REACT JS

INSTALLATION OF REACT JS

Step 1: Install node js.
Step 2: open window powershell.

Step 3: for inital information.
npm init -y
Step 4: install react and its DOM packages.
npm install react react-dom --save

Step 5: To install webpack
npm install webpack webpack-dev-server webpack-cli --save
Step 6: Install Babel
npm install babel-core babel-loader babel-preset-env babel-preset-react babel-webpack-plugin --save-dev

1. npm install babel-core --save-dev
2. npm install babel-loader --save-dev
3. npm install babel-preset-env --save-dev
4. npm install babel-preset-react --save-dev
5. npm install babel-webpack-plugin --save-dev

Step 7: create files
touch index.html
touch App.js
touch main.js
touch webpack.config.js
touch .babelrc

Set Compiler, Loader, and Server for React Application

Configure webpack

webpack.config.json

```
1. const path = require('path');
2. const HtmlWebpackPlugin = require('html-webpack-plugin');
3.
4. module.exports = {
5.    entry: './main.js',
6.    output: {
7.       path: path.join(__dirname, '/bundle'),
8.    filename: 'index_bundle.js'9.
      },
10.      devServer: {
11.        inline: true,
12.        port: 8080
13.      },
14.      module: {
15.        rules: [
16.          {
17.            test: /\.jsx?$/,
18.            exclude: /node_modules/,
19.          use: {
20.              loader: "babel-loader",
21.            }
22.          }
23.        ]
24.      },
25.      plugins:[
26.        new HtmlWebpackPlugin({
27.        template: './index.html'
28.        })
29.      ]
30.   }
```

open the package.json file and delete "test" "echo \" Error: no test specified\" && exit 1" inside "scripts" object, then add the start and build commands instead. It is because we will not perform any testing in this app.

```
{
  "name": "reactApp",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "start": "webpack-dev-server --mode development --open --hot",
  "build": "webpack --mode production"
  },
    "keywords": [],
    "author": "",
    "license": "ISC",
    "dependencies": {
    "react": "^16.8.6",
      "react-dom": "^16.8.6",
      "webpack-cli": "^3.3.1",
      "webpack-dev-server": "^3.3.1"
    },
    "devDependencies": {
      "@babel/core": "^7.4.3",
      "@babel/preset-env": "^7.4.3",
      "@babel/preset-react": "^7.0.0",
      "babel-core": "^6.26.3",
      "babel-loader": "^8.0.5",
      "babel-preset-env": "^1.7.0",
      "babel-preset-react": "^6.24.1",
      "html-webpack-plugin": "^3.2.0",
      "webpack": "^4.30.0"
```

```
    }
  }
```

HTML webpack template for index.html

```
<!DOCTYPE html>
<html lang = "en">
  <head>
    <meta charset = "UTF-8">
    <title>React App</title>
  </head>
  <body>
    <div id = "app"></div>
    <script src = 'index_bundle.js'></script>
  </body>
 </html>

```

App.js

```
import React, { Component } from 'react';
class App extends Component{
  render(){
    return(
      <div>
        <h1>Hello World</h1>
      </div>8.
      );
  }
  }
  export default App;
```

Main.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App.js';
```

4.
5. ReactDOM.render(<App />, document.getElementById('app'));

Create .babelrc file

```
{
   "presets":[
  "@babel/preset-env", "@babel/preset-react"]
}
```

Running the Server: npm start

Generate the Bundle: npm run build

2. Using the create-react-app command

Install React

1. npm install -g create-react-app

Create a new React project

create-react-app jtp-reactapp

npx create-react-app jtp-reactapp

App.js

1. import React from 'react';

```
2. import logo from './logo.svg';
3. import './App.css';
4.
5. function App() {
6.   return (
7.     <div className="App">
8.       <header className="App-header">
9.         <img src={logo} className="App-logo" alt="logo" />
10.           <p>
11.             Welcome To  myheaven
12.
13.         <p>To get started, edit src/App.js and save to reload.</p>
14.           </p>
15.           <a
16.             className="App-link"
17.             href="https://reactjs.org"
18.             target="_blank"
19.             rel="noopener noreferrer"
20.           >
21.             Learn React
22.           </a>
23.         </header>
24.       </div>
25.     );
26.   }
27.
28.   export default App;
```


Running the Server: npm start

React Components

Class Components

Class components are more complex than functional components.

It requires you to extend from React.

Component and create a render function which returns a React element.

You can pass data from one class to other class components.

You can create a class by defining a class that extends Component and has a render function.

Valid class component is shown in the below example.

```
1. import React, { Component } from 'react';
2. class App extends React.Component {
3.  constructor() {
4.      super();
5.      this.state = {
6.        data:
7.        [
8.          {
9.            "name":"Abhishek"
10.           },
11.           {
12.             "name":"Saharsh"
13.           },
14.           {
15.             "name":"Ajay"
16.           }
17.         ]
18.       }
19.     }
20.     render() {
21.       return (
22.         <div>
```

```
            <StudentName/>
            <ul>
                {this.state.data.map((item) => <List data = {item}
  />)}
            </ul>
          </div>
        );
      }
    }
    class StudentName extends React.Component {
      render() {
        return (
          <div>
            <h1>Student Name Detail</h1>
        </div>
        );
      }
    }
    class List extends React.Component {
      render() {
        return (
          <ul>
            <li>{this.props.data.name}</li>
          </ul>
        );
      }
    }
    export default App;
```

EXAMPLE 2.

```
import React from 'react';
import ReactDOM from 'react-dom/client';
```

```
class Car extends React.Component {
  constructor() {
    super();
    this.state = {color: "red"};
  }
  render() {
    return <h2>I am a {this.state.color} Car!</h2>;
  }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Car />);
```

1. What are functional components in React?

Functional components are just javascript functions, which contains some logic to perform certain actions. These components accept the data as props and return the React element which is nothing HTML content. With introduction to the React 16, writing functional components is the standard way of creating components in modern react applications.

2. Are functional components better React?

Yes functional components are better in React, with the introduction to the React hooks, we can do so much of things in functional components, even we can use lifecycle methods inside the functional component. We can even create functional components using ES6 arrow functions.

3. Why use functional components React?

Using the react functional components, one can easily understand the code. It has several advantages, first of all, functional components are just simple javascript functions that accept the information in the form of prop and return the react element, which is HTML content. There is no need to use the render() method inside functional components.

4. How do you create a functional component of a React?

Creating the functional component is just like making the javascript functions, but it accepts the props and returns the react element. Take a look at the below code snippet

```
import React from 'react.'
export default function Hello() {
   return (
      <div>
         Hello World!
      </div>
```


5. What are the functional component and class component in React?

Functional component is just a simple javascript function; it accepts the data in the form of props and returns the react element. Whereas the class component will be created using the class keyword, and it extends the React. Component to make the class as a react component.

6. How do you call a functional component in React?

To call functional components first, we need to export that component using export statements to make it available. Next, we can call the components just like any other HTML elements. Check the below example for your reference.

```
Import React from 'react';
import Video from "./components/Video";
import './App.css';
function App() {
  return (
    <div className="App">
      <Video />
    </div>
  );
}
```

export default App;

React Bootstrap

Adding Bootstrap for React

npm install bootstrap --save

The three most common ways are given below:

1. Using the Bootstrap CDN
2. Bootstrap as Dependency
3. React Bootstrap Package

Using the Bootstrap CDN

1. <link rel="stylesheet"
   href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.min.css"
   integrity="sha384-ggOyR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/iJTQUOhcWr7x9JvoRxT2MZw1T"
   crossorigin="anonymous">

2. <script src="https://code.jquery.com/jquery-3.3.1.slim.min.js" integrity="sha384-q8i/X+965DzO0rT7abK41JStQIAqVgRVzpbzo5smXKp4YfRvH+8abtTE1Pi6jizo" crossorigin="anonymous"></script>

3. <script src="https://cdnjs.cloudflare.com/ajax/libs/popper.js/1.14.7/umd/popper.min.js" integrity="sha384-UO2eT0CpHqdSJQ6hJty5KVphtPhzWj9WO1clHTMGa3JDZwrnQq4sF86dIHNDz0W1" crossorigin="anonymous"></script>

4. <script src="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/js/bootstrap.min.js" integrity="sha384-JjSmVgyd0p3pXB1rRibZUAYoIIy6OrQ6VrjIEaFf/nJGzIxFDsf4x0xIM+B07jRM" crossorigin="anonymous"></script>

STYLE IN REACT JS

Inline Styles

Inline styles are the most direct away to style any React application.

Styling elements inline doesn't require you to create a separate stylesheet.

Style applied directly to the elements as compared to styles in a stylesheet also have higher precedence. This means that they "override" other style rules that may be applied to an element.

Here is our testimonial card styled with inline styles:

```
export default function App() {
  return (
    <section
      style={{
        fontFamily: '-apple-system',
        fontSize: "1rem",
        fontWeight: 1.5,
        lineHeight: 1.5,
        color: "#292b2c",
        backgroundColor: "#fff",
        padding: "0 2em"
      }}
    >
      <div
        style={{
          textAlign: "center",
          maxWidth: "950px",
          margin: "0 auto",
          border: "1px solid #e6e6e6",
          padding: "40px 25px",
          marginTop: "50px"
        }}
```

```
    >
      <img
        src="https://randomuser.me/api/portraits/women/48.jpg"
        alt="Tammy Stevens"
        style={{
          margin: "-90px auto 30px",
          width: "100px",
          borderRadius: "50%",
          objectFit: "cover",
          marginBottom: "0"
        }}
      />
      <div>
        <p
          style={{
            lineHeight: 1.5,
            fontWeight: 300,
            marginBottom: "25px",
            fontSize: "1.375rem"
          }}
        >
          This is one of the best developer blogs on the planet! I read it daily to improve my skills.
        </p>
      </div>
      <p
        style={{
          marginBottom: "0",
          fontWeight: 600,
          fontSize: "1rem"
        }}
      >
        Tammy Stevens
        <span style={{ fontWeight: 400 }}> · Front End Developer</span>
```

```
        </p>
      </div>
    </section>
  );

}
```

Despite a few quick benefits, inline styles are only an acceptable choice for very small applications. The difficulties with inline styles become apparent as your code base grows even slightly.

As the code example above shows, even a small component like this becomes very bulky if all the styles are inline.

One quick trick however is to put inline styles into reusable variables, which can be stored in a separate file:

```
const styles = {
  section: {
    fontFamily: "-apple-system",
    fontSize: "1rem",
    fontWeight: 1.5,
    lineHeight: 1.5,
    color: "#292b2c",
    backgroundColor: "#fff",
    padding: "0 2em"
  },
  wrapper: {
    textAlign: "center",
    maxWidth: "950px",
    margin: "0 auto",
    border: "1px solid #e6e6e6",
```

```
    padding: "40px 25px",
    marginTop: "50px"
  },
  avatar: {
    margin: "-90px auto 30px",
    width: "100px",
    borderRadius: "50%",
    objectFit: "cover",
    marginBottom: "0"
  },
  quote: {
    lineHeight: 1.5,
    fontWeight: 300,
    marginBottom: "25px",
    fontSize: "1.375rem"
  },
  name: {
    marginBottom: "0",
    fontWeight: 600,
    fontSize: "1rem"
  },
  position: { fontWeight: 400 }
};

export default function App() {
  return (
    <section style={styles.section}>
      <div style={styles.wrapper}>
        <img
          src="https://randomuser.me/api/portraits/women/48.jpg"
          alt="Tammy Stevens"
          style={styles.avatar}
        />
        <div>
          <p style={styles.quote}>
```

```
          This is one of the best developer blogs on the planet! I read it
          daily to improve my skills.
        </p>
      </div>
      <p style={styles.name}>
        Tammy Stevens
        <span style={styles.position}> · Front End Developer</span>
      </p>
    </div>
  </section>
 );

}
```

Despite this improvement, inline styles do not have a number of essential features that any simple CSS stylesheet could provide.

For example, you cannot write animations, styles for nested elements (i.e. all child elements, first-child, last-child), pseudo-classes (i.e. :hover), and pseudo-elements (::first-line) to name a few.

If you're prototyping an application, inline styles are great. However, as you get further into making it, you will need to switch to another CSS styling option to give you basic CSS features.

- Pros:

- Quickest way to write styles
- Good for prototyping (write inline styles then move to stylesheet)
- Has great preference (can override styles from a stylesheet)

▫ Cons:

- Tedious to convert plain CSS to inline styles
- Lots of inline styles make JSX unreadable
- You can not use basic CSS features like animations, selectors, etc.
- Does not scale well

Plain CSS

Instead of using inline styles, it's common to import a CSS stylesheet to style a component's elements.

Writing CSS in a stylesheet is probably the most common and basic approach to styling a React application, but it shouldn't be dismissed so easily.

Writing styles in "plain" CSS stylesheets is getting better all the time, due to an increasing set of features available in the CSS standard.

This includes features like CSS variables to store dynamic values, all manner of advanced selectors to select child elements with precision, and new pseudo-classes like :is and :where.

Here is our testimonial card written in plain CSS and imported at the top of our React application:

```
/* src/styles.css */

body {
```

```
   font-family: -apple-system, system-ui, BlinkMacSystemFont, "Segoe UI", Roboto, "Helvetica Neue", Arial, sans-serif;
  margin: 0;
  font-size: 1rem;
  font-weight: 1.5;
  line-height: 1.5;
  color: #292b2c;
  background-color: #fff;
}
.testimonial {
  margin: 0 auto;
  padding: 0 2em;
}
.testimonial-wrapper {
  text-align: center;
  max-width: 950px;
  margin: 0 auto;
  border: 1px solid #e6e6e6;
  padding: 40px 25px;
  margin-top: 50px;
}
.testimonial-quote p {
  line-height: 1.5;
  font-weight: 300;
  margin-bottom: 25px;
  font-size: 1.375rem;
}
.testimonial-avatar {
  margin: -90px auto 30px;
  width: 100px;
  border-radius: 50%;
  object-fit: cover;
  margin-bottom: 0;
}
.testimonial-name {
```

```
  margin-bottom: 0;
  font-weight: 600;
  font-size: 1rem;
}
.testimonial-name span {
  font-weight: 400;

}
```

```
// src/App.js

import "./styles.css";

export default function App() {
  return (
    <section className="testimonial">
      <div className="testimonial-wrapper">
        <img
          className="testimonial-avatar"
          src="https://randomuser.me/api/portraits/women/48.jpg"
          alt="Tammy Stevens"
        />
        <div className="testimonial-quote">
          <p>
              This is one of the best developer blogs on the planet! I read it
daily to improve my skills.
          </p>
        </div>
        <p className="testimonial-name">
          Tammy Stevens<span> · Front End Developer</span>
        </p>
      </div>
    </section>
  );
```

}

For our testimonial card, note that we are creating classes to be applied to each individual element. These classes all start with the same name testimonial-.

CSS written in a stylesheet is a great first choice for your application. Unlike inline styles, it can style your application in virtually any way you need.

One minor problem might be your naming convention. Once you have a very well-developed application, it becomes harder to think of unique classnames for your elements, especially when you have 5 divs wrapped inside each other.

If you don't have a naming convention you are confident with (i.e. BEM), it can be easy to make mistakes, plus create multiple classes with the same name, which leads to conflicts.

Additionally, writing normal CSS can be more verbose and repetitive than newer tools like SASS/SCSS. As a result, it can take a bit longer to write your styles in CSS versus a tool like SCSS or a CSS-in-JS library.

Plus, it's important to note that since CSS cascades to all children elements, if you apply a CSS stylesheet to a component it is not just scoped to that component. All its declared rules will be transferred to any elements that are children of your styled component.

If you are confident with CSS, it is definitely a viable choice for you to style any React application.

With that being said, there are a number of CSS libraries that give us all the power of CSS with less code and include many additional features that CSS will never have on its own (such as scoped styles and automatic vendor prefixing).

- Pros:

  - Gives us all of the tools of modern CSS (variables, advanced selectors, new pseudo-classes, etc.)
  - Helps us clean up our component files from inline styles

- Cons:

  - Need to setup vendor prefixing to ensure latest features work for all users
  - Requires more typing and boilerplate than other CSS libraries (i.e. SASS)
  - Any stylesheet cascades to component and all children; not scoped
  - Must use a reliable naming convention to ensure styles don't conflict

SASS / SCSS

What is SASS? SASS is an acronym that stands for: Syntactically Awesome Style Sheets.

SASS gives us some powerful tools, many of which don't exist in normal CSS stylesheets. It includes features like variables, extending styles, and nesting.

SASS allows us to write styles in two different kinds of stylesheets, with the extensions .scss and .sass.

SCSS styles are written in a similar syntax to normal CSS, however SASS styles do not require us to use open and closing brackets when writing style rules.

Here is a quick example of an SCSS stylesheet with some nested styles:

```
/* styles.scss */

nav {
  ul {
```

```
    margin: 0;
    padding: 0;
    list-style: none;
  }

  li { display: inline-block; }

  a {
    display: block;
    padding: 6px 12px;
    text-decoration: none;
  }

}
```

Compare this with the same code written in a SASS stylesheet:

```
/* styles.sass */

nav
  ul
    margin: 0
    padding: 0
    list-style: none

  li
    display: inline-block

  a
    display: block
    padding: 6px 12px
    text-decoration: none
```

Since this is not regular CSS, it needs to be compiled from SASS into plain CSS. To do so in our React projects, you can use a library like node-sass.

If you are using a Create React App project, to start using .scss and .sass files, you can install node-sass with npm:

```
npm install node-sass
```

Here is our testimonial card styled with SCSS:

```
/* src/styles.scss */

$font-stack: -apple-system, system-ui, BlinkMacSystemFont, "Segoe UI", Roboto, "Helvetica Neue", Arial, sans-serif;
$text-color: #292b2c;

%font-basic {
  font-size: 1rem;
}

body {
  @extend %font-basic;
  font-family: $font-stack;
  color: $text-color;
  margin: 0;
  font-size: 1rem;
  font-weight: 1.5;
  line-height: 1.5;
```

```
  background-color: #fff;
}

/* unchanged rules skipped */

.testimonial-name {
  @extend %font-basic;
  margin-bottom: 0;
  font-weight: 600;

  span {
    font-weight: 400;
  }

}
```

These styles give us the following features: variables, extending styles and nested styles.

Variables: You can use dynamic values by writing variables, just like in JavaScript, by declaring them with a $ at the beginning.

There are two variables that can be used in multiple rules: $font-stack, $text-color.

Extending / Inheritance: You can add onto style rules by extending them. To extend rules, you create your own selector which can be reused like a variable. The name of rules that you want to extend start with %.

The variable %font-basic is inherited by the rules body and .testimonial-name.

Nesting: Instead of writing multiple rules that begin with the same selector, you can nest them.

In .testimonial-name , we use a nested selector to target the span element within it.

You can find a working version of a React application with SCSS here.

- Pros:

  - Includes many dynamic CSS features like extending, nesting, and mixins
  - CSS styles can be written with much less boilerplate over plain CSS

- Cons:

  - Like plain CSS, styles are global and not scoped to any one component
  - CSS stylesheets is starting to include a number of features that SASS had exclusively, such as CSS variables (not necessarily a con, but worth noting)
  - SASS / SCSS often requires setup, such as installing the Node library node-sass

CSS Modules

CSS modules are another slight alternative to something like CSS or SASS.

What is great about CSS modules is that they can be used with either normal CSS or SASS. Plus, if you are using Create React App you can start using CSS modules with no setup at all.

Here is our application written with CSS modules:

```
/* src/styles.module.css */

body {
   font-family: -apple-system, system-ui, BlinkMacSystemFont, "Segoe UI", Roboto, "Helvetica Neue", Arial, sans-serif;
  margin: 0;
  font-size: 1rem;
  font-weight: 1.5;
  line-height: 1.5;
  color: #292b2c;
  background-color: #fff;
}

/* styles skipped */

.testimonial-name span {
  font-weight: 400;

}
```

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

```
export default function App() {
  return (
    <section className={styles.testimonial}>
      <div className={styles['testimonial-wrapper']}>
        <img
          src="https://randomuser.me/api/portraits/women/48.jpg"
          alt="Tammy Stevens"
          className={styles['testimonial-avatar']}
        />
        <div>
          <p className={styles['testimonial-quote']}>
            This is one of the best developer blogs on the planet! I read it
            daily to improve my skills.
          </p>
        </div>
        <p className={styles['testimonial-name']}>
          Tammy Stevens
          <span> · Front End Developer</span>
        </p>
      </div>
    </section>
  );

}
```


Our CSS file has the name .module in it before the extension .css. Any CSS module file must have the name "module" in it and end in the appropriate extension (if we are using CSS or SASS/SCSS).

What is interesting to note if we look at the code above is that CSS modules are written just like normal CSS, but are imported and used as if it were created as objects (inline styles).

The benefit of CSS modules is that it helps avoid our problem of class conflicts with normal CSS. The properties that we are referencing turn into unique classnames that cannot conflict with one another when our project is built.

Our generated HTML elements will look like this:

```
<p class="_styles__testimonial-name_309571057">
  Tammy Stevens

</p>
```

Plus, CSS modules solve the problem of global scope in CSS. As compared to our normal CSS stylesheets, CSS declared using modules to individual components will not cascade to child components.

Therefore, CSS modules are best to use over CSS and SASS to make sure classes don't conflict and to write predictable styles that only apply to one or another component.

- Pros:

- Styles are scoped to one or another component (unlike CSS / SASS)
- Unique, generated classnames ensure no style conflict
- Can use them immediately without setup in CRA projects
- Can be used with SASS / CSS

▫ Cons:

- Can be tricky to reference classnames
- May be a learning curve to use CSS styles like object properties

CSS-in-JS

Similar to how React allowed us to write HTML as JavaScript with JSX, CSS-in-JS has done something similar with CSS.

CSS-in-JS allows us to write CSS styles directly in our components' javascript (.js) files.

Not only does it allow you write CSS style rules without having to make a single .css file, but these styles are scoped to individual components.

In other words, you can add, change or remove CSS without any surprises. Changing one component's styles will not impact the styles of the rest of your application.

CSS-in-JS often makes use of a special type of JavaScript function called a tagged template literal. What's great about this is that we can still write plain CSS style rules directly in our JS!

Here's a quick example of a popular CSS-in-JS library, Styled Components:

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

const Button = styled.button`
```

```
  color: limegreen;
  border: 2px solid limegreen;
  font-size: 1em;
  margin:  1em;
  padding: 0.25em 1em;
  border-radius: 3px;

  &:hover {
    opacity: 0.9;
  }
`;

export default function App() {
  return (
    <div>
      <Button>Click me</Button>
    </div>
  );

}
```

Note a few things here:

1. You can write normal CSS styles, but can include nested styles and pseudo-classes (like hover).
2. You can associate styles with any valid HTML element, such as the button element above (see styled.button).
3. You can create new components with these associated styles. See how Button is used in our App component.

Since this is a component, can it be passed props? Yes! We can export this component and use it anywhere in our app we like, plus give it dynamic features through props.

Let's say that you want an inverted variant of Button above with an inverted background and text. No problem.

Pass the inverted prop to our second button and in Button, you can access all props passed to the component using the ${} syntax with an inner function.

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

const Button = styled.button`
  background: ${props => props.inverted ? "limegreen" : "white"};
  color: ${props => props.inverted ? "white" : "limegreen"};
  border: 2px solid limegreen;
  font-size: 1em;
  margin:  1em;
  padding: 0.25em 1em;
  border-radius: 3px;

  &:hover {
    opacity: 0.9;
  }
`;

export default function App() {
  return (
    <div>
      <Button>Click me</Button>
      <Button inverted>Click me</Button>
    </div>
  );
}
```

In the return of the function, you can select the inverted prop and use a ternary to conditionally determine the color of the background and text.

Here is the result:

There are a great deal more benefits to using a CSS-in-JS library to style your React applications (too many to cover here), some of which I will list below.

Be sure to check out the two most popular CSS-in-JS libraries for React: Emotion and Styled Components.

One downside to using a CSS-in-JS libraries is adding an additional library to your project. However, I would argue this is easily worth the improved developer experience you have when styling your React apps versus plain CSS.

- Pros:

  - CSS-in-JS is predictable – styles are scoped to individual components
  - Since our CSS is now JS, we can export, reuse, and even extend our styles through props
  - CSS-in-JS libraries ensure there are no styling conflicts by generating unique classnames for your written styles
  - No need to focus on naming conventions for your classes, just write styles!

- Cons:

  - Unlike plain CSS, you will need to install one or more third-party JavaScript libraries, which will add weight to your built project

Navbar

React Bootstrap 5 Navbar component

Documentation and examples for powerful, responsive navigation header - MDB navbar. Includes support for branding, navigation, and more, including support for our collapse plugin.

Basic example

A basic example of the navbar with the most common elements like link, search form, brand, and dropdown. All of them are explained in detail in the supported content section.

Note: this example uses color (bg-light) and spacing (my-2, my-lg-0, me-sm-0, my-sm-0) utility classes.

Navbar

React Bootstrap 5 Navbar component

Documentation and examples for powerful, responsive navigation header - MDB navbar. Includes support for branding, navigation, and more, including support for our collapse plugin.

Basic example

A basic example of the navbar with the most common elements like link, search form, brand, and dropdown. All of them are explained in detail in the supported content section.

Note: this example uses color (bg-light) and spacing (my-2, my-lg-0, me-sm-0, my-sm-0) utility classes.

Brand

- Home
- Link
- Dropdown
- Disabled

SEARCH

SHOW CODE

How it works

Here’s what you need to know before getting started with the navbar:

- Navbars require a wrapping .navbar with .navbar-expand{-sm|-md|-lg|-xl|-xxl} for responsive collapsing and color scheme classes.
- Navbars and their contents are fluid by default. Change the container to limit their horizontal width in different ways.
- Use our spacing and flex utility classes for controlling spacing and alignment within navbars.
- Navbars are responsive by default, but you can easily modify them to change that. Responsive behavior depends on our Collapse JavaScript plugin.
- Ensure accessibility by using a <nav> element or, if using a more generic element such as a , add a role="navigation" to every navbar to explicitly identify it as a landmark region for users of assistive technologies.
- Indicate the current item by using aria-current="page" for the current page or aria-current="true" for the current item in a set.

Supported content

Navbars come with built-in support for a handful of sub-components. Choose from the following as needed:

- .navbar-brand for your company, product, or project name.
- .navbar-nav for a full-height and lightweight navigation (including support for dropdowns).
- .navbar-toggler for use with our collapse plugin and other navigation toggling behaviors.
- Flex and spacing utilities for any form controls and actions.
- .navbar-text for adding vertically centered strings of text.
- .collapse.navbar-collapse for grouping and hiding navbar contents by a parent breakpoint.

Apart from listed above, navbar supports also components like breadcrumbs, forms, buttons, icons, flags, avatars, badges, and a few more.

Brand

The .navbar-brand can be applied to most elements, but an anchor works best as some elements might require utility classes or custom styles.

```
JSX
import React from 'react'; import { MDBContainer, MDBNavbar, MDBNavbarBrand } from 'mdb-react-ui-kit'; export default function App() { return ( <> <MDBNavbar light bgColor='light'> <MDBContainer fluid> <MDBNavbarBrand href='#'>Navbar</MDBNavbarBrand> </MDBContainer> </MDBNavbar> <br /> <MDBNavbar light bgColor='light'> <MDBContainer fluid> <MDBNavbarBrand tag="span" className='mb-0 h1'>Navbar</MDBNavbarBrand> </MDBContainer> </MDBNavbar> </> ); }
```

Adding images to the .navbar-brand will likely always require custom styles or utilities to properly size. Here are some examples to demonstrate.

- JSX

```
import React from 'react';
import {
  MDBContainer,
  MDBNavbar,
  MDBNavbarBrand
} from 'mdb-react-ui-kit';

export default function App() {
```

```
  return (
    <>
      <MDBNavbar light bgColor='light'>
        <MDBContainer>
          <MDBNavbarBrand href='#'>
            <img

src='https://mdbootstrap.com/img/logo/mdb-transaprent-noshadows.webp'
              height='30'
              alt=''
              loading='lazy'
            />
          </MDBNavbarBrand>
        </MDBContainer>
      </MDBNavbar>
    </>
  );
}
```

Nav

Navbar navigation links build on our .nav options with their own modifier class and require the use of toggler classes for proper responsive styling. Navigation in navbars will also grow to occupy as much horizontal space as possible to keep your navbar contents securely aligned.

Active states—with .active—to indicate the current page can be applied directly to .nav-links or their immediate parent .nav-items.

Please note that you should also add the aria-current attribute on the .nav-link itself.

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

```
import {
  MDBContainer,
  MDBNavbar,
  MDBNavbarBrand,
  MDBNavbarToggler,
  MDBNavbarNav,
  MDBNavbarItem,
  MDBNavbarLink,
  MDBCollapse
} from 'mdb-react-ui-kit';

export default function App() {
  const [showNav, setShowNav] = useState(false);

  return (
    <MDBNavbar expand='lg' light bgColor='light'>
      <MDBContainer fluid>
        <MDBNavbarBrand href='#'>Navbar</MDBNavbarBrand>
        <MDBNavbarToggler
          type='button'
          aria-expanded='false'
```

```
  aria-label='Toggle navigation'
  onClick={() => setShowNav(!showNav)}
>
  <MDBIcon icon='bars' fas />
</MDBNavbarToggler>
<MDBCollapse navbar show={showNav}>
  <MDBNavbarNav>
    <MDBNavbarItem>
      <MDBNavbarLink active aria-current='page' href='#'>
        Home
      </MDBNavbarLink>
    </MDBNavbarItem>
    <MDBNavbarItem>
      <MDBNavbarLink href='#'>Features</MDBNavbarLink>
    </MDBNavbarItem>
    <MDBNavbarItem>
      <MDBNavbarLink href='#'>Pricing</MDBNavbarLink>
    </MDBNavbarItem>
    <MDBNavbarItem>
                    <MDBNavbarLink disabled href='#' tabIndex={-1}
aria-disabled='true'>
        Disabled
```

```
            </MDBNavbarLink>
          </MDBNavbarItem>
        </MDBNavbarNav>
      </MDBCollapse>
    </MDBContainer>
  </MDBNavbar>
 );
}
```

Forms

Place various form controls and components within a navbar:

```
import React from 'react';
import {
 MDBContainer,
 MDBNavbar,
 MDBBtn,
 MDBInputGroup
} from 'mdb-react-ui-kit';

export default function App() {
 return (
  <MDBNavbar light bgColor='light'>
```

```
      <MDBContainer fluid>
        <MDBInputGroup tag="form" className='d-flex w-auto mb-3'>
                <input className='form-control' placeholder="Type query" aria-label="Search" type='Search' />
          <MDBBtn outline>Search</MDBBtn>
        </MDBInputGroup>
      </MDBContainer>
    </MDBNavbar>
  );
}
```

Immediate children elements in .navbar use flex layout and will default to justify-content: space-between. Use additional flex utilities as needed to adjust this behavior.

React Component Life-Cycle

In ReactJS, every component creation process involves various lifecycle methods. These lifecycle methods are termed as component's lifecycle. These lifecycle methods are not very complicated and called at various points during a component's life. The lifecycle of the component is divided into **four phases**. They are:

1. Initial Phase
2. Mounting Phase
3. Updating Phase

4. Unmounting Phase

Each phase contains some lifecycle methods that are specific to the particular phase. Let us discuss each of these phases one by one.

1. Initial Phase

It is the **birth** phase of the lifecycle of a ReactJS component. Here, the component starts its journey on a way to the DOM. In this phase, a component contains the default Props and initial State. These default properties are done in the constructor of a component. The initial phase only occurs once and consists of the following methods.

- **getDefaultProps()**

  It is used to specify the default value of this.props. It is invoked before the creation of the component or any props from the parent is passed into it.

- **getInitialState()**

  It is used to specify the default value of this.state. It is invoked before the creation of the component.

2. Mounting Phase

In this phase, the instance of a component is created and inserted into the DOM. It consists of the following methods.

- **componentWillMount()**

  This is invoked immediately before a component gets rendered

into the DOM. In the case, when you call **setState()** inside this method, the component will not **re-render**.

- **componentDidMount()**

  This is invoked immediately after a component gets rendered and placed on the DOM. Now, you can do any DOM querying operations.

- **render()**

  This method is defined in each and every component. It is responsible for returning a single root **HTML node** element. If you don't want to render anything, you can return a **null** or **false** value.

3. Updating Phase

It is the next phase of the lifecycle of a react component. Here, we get new **Props** and change **State**. This phase also allows to handle user interaction and provide communication with the components hierarchy. The main aim of this phase is to ensure that the component is displaying the latest version of itself. Unlike the Birth or Death phase, this phase repeats again and again. This phase consists of the following methods.

- **componentWillRecieveProps()**

  It is invoked when a component receives new props. If you want to update the state in response to prop changes, you should compare this.props and nextProps to perform state transition by using **this.setState()** method.

- **shouldComponentUpdate()**

  It is invoked when a component decides any changes/updation to the DOM. It allows you to control the component's behavior of updating itself. If this method returns true, the component will update. Otherwise, the component will skip the updating.

- **componentWillUpdate()**

  It is invoked just before the component updating occurs. Here, you can't change the component state by invoking **this.setState()** method. It will not be called, if **shouldComponentUpdate()** returns false.

- **render()**

  It is invoked to examine **this.props** and **this.state** and return one of the following types: React elements, Arrays and fragments, Booleans or null, String and Number. If shouldComponentUpdate() returns false, the code inside render() will be invoked again to ensure that the component displays itself properly.

- **componentDidUpdate()**

  It is invoked immediately after the component updating occurs. In this method, you can put any code inside this which you want to execute once the updating occurs. This method is not invoked for the initial render.

4. Unmounting Phase

It is the final phase of the react component lifecycle. It is called when a component instance is **destroyed** and **unmounted** from the DOM. This phase contains only one method and is given below.

- **componentWillUnmount()**

  This method is invoked immediately before a component is destroyed and unmounted permanently. It performs any necessary **cleanup** related task such as invalidating timers, event listener, canceling network requests, or cleaning up DOM elements. If a component instance is unmounted, you cannot mount it again.

Example

```
1. import React, { Component } from 'react';
2.
3. class App extends React.Component {
4.    constructor(props) {
5.       super(props);
6.       this.state = {hello: "yess infotech"};
7.    this.changeState = this.changeState.bind(this)8.
      }
9.    render() {
10.         return (
11.            <div>
```

```
12.             <h1>ReactJS component's Lifecycle</h1>
13.             <h3>Hello {this.state.hello}</h3>
14.             <button onClick = {this.changeState}>Click
    Here!</button>
15.         </div>
16.         );
17.     }
18.     componentWillMount() {
19.         console.log('Component Will MOUNT!')
20.     }
21.     componentDidMount() {
22.         console.log('Component Did MOUNT!')
23.     }
24.     changeState(){
25.         this.setState({hello:"All!!- Its a great reactjs tutorial."});
26.     }
27.     componentWillReceiveProps(newProps) {
28.         console.log('Component Will Recieve Props!')
29.     }
30.     shouldComponentUpdate(newProps, newState) {
31.         return true;
32.     }
```

```
33.     componentWillUpdate(nextProps, nextState) {
34.       console.log('Component Will UPDATE!');
35.     }
36.     componentDidUpdate(prevProps, prevState) {
37.       console.log('Component Did UPDATE!')
38.     }
39.     componentWillUnmount() {
40.       console.log('Component Will UNMOUNT!')
41.     }
42.   }
43.   export default App;
```

React Conditional Rendering

In React, we can create multiple components which encapsulate behavior that we need. After that, we can render them depending on some conditions or the state of our application. In other words, based on one or several conditions, a component decides which elements it will return. In React, conditional rendering works the same way as the conditions work in JavaScript. We use JavaScript operators to create elements representing the current state, and then React Component update the UI to match them.

From the given scenario, we can understand how conditional rendering works. Consider an example of handling a **login/logout** button. The login and logout buttons will be separate components. If a user logged

in, render the **logout component** to display the logout button. If a user not logged in, render the **login component** to display the login button. In React, this situation is called as **conditional rendering**.

There is more than one way to do conditional rendering in React. They are given below.

- if
- ternary operator
- logical && operator
- switch case operator
- Conditional Rendering with enums

if

It is the easiest way to have a conditional rendering in React in the render method. It is restricted to the total block of the component. IF the condition is **true**, it will return the element to be rendered. It can be understood in the below example.

Example

```
1. function UserLoggin(props) {
2.   return <h1>Welcome back!</h1>;
3. }
4. function GuestLoggin(props) {
5.   return <h1>Please sign up.</h1>;
6. }
```

```
function SignUp(props) {
  const isLoggedIn = props.isLoggedIn;
  if (isLoggedIn) {
    return <UserLogin />;
  }
  return <GuestLogin />;
}

ReactDOM.render(
  <SignUp isLoggedIn={false} />,
document.getElementById('root')
);
```

Logical && operator

This operator is used for checking the condition. If the condition is **true**, it will return the element **right** after **&&**, and if it is **false**, React will **ignore** and skip it.

Syntax

```
{
  condition &&
  // whatever written after && will be a part of output.
}
```

We can understand the behavior of this concept from the below example.

If you run the below code, you will not see the **alert** message because the condition is not matching.

```
('javatpoint' == 'JavaTpoint') && alert('This alert will never be shown!')
```

If you run the below code, you will see the **alert** message because the condition is matching.

```
(10 > 5) && alert('This alert will be shown!')
```

Example

```
import React from 'react';
import ReactDOM from 'react-dom';
// Example Component
function Example()
{
    return(<div>
            {
                (10 > 5) && alert('This alert will be shown!')
            }
              </div>
            );
    }
```

You can see in the above output that as the condition **(10 > 5)** evaluates to true, the alert message is successfully rendered on the screen

Parent to Child component communication in React

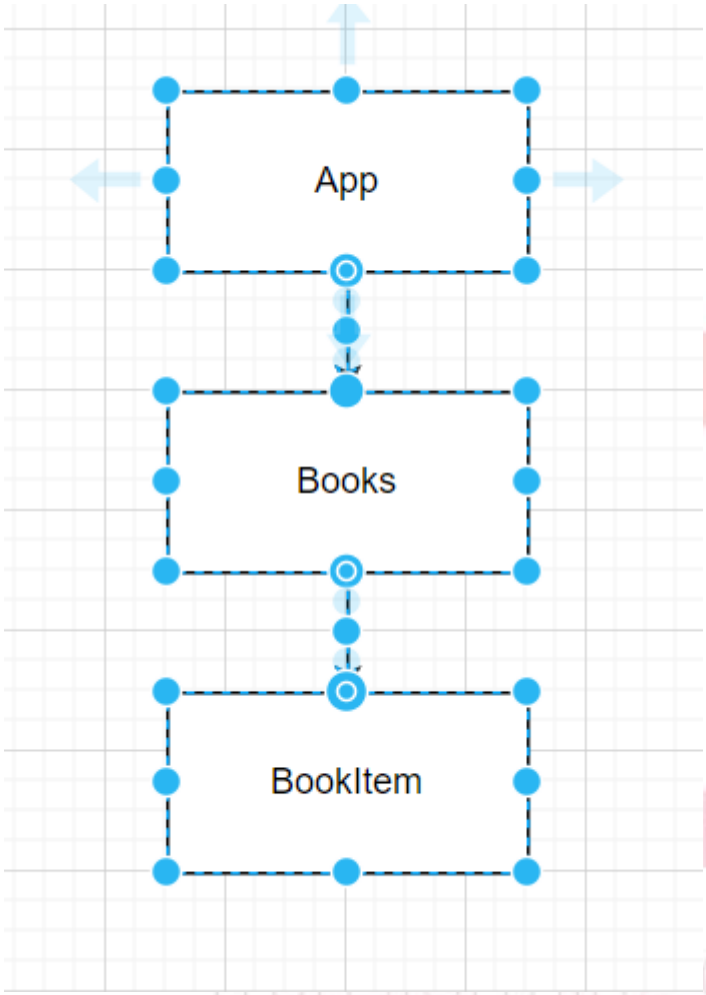
React is all about **Components** and we will be more often creating parent to child related components in most of the react applications which will require to pass data from parent to child components.

In React parent components can communicate to child components using a special property defined by React called as **Props**.

All the components in React will be having this property defined by default which will hold all the properties as key value pairs that are sent from the parent component.

Before understanding on how to use props to pass the data from parent component and use this data in child component , lets understand what is parent component and child component.

Lets say we want to display some list of books on UI and our component tree looks as below.



Component tree

App component is the root component of the application.

Books component is the wrapper component to display book items.

BookItem component is the reusable component which will display a single book item in the UI.

In the above component tree, App component is the parent component of the Books component and Books component is the parent component of the BookItem component.

Similarly, BookItem component is the child component of the Books component and Books component is the child component of the App component.

Why we need to pass data from parent component to child component?

For a BookItem component to display about the details of the book, the component has to be fed with book details and this data can be passed from its parent Books component using React feature called **props.**

Props is a feature of React which allows to pass data from parent component to child component (it holds all the attributes / properties which are passed to the component).

Lets discuss it with an example.

App component

In App component we have created a list of books and passing this list as **“books”** attribute to **Books** component to display the books in UI.

Books Component

In Books component , we are getting the books list passed from App component using props React feature as **“props.books”** and passing single book object to **BookItem** component to display book details on UI.

BookItem Component

In **BookItem** component we are accessing each book using props as **“props.book”** and displaying the details on UI.