# React Components-I

## Components in React

A component is a small, reusable chuck 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?

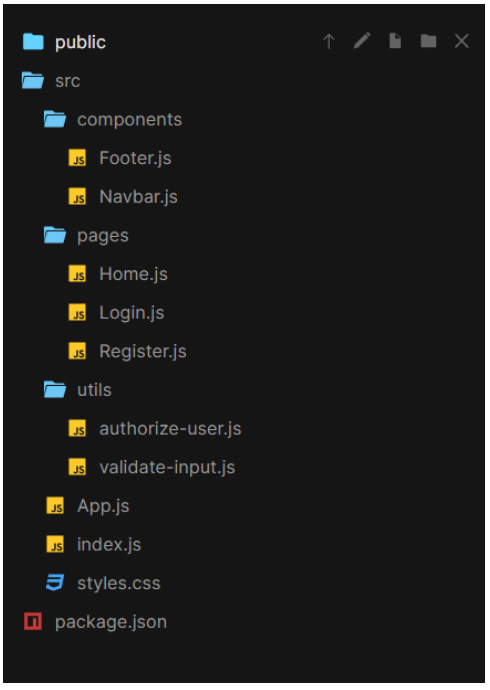
- If you prefer working with classes
- Still used in some legacy projects

## 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