs");
await addDoc(docRef, {
  title: formData.title,
  content: formData.content,
  createdOn: new Date()
});
```

Set a document

To create or overwrite a single document, use the following language-specific set() methods:

```
import { doc, setDoc } from "firebase/firestore";

// Add a new document in collection "cities"
await setDoc(doc(db, "cities", "LA"), {
  name: "Los Angeles",
  state: "CA",
  country: "USA"
});
```

If the document does not exist, it will be created. If the document does exist, its contents will be overwritten with the newly provided data unless you specify that the data should be merged into the existing document, as follows:

```
import { doc, setDoc } from "firebase/firestore";

const cityRef = doc(db, 'cities', 'BJ');
setDoc(cityRef, { capital: true }, { merge: true });
```

setDoc is useful where you are generating IDs by yourself or adding a new one.

Fetching Data from the Database

Get Data

The following example shows how to retrieve the contents of a single document using get():

```
import { doc, getDoc } from "firebase/firestore";

const docRef = doc(db, "cities", "SF");
const docSnap = await getDoc(docRef);

if (docSnap.exists()) {
  console.log("Document data:", docSnap.data());
} else {
  // doc.data() will be undefined in this case
}
```

```
console.log("No such document!");  
}
```

You can also retrieve multiple documents with one request by querying documents in a collection. For example, you can use `where()` to query for all of the documents that meet a certain condition, then use `get()` to retrieve the results:

```
import { collection, query, where, getDocs } from "firebase/firestore";  
  
const q = query(collection(db, "cities"), where("capital", "==", true));  
  
const querySnapshot = await getDocs(q);  
querySnapshot.forEach((doc) => {  
  // doc.data() is never undefined for query doc snapshots  
  console.log(doc.id, " => ", doc.data());  
});
```

In addition, you can retrieve all documents in a collection by omitting the `where()` filter entirely:

```
import { collection, getDocs } from "firebase/firestore";  
  
const querySnapshot = await getDocs(collection(db, "cities"));  
querySnapshot.forEach((doc) => {  
  // doc.data() is never undefined for query doc snapshots  
  console.log(doc.id, " => ", doc.data());  
});
```

Data Sync - Getting Realtime updates

You can listen to a document with the `onSnapshot()` method. An initial call using the callback you provide creates a document snapshot immediately with the current contents of the single document. Then, each time the contents change, another call updates the document snapshot.

```
import { doc, onSnapshot } from "firebase/firestore";  
  
const unsub = onSnapshot(doc(db, "cities", "SF"), (doc) => {  
  console.log("Current data: ", doc.data());  
});
```

For Blogs App, we are using the following code:

```
useEffect(() => {
  async function fetchData(){
    const snapShot =await getDocs(collection(db, "blogs"));
    console.log(snapShot);

    const blogs = snapShot.docs.map((doc) => {
      return{
        id: doc.id,
        ...doc.data()
      }
    })
    console.log(blogs);
    setBlogs(blogs);
  }

  fetchData();
},[]);
```

Deleting the Documents from Database

To delete a document, use the following language-specific delete() methods:

```
import { doc, deleteDoc } from "firebase/firestore";
await deleteDoc(doc(db, "cities", "DC"));
```

For blogs app, we are using the following code:

```
async function removeBlog(id){
  const docRef = doc(db,"blogs",id);
  await deleteDoc(docRef);
}
```

Summarizing it

Let's summarize what we have learned in this Lecture:

- Learned about SQL and No SQL Databases.

- Learned about Realtime Databases.
- Learned about Cloud Firestore and configuration.
- Learned how to add and get data from Cloud Firestore.
- Learned how to display real-time updates.
- Learned how to delete documents from the database.

Some References:

- Realtime DB vs Cloud Firestore: [link](#)
- Cloud Firestore: [link](#)
- Data Model: [link](#)
- Firestore Configuration: [link](#)
- Add Data: [link](#)
- Get Data: [link](#)
- Get Real Time Updates: [link](#)
- Delete Documents: [link](#)