Introduction to React

Topics covered:

- What is ReactJs?
- Why do we use ReactJs?
- What is SPA and MPA?
- What are the Prerequisites to learn ReactJs?
- How to Create a React App?
    - ★ Create-react-app tool
    - ★ What is Npm and Npx?
    - ★ How to run a react project?
- What is the Default structure of Project?
- Let's write our first program in react.

1. ReactJs:

React.Js is a JavaScript library for creating interactive and fast user interfaces. It was created by Facebook in 2011 and is now the most widely used JavaScript library for creating SPA and Reusable Components. According to Google, React is the most used JS library for designing user interfaces. React is just concerned with components. A React application is simply a collection of components.

- ReactJs uses JavaScript as it is a JavaScript library.
- Moreover it can use typescript also (strongly typed but similar to javascript).
- React is not a javascript's framework.
- ReactJs is used to make SPA(single page Application).
- ReactJs is used to create Reusable Components.

2. Why learn React?

- *Dynamic and Complex UI can be built easily using React* : React is simple to learn and easy to set up. Due to its component architecture, a highly dynamic and complex UI can be built easily and provides good performance.
- *Large Community Support* : React has a large community over Github, It is actively developed and maintained by Facebook and there are too many resources available worldwide.

3. SPA(Single Page Application) vs MPA(Multiple Page Application):

a. SPA(Single Page Application)

- A Single Page Application in an app that works inside a browser and does not require page reload during use.
- SPAs are all about serving an outstanding user experience (UX) by trying to imitate a “Natural” environment in the browser – no page reloads, no extra wait time.
- It is just one HTML page that you visit which then loads all other content using JavaScript.
- For Example: Gmail, Facebook, Instagram etc.

b. MPA(Multiple Page Application):

- A Multi-Page Application works in a traditional way like displaying data, submitting data back to the server, and sending requests to the server to render new pages in the browser.
- Whenever a user navigates from one page to another, a request is sent to the server to send a new HTML file for the URL. The server returns the file and then the HTML file is loaded in the browser.

4. Prerequisites:

- Basic Knowledge of HTML and Javascript.
- Basic Programming concepts like: Array, functions, objects, classes etc.
- Some of the ES6 Features.
- Like : Arrow function, Let, Const etc

5. How to create a React app:

When we create a react app there are a lot of libraries which are required to build a react application function properly such as React, React-dom, Babel etc.
There are 3 ways to build a react app:

1. Either you can install all the packages required and configure the entire project ourselves.
2. We can use the tool officially provided to build a react app and develop it accordingly.
3. Use an online code playground. Try CodeSandbox (codesandbox.io).
   how to use it - Navigate to the link; this will create the React project from scratch. In the center window, start making the changes in App.js. The changes will immediately reflect in the right window.

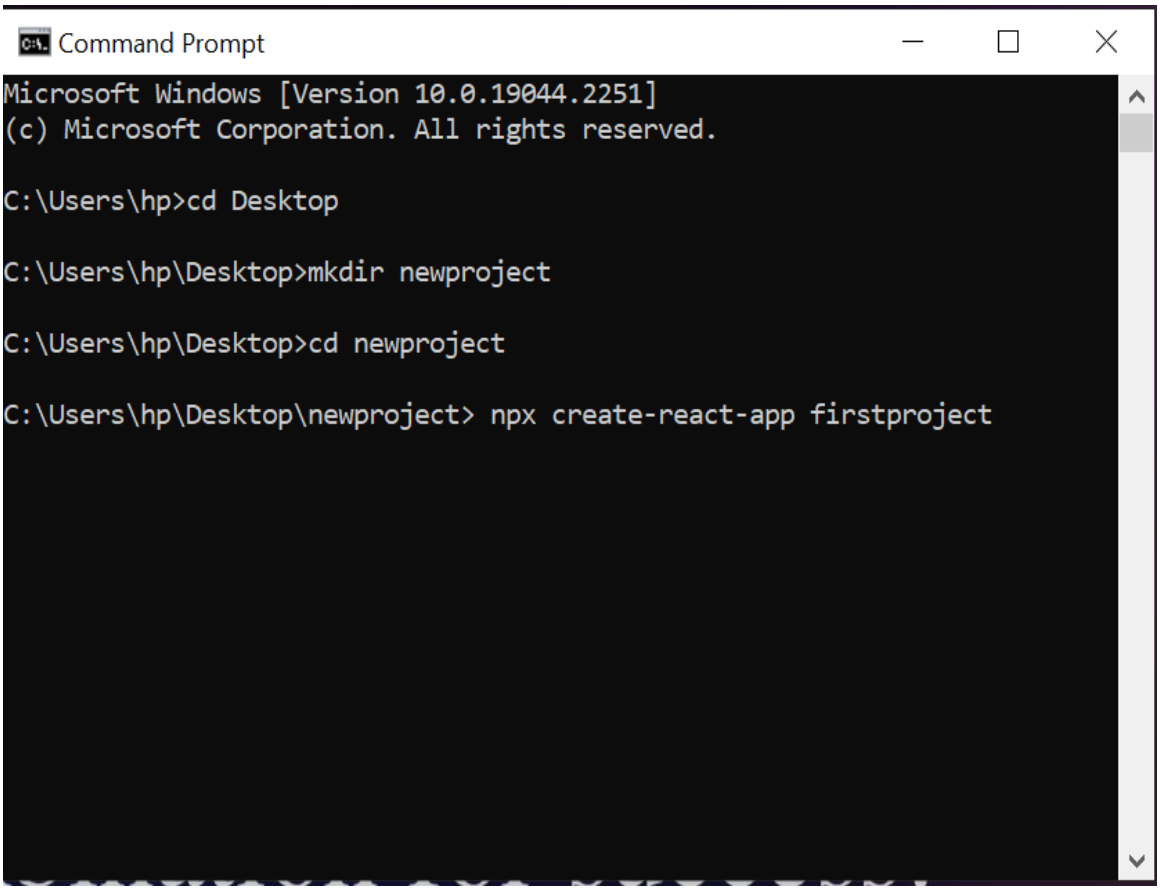
We are gonna use the second one:

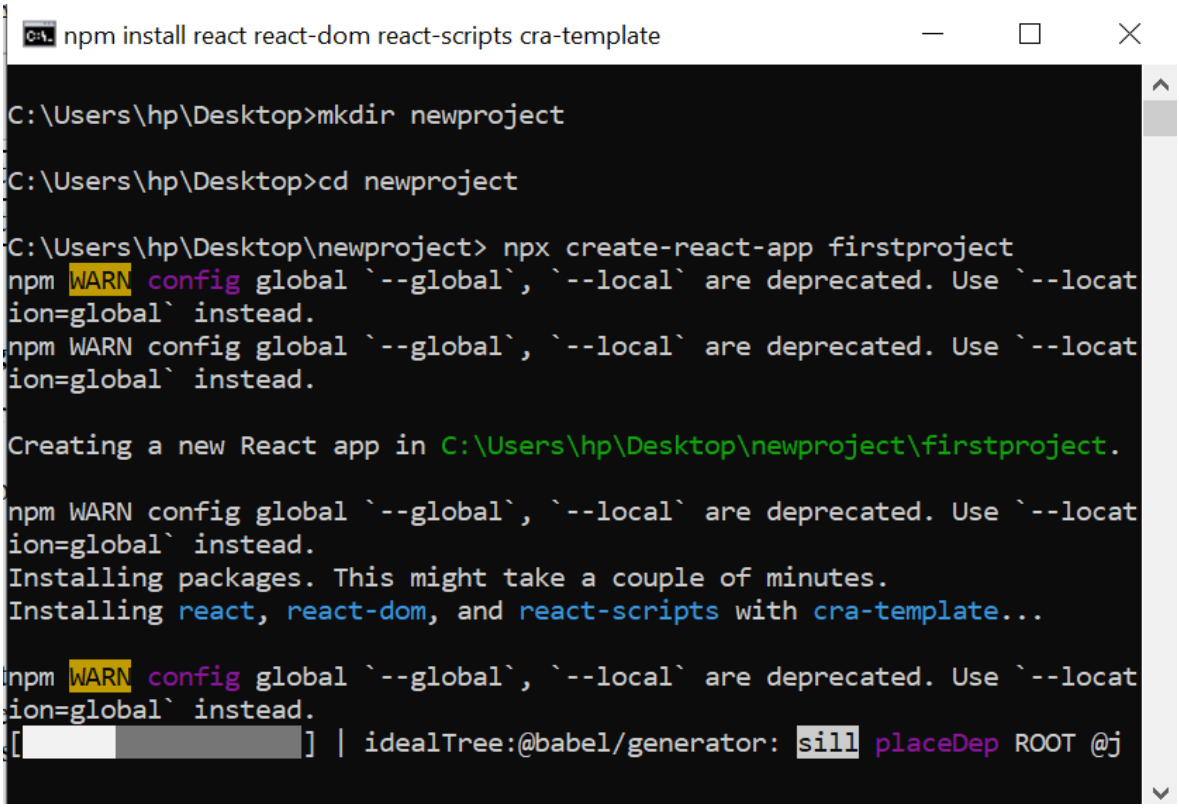
npx create-react-app firstproject

Must remember few things:

- Node must be there in your device.
  To check open the command prompt and check node -v and npm -v.
- If node is not downloaded : download it from - https://nodejs.org/en/download/
- The name of the project must be in small letters.
- Even the whole command must be in small letters.
- Write the command in the command prompt where you want to build a new react project template. (path must be accurate).

Just write this command in your terminal, like this:





a. **Create-react-app Tool**
   - The create-react-app Tool Installs all the dependencies and packages required to build a react application.
   - It creates a default configuration for our React Project.
   - It also adds some starter files in the newly created app.

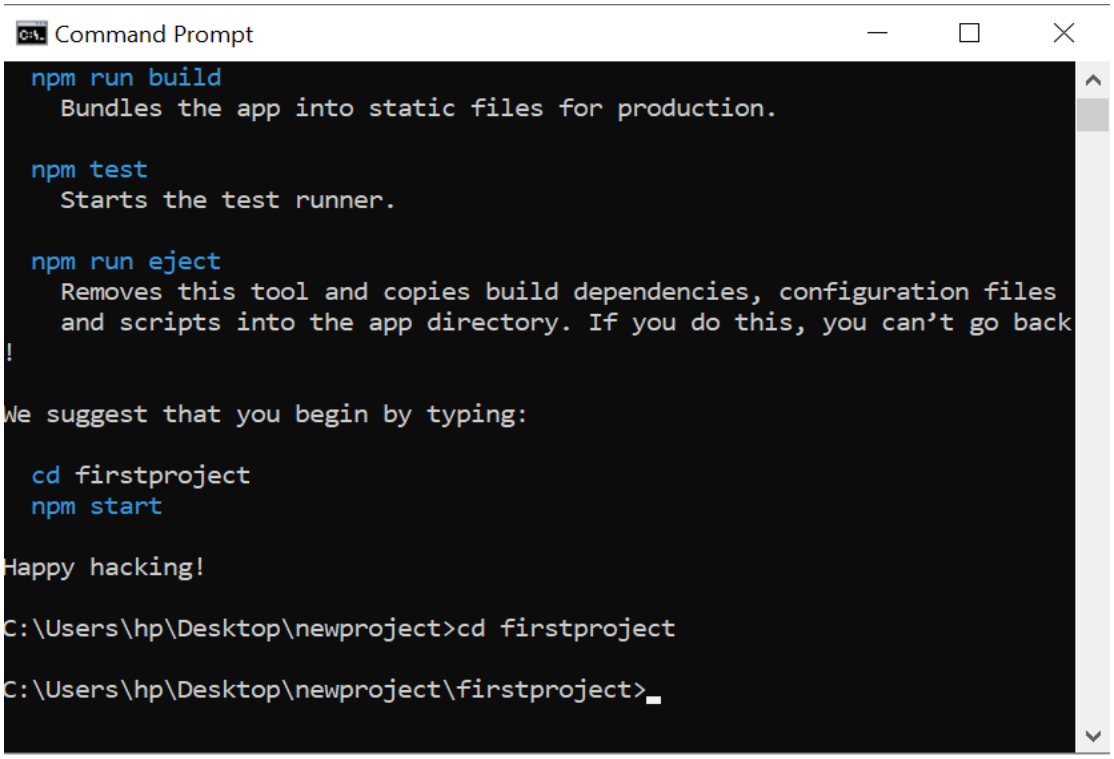
b. **Npm and Npx**

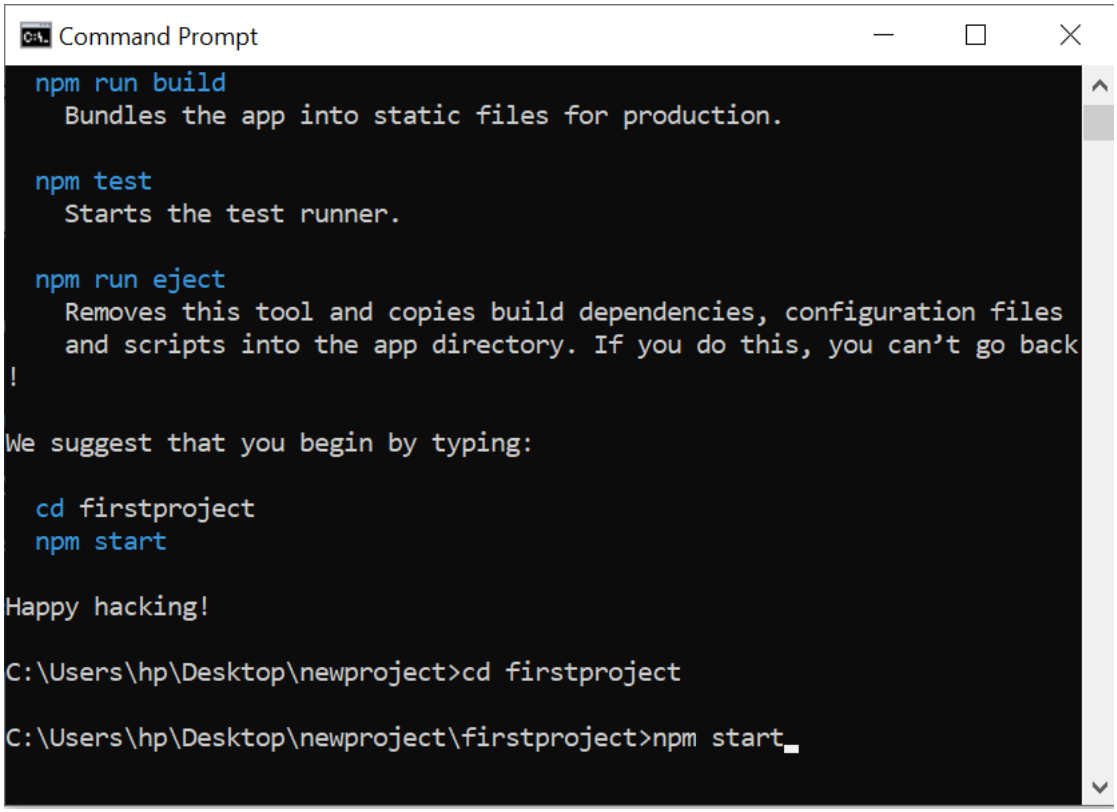
- Npm stands for node package manager.
- It helps to manage third party packages and dependencies that you will be installing you application.
- It is installed automatically when you install Node on your device.
- Npx is the node package executer or runner.
- It is used to download and run packages temporarily.
- Here we are using npx to create a react app because we will be using create react app once to just create an application so we don't need this package permanently in our project.

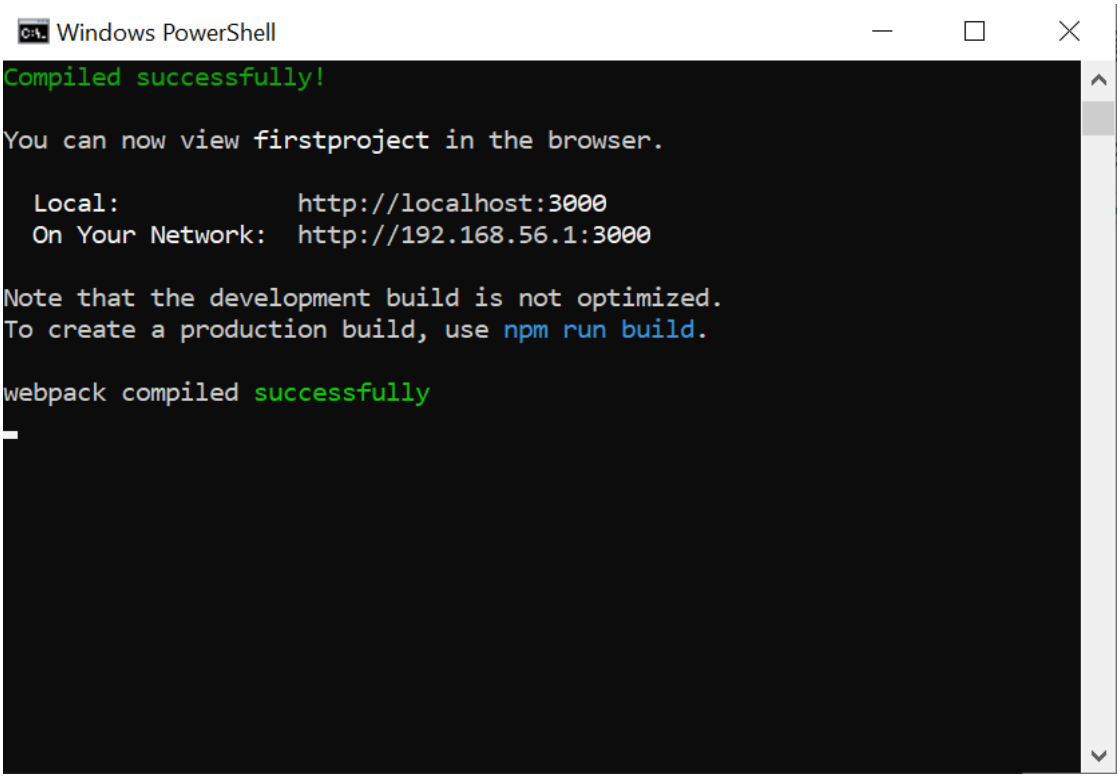
c. **Let's see how to run a react application:**
   Write this command on the terminal:

npm start

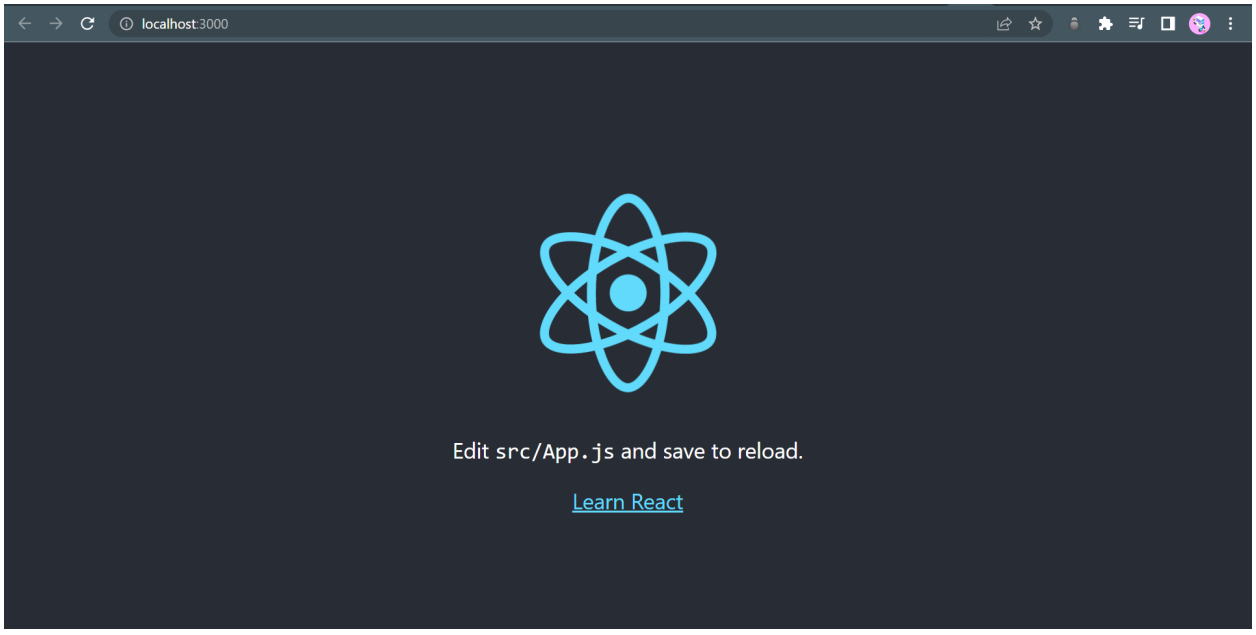
(Before running the above command you must change your directory to your project directory as shown in the image)


Command Prompt
npm run build
Bundles the app into static files for production.
npm test
Starts the test runner.
npm run eject
Removes this tool and copies build dependencies, configuration files
and scripts into the app directory. If you do this, you can't go back
!
We suggest that you begin by typing:
cd firstproject
npm start
Happy hacking!
C:\Users\hp\Desktop\newproject>cd firstproject
C:\Users\hp\Desktop\newproject\firstproject>


Command Prompt
npm run build
Bundles the app into static files for production.
npm test
Starts the test runner.
npm run eject
Removes this tool and copies build dependencies, configuration files
and scripts into the app directory. If you do this, you can't go back
!
We suggest that you begin by typing:
cd firstproject
npm start
Happy hacking!
C:\Users\hp\Desktop\newproject>cd firstproject
C:\Users\hp\Desktop\newproject\firstproject>npm start



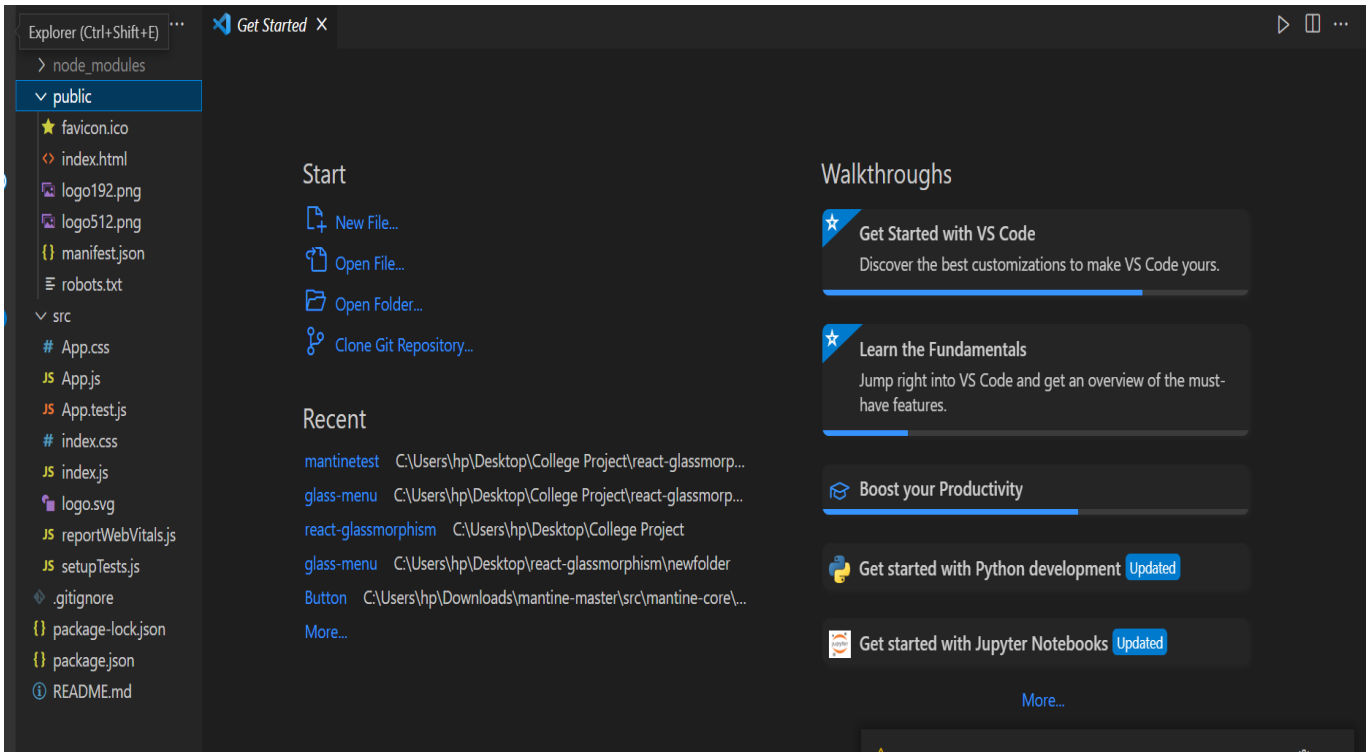
Server will start at 3000 port and default page will show:



6. Default Structure of React project Folder:

When you look at the project structure, you'll notice a /public and /src directory, as well as the standard node modules,.gitignore, README.md, and package.json files.

Our important file in /public is index.html, which is quite identical to the static index.html file we created before — just a root div.
No libraries or scripts are being loaded at this time. All of our React code will be stored in the /src directory.
Find the following line in /src/App.js to show how the environment automatically compiles and updates your React code:

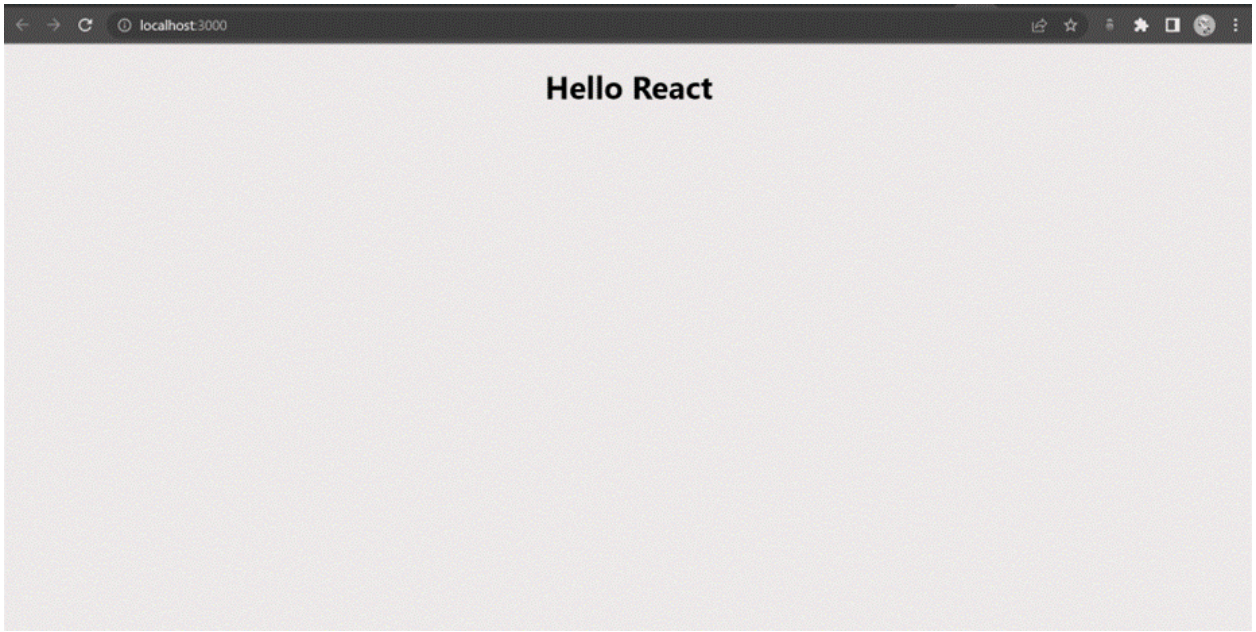


7. Let's write our first program in react:

App.js:

```
JS App.js
src > JS App.js > App
1   import "./App.css";
2
3   function App() {
4     return (
5       <div className="App">
6         <h1>Hello React</h1>
7       </div>
8     );
9   }
10
11  export default App;
12
```

In the result of the App function that was returned above, take note of how we wrote what appears to be HTML.
This is not HTML nor JavaScript, nor is it React. It's called JSX. It's a JavaScript extension that allows us to write function calls in HTML-like syntax.
Save this and check your browser; it should look like this -



Understand ReactJS library and directory

Topics covered:

- What is directory and Library?
- What is File Structure?
- What are modules in React?
  - ❖ Node module
  - ❖ exporting and importing modules

1. **Directories and library:**
   - Directories are the collection of files, subdirectories or both.
     Represented like:

     ```
     PS C:\Users\hp\Desktop\newproject>
     ```

     To change the directories we use commands:
     cd folder_name

     ```
     PS C:\Users\hp\Desktop\newproject> cd .\firstproject\
     PS C:\Users\hp\Desktop\newproject\firstproject>
     ```

     To move back to the previous directory:
     cd..

     ```
     PS C:\Users\hp\Desktop\newproject\firstproject> cd..
     PS C:\Users\hp\Desktop\newproject>
     ```

     To add a new directory:
     mkdir directory_name

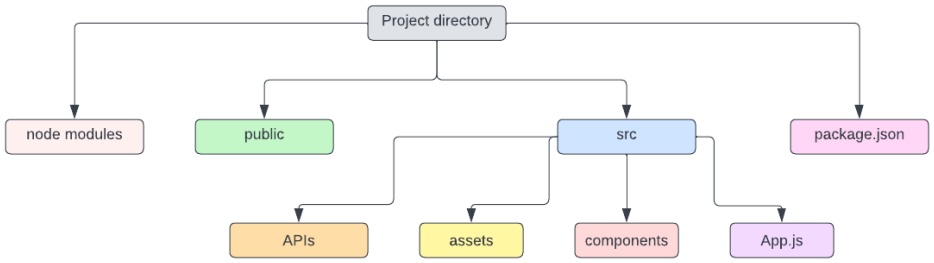
     ```
     PS C:\Users\hp\Desktop\newproject> mkdir directory1
     PS C:\Users\hp\Desktop\newproject> cd .\directory1\
     PS C:\Users\hp\Desktop\newproject\directory1>
     ```

     (note: Whenever you run a react app you must be in the right directory before running the command “npm start”).

   - Library is a combination of multiple directories and has multiple directories.

2. **File structure:**
   When you look at the project structure, you'll notice a /public and /src directory, as well as the standard node modules,.gitignore, README.md, and package.json files.
   Our important file in /public is index.html, which is quite identical to the static index.html file we created before — just a root div.
   No libraries or scripts are being loaded at this time. All of our React code will be stored in the /src directory.
   Find the following line in /src/App.js to show how the environment automatically compiles and updates your React code:

   - **Classification based on features or routes**
     One common method for organizing projects is to group CSS, JS, and tests into folders organized by feature or route. For example:



   - **File type identification**
     Another frequent method for organizing tasks is to put related files together.
   - **Avoid excessive nesting**
     There are numerous drawbacks to deep directory nesting in JavaScript programs. When the files are moved, it becomes more difficult to write relative imports between them or to update those imports.

3. **Modules:**
   - JavaScript modules allow you to split your code into individual files.
   - This simplifies code maintenance.
   - The import and export statements are used by ES Modules.

❖ **Node modules:**

➔ **Package:** A file or directory that is represented by a package.json file is referred to as a package. A package cannot be published to the npm registry without a package.json file.

➔ **Node module:** Node module is the online directory that contains the various already registered open-source packages.
NPM modules consume the various functions as a third-party package when installed into an app using the NPM command npm install.

❖ **Exporting:**

➔ Type of Export:
- Default export
- Named export

➔ **Default export**: Every module is said to have a maximum of one default export, as we already know. We must adhere to the syntax outlined below in order to export the default export from a file:

```
export default App;
```

➔ **Named export:** A named parameter can be exported using the following syntax. Each module may have several named parameters.

```
export { Classes };
```

❖ **Importing:**

➔ Type of Import:
- Import default export
- Import named export

➔ **Import default export:** Each module is said to have a single default export at most. It is possible to import the default export from a file by using merely the address and the term import before it, or by naming the import and using the syntax shown below:

import NAME from ADDRESS

Example;

```
import React from "react";
```

➔ **Import named export:** Every module has a number of named arguments, and we should use the following syntax to import one of them:

import { NAME } from ADDRESS

Example;

```
import { Classes } from "./components/Classes";
```

Introduction to JSX

Topics covered:

- What is JSX?
- What is react.createElement?
- How to Return JSX?
- How to add comments in JSX?
- How to add Javascript in JSX?
- How to add conditions in JSX?
- How to add classes in JSX?
- How to add CSS in JSX?
- Mini project

1. JSX:

- JSX is an acronym for Javascript XML (eXtensible Markup Language).
- JSX (JavaScript Extension Syntax) makes it easy to combine JavaScript and HTML in React.
- If we use html code in javascript it will look like this:

```
const JSX = <h1>Hey, Testbook Users</h1>
```

- This is basic React JSX code. This JSX, however, is not understood by the browser because the JavaScript code is invalid. This is due to the fact that we are assigning an HTML tag to a variable that is simply HTML code and not a string.
- Therefore, we need a tool like Babel, a JavaScript compiler/transpiler, to translate it into browser-friendly JavaScript code.
- You can also try create-react-app, which internally utilizes Babel to convert JSX to JavaScript.
- This is how our React code may use the JSX from above example:

```
export default class demo extends Component {
  render() {
    return <h1>Hey, TestBook UsersX</h1>;
    }
}
```

2. React Without JSX:

- Using *React.createElement()*

```
export class demo1 extends Component {
  render() {
      return React.createElement("h1", null, "Hey, Testbook Users");
  }
}
```

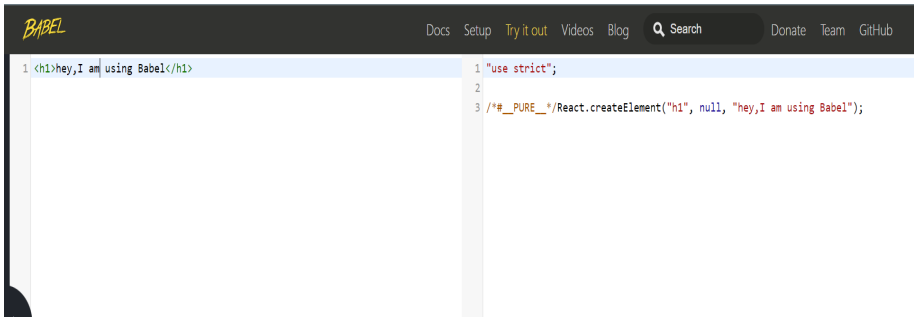
- This was the old approach of creating code in React - however writing the React is tedious. Every time, even when adding a basic div, use createElement.
- As a result, React released the JSX programming, which makes code easier to write and understand.

Let's talk about React.createElement() function:

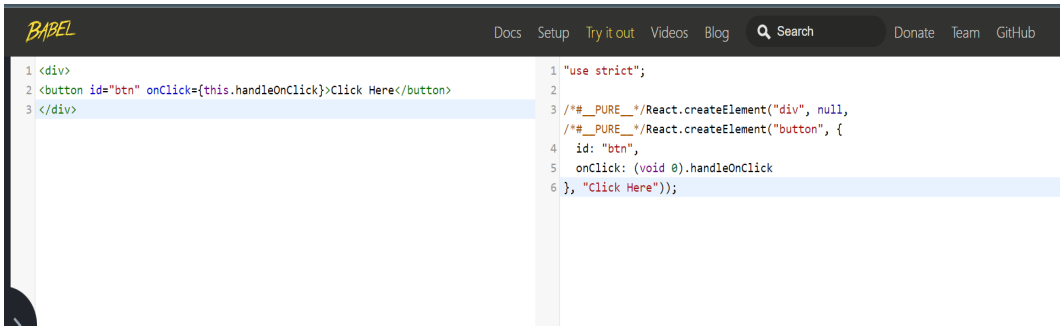
- Each JSX file is translated to React. A browser-friendly createElement function call.
- The syntax for React.createElement is:

```
React.createElement(type, [props], [...children]);
```

- Examine the parameters of the createElement function.
  - ➢ **Type** can be an HTML tag such as h1, div, or a React component.
  - ➢ **Props** are the characteristics that you want the element to have.
  - ➢ **Children** might be other HTML tags or a component.
- Let's see how babel convert JSX in createElement():



- Now, let's make the JSX more complex to see how it's transformed to the React.createElement method.

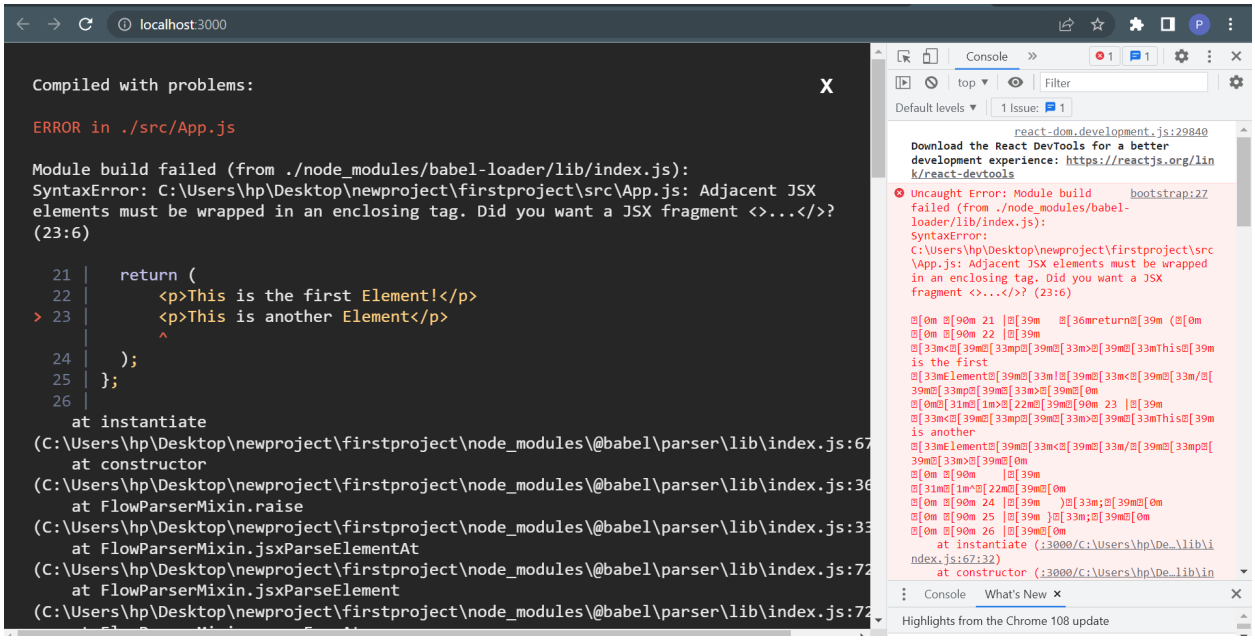


3. How to Return Complex JSX?

Examine the following code:

```
import ReactDOM from "react-dom";


const App = () => {
  return (
    <p>This is the first Element!</p>
    <p>This is another Element</p>
  );
};


const rootElement = document.getElementById("root");
ReactDOM.render(<App />, rootElement);
```

We're returning two paragraphs from the App component here. However, when you run the code, you will see the following error:



We're getting an error because React needs neighboring elements to be wrapped in a parent tag. We're getting an error since react only returns one element. As a result, a parent tag is required to enclose these nearby items.

```
2   import ReactDOM from "react-dom";
3
4   const App = () => {
5     return (
6       <div>
7         <p>This is the first Element!</p>
8         <p>This is another Element</p>
9       </div>
10    );
11  };
12
13  const rootElement = document.getElementById("root");
14  ReactDOM.render(<App />, rootElement);
15
```

Additionally, you can use the React.Fragment component to solve the problem:

src > components > JS DemoJSX.js > ...

```
2   import React from "react";
3
4   const App = () => {
5     return (
6       <React.Fragment>
7         <p>This is the first Element!</p>
8         <p>This is another Element</p>
9       </React.Fragment>
10    );
11  };
12
13  const rootElement = document.getElementById("root");
14  ReactDOM.render(<App />, rootElement);
15
```

- In React version 16.2, React.Fragment was introduced because we always have to wrap multiple adjacent elements in some tag (like div) inside every JSX returned by a component. As a result, unnecessary div tags are added.
- Most of the time, this is fine, but there are some exceptions.
- Flexbox, for example, has a special parent-child relationship in its structure. And adding divs in the middle makes it difficult to achieve the desired layout.
- Using ReactFragment solves this problem:
  - Fragments allow you to group a list of children without introducing new nodes to the DOM.

4. Comments in JSX:

If you have a line of code that looks like this:

```
<p>This is some text</p>
```

If you want to add a comment to that code, you must wrap it in JSX expression syntax inside the /* and */ comment symbols, as seen below:

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

5. Use JavaScript in JSX:
We have just used HTML tags as part of JSX up to this point. However, JSX becomes more useful when we include JavaScript code within it.

To include JavaScript code within JSX, use curly brackets like this:

```
const App = () => {
  const name = "Rohit";
  return (
    <div>
      <p>Name: {name}</p>
    </div>
  );
};
```

We can only write an expression inside curly brackets that evaluates to some value**.**
Therefore, JSX Expression Syntax is a common name for this use of curly brackets in syntax.

The following elements are valid in a JSX expression:

- A string similar to "hello"
- A number such as ten
- An array of the form [1, 2, 4, 5]
- An object property that gives a value.
- A function call that returns a value that could be anything. JSX
- A map method that returns a distinct array every time.
- Also JSX

  Arrays can be written in JSX Expressions because [{1, 2, 3, 4}] is finally converted to {1}{2}{3}{4} when rendered (which can be rendered without any issue).

The following are invalid and cannot be used in a JSX Expression:

- The following are invalid and cannot be used in a JSX Expression:
- A for loop, a while loop, or any other type of loop
- A declaration of variables
- A declaration of a function
- An if condition
- An object

Also, when used within JSX, undefined, null, and boolean are not displayed on the UI.

So, if you have a boolean value that you wish to display on the UI, you must wrap it in ES6 template literal syntax, as shown below:

```
const App = () => {

  const isAdmin = true;

  return (

    <div>

      <p>isAdmin is {`${isAdmin}`} </p>

    </div>

  );

};
```

6. Conditional Operators in JSX Expressions:

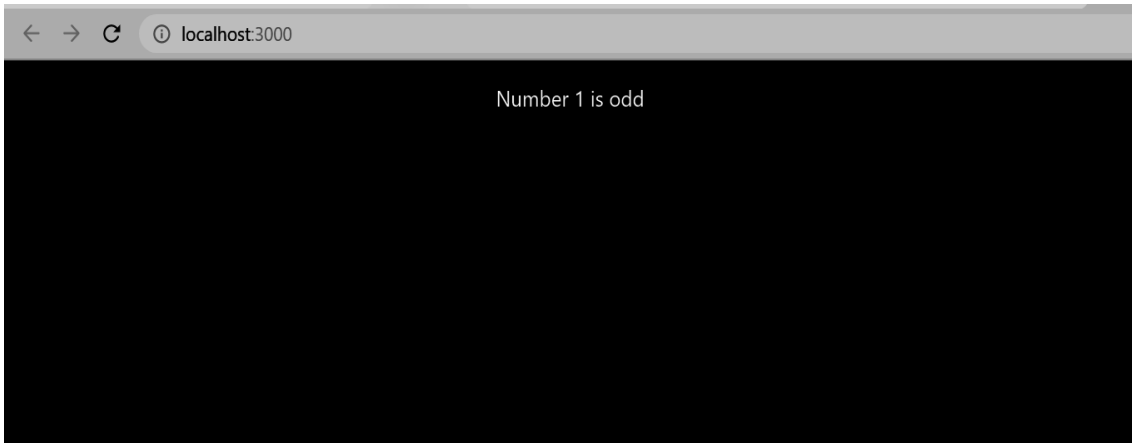
We can't write if conditions in JSX expressions, which may seem like a limitation. However, React allows us to build conditional operators, such as ternary operators, as well as the logical short circuit && operator, as shown below:

```
<p>{a > b ? "Greater" : "Smaller"}</p>
<p>{shouldShow && "Shown"}</p>
```

Example of conditional rendering:

```
const EvenOdd = () => {
  const number = 1;
  return (
    <div>
      {number % 2 == 0 ? (
        <p>Number {number} is even</p>
      ) : (
        <p>Number {number} is odd</p>
      )}
    </div>
  );
};
export default EvenOdd;
```

Output:



Example of trying every expression:

```
import React from "react";
import ReactDOM from "react-dom";

const greet = () => {
  return <p>Hello</p>;
```

```
};

const Expression = () => {
  const number = 10;
  const string = "Hello Testbook Users";
  const array = [25, 35, 45];
  const object = { name: "Rohan" };
  const noValue = undefined;
  const nullValue = null;
  const booleanValue = false;
  const a = 20;
  const b = 30;
  const shouldShow = true;
  const isFalse = false;
  return (
    <div>
      <p>Number: {number}</p>
      <p>String: {string} </p>
      <p>String Method: {string.toUpperCase()} </p>
      <p>Array: {array}</p>
      <p>Map: {array.map((value) => value * 2)}</p>
      <p>Name: {object.name} </p>
      <p>Function Call: {greet()}</p>
      <p>NoValue: {noValue} </p> {/* This will not be displayed */}
      <p>NullValue: {nullValue} </p>
      <p>BooleanValue: {booleanValue}</p>
      <p>{a > b ? "Greater" : "Smaller"}</p>
      <p>{shouldShow && "Shown"}</p>
      <p>{isFalse && "Won't Displayed"}</p>
      <p>{true && <p>This is nested inside JSX</p>}</p>
      <p>{<h3>The value of number is: {number}</h3>}</p>
    </div>
  );
};

export default Expression;
```

Output:

Number: 10

String: Hello Testbook Users

String Method: HELLO TESTBOOK USERS

Array: 253545

Map: 507090

Name: Rohan

Function Call:

Hello

NoValue:

NullValue:

BooleanValue:

Smaller

Shown

This is nested inside jsx

The value of number is: 10

7. How to add class in JSX:

As in HTML, we can add properties to JSX elements, such as id and class.
It is important to note that with React, we must use className instead of class.
Example:

```
import React from "react";

export const Classes = () => {
  const id = "id-1";
  return (
    <div>
      <h1 id={id}>This is heading 1</h1>
      <h2 className="active">This is heading 2</h2>
    </div>
  );
```

```
};
```

8. Styling in React with CSS:
There are various ways to style React using CSS, mainly 3 ways with CSS:

- Inline styling
- CSS stylesheets
- CSS Modules

1. **Inline Styling:**

   The value of the inline style attribute must be a JavaScript object in order to style an element:

Example:

```
const Header = () => {
  return (
    <>
      <h1 style={{ color: "red" }}>Hello Style!</h1>
      <p>Add a little style!</p>
    </>
  );
};
```

In JSX, JavaScript expressions are enclosed by curly braces, and because JavaScript objects are also enclosed by curly braces, the style in the above example is enclosed by two sets of curly braces.

Due to the fact that the inline CSS is written in a JavaScript object, values with hyphen separators, such as background-color, must be written in camel case:

```
const Header = () => {
  return (
    <>
```

```
      <h1 style={{ backgroundColor: "lightblue" }}>Hello Style!</h1>
      <p>Add a little style!</p>
    </>
  );
};
```

JavaScript Object (Internal Css):

You can also build a styled object and refer to it in the style attribute:

Example:

```
const Header = () => {
  const myStyle = {
    color: "white",
    backgroundColor: "DodgerBlue",
    padding: "10px",
    fontFamily: "Sans-Serif",
  };
  return (
    <>
      <h1 style={myStyle}>Hello Style!</h1>
      <p>Add a little style!</p>
    </>
  );
};
```

2. **Css Stylesheet:**

You can write your CSS styling in a separate file and import it into your application by saving it with the .css file extension.

Example: App.css

```
body {
  background-color: #282c34;
  color: white;
  padding: 40px;
  font-family: Sans-Serif;
  text-align: center;
}
```

Import the stylesheet in App.js

```
import React from "react";
import ReactDOM from "react-dom/client";
import "./App.css";

const App = () => {
  return (
    <>
      <h1>Hello Style!</h1>
      <p>Add a little style!.</p>
    </>
  );
};
```

3. **CSS Modules:**
    - CSS Modules are another option to add styles to your application.
    - CSS Modules are useful for components that are placed in different files.
    - The CSS inside a module is only available to the component that imported it, and there are no name conflicts.
    - Make a CSS module using the.module.css extension, such as my-style.module.css.

Create a CSS module with the .module.css extension, such as my-style.module.css.

Example: my-style.module.css

```
.bigblue {
  color: DodgerBlue;
  padding: 40px;
  font-family: Sans-Serif;
```

```
  text-align: center;
}
```

Import the stylesheet in your component:

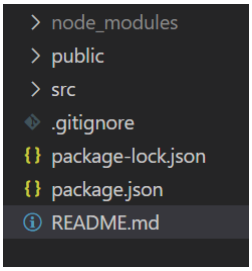
```
import styles from './my-style.module.css';

const Desktop = () => {
  return <h1 className={styles.bigblue}>Hello Car!</h1>;
}
export default Desktop;
```

Import the component in index.js:

```
import ReactDOM from "react-dom/client";
import Desktop from "./Desktop.js";

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

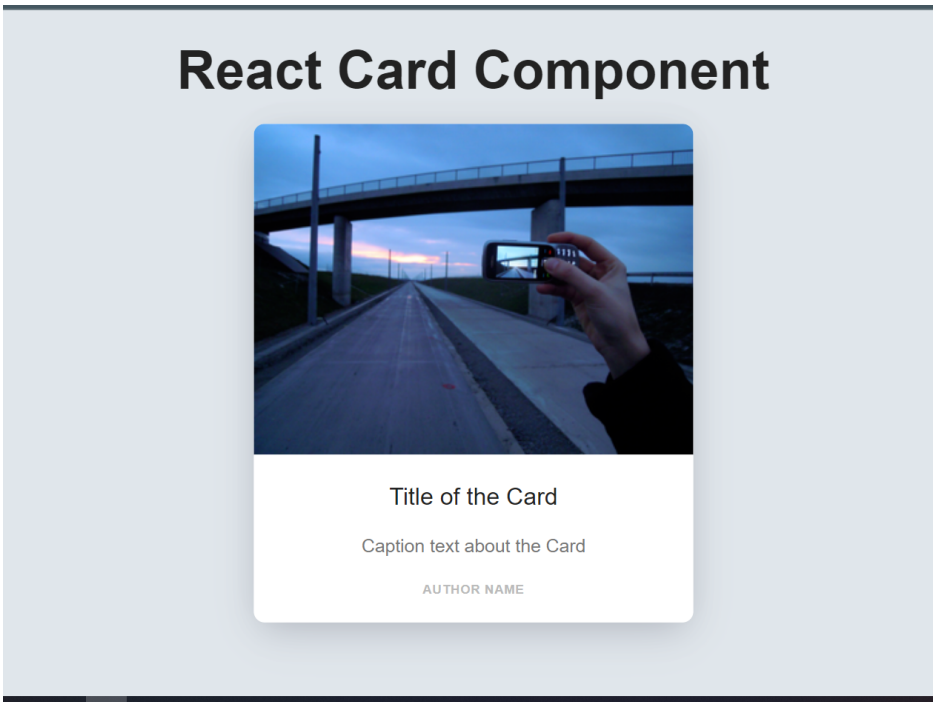
Assignment 1

Create a React App by using npx and you will see a file structure like this:



Open src
Create components directory

- Create a component named “Card.js”.
- Create an arrow function named “Card”.
- Create a .css file named “Card.css”.
- Add CSS by Applying External Css in Card.css.
- Add classNames as “cards” or “card-body”.
- Import it inside the App.js
- And render it inside the App function.
- Add hover property to the card.
  (on hover size of the card must increase or add transition of your choice).
  (Try to use camelCasing as It's a good Practice.)



Solution

React Component

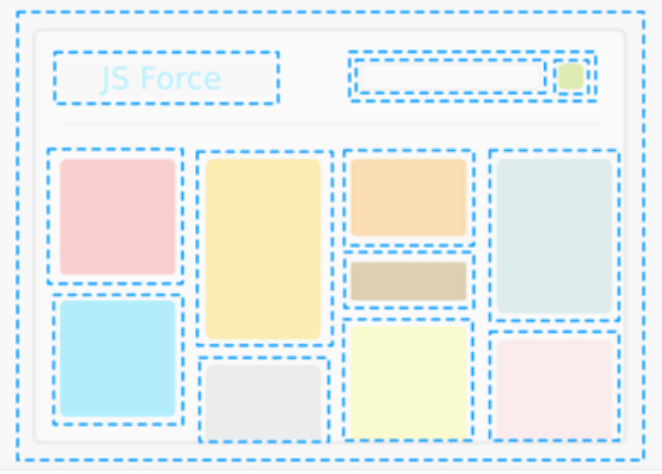
Topics covered:

- What are React components?
- Component based architecture
    - ❖ Types of react components
- Component rendering
    - ❖ Virtual Dom
    - ❖ Folder structure
    - ❖ rendering of component

1. React components:

- Components are one of React's fundamental building blocks. To put it another way, every application you create with React will be built up of what are known as components. Using components makes creating user interfaces considerably simpler. You can see a user interface (UI) divided into numerous separate parts, or components, and work on them independently before merging them all into a parent component to create your final UI.

Reusable components based ui



2. Component based architecture:

A component-based design divides a web page or web application into its smallest parts or units, allowing us to reuse it repeatedly without having to write the same code twice.

React offers two different kinds of components:

- ➔ Class-based components
- ➔ Function-based components

React components come with a built-in state object. When a component renders a component, persistent assets are stored in a state, which is encapsulated data.

When a user interacts with our application and changes its state, the UI may then appear entirely different. This is because the newly created state is used to represent that change, not the previous one. The process of initializing, updating, and altering a state's behavior in accordance with requirements is called State management.

- **Types of React components**
  - ❖ **Class-based components:**
    - ➔ A class is a particular kind of component in React that enables state management.
    - ➔ Until Hooks were introduced in React 16.8, the only way to use and maintain states was through a class-based component.
    - ➔ This is the reason why the only stateful component available at the time was a class.
    - ➔ Classes let us handle and alter the state in an efficient manner.

      A common class-based component contains the following:

      - ○ For developing JSX, a React component was imported from the React library.
      - ○ Classes can be created using this component imported from the React library.
      - ○ A class component that extends the React Component.
      - ○ To initialize the props, we need a constructor.
      - ○ To paint/create the UI, we need a render method.
      - ○ An export with the same name as the class.
      - ○ The class component in this sample is named Demo, and it creates a header called Hello Everyone!

➔ Example:

```
import React, { Component } from 'react'
export default class Demo extends Component {
    constructor(props){
          super(props)
           }
  render() {
    return (
      <div>
          <h1>Hello Everyone!</h1>
      </div>
    )
  }
}
```

❖ **Function-based component:**

- A functional component is like a JavaScript function.
- It begins with the term "function", then comes a name enclosed in parentheses and curly braces.
- Functional components did not offer state management prior to React Hooks' introduction in version 16.5 of React.
- Consequently, stateless components were referred to as functional components.
- With the creation of Hooks, we can essentially perform everything that a class component can do, but faster and with less lines of code.
  A common function-based component contains the following:

    - For developing JSX, a React component was loaded from the Reacting library.
    - A function declaration should begin with a capital letter, followed by the function name.
    - Between parentheses, a parameter is passed; in React, this parameter is called props.
    - A return method has JSX in it.
    - An export with the same name as the function.
    - The ES6 syntax can also be used to create functional components, such as arrow functions. The function keyword is removed, but everything else is the same.
    - There is only one container we can return. For instance, in this

case the div container is the one we are returning, and we are unable to return any tags outside of it.

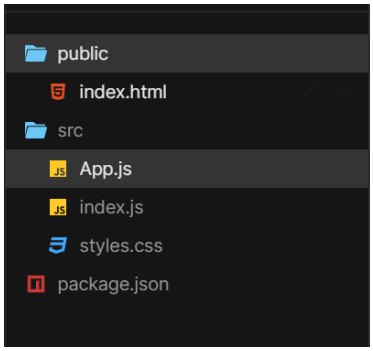
- Example:

```
import React from 'react'
export default function Demo() {
  return (
    <div>
        <h1>Hello Everyone!</h1>
    </div>
  )
}
```

3. Component rendering:

- **Virtual DOM:**
  - The virtual DOM (VDOM) is a programming concept in which a library like ReactDOM maintains an ideal, or "virtual," representation of a user interface in memory and keeps it in sync with the "actual" DOM. This is referred to as *reconciliation*.
  - One of the reasons React is quick is because it keeps a replica of the Real DOM called the *Virtual DOM.* This is merely a virtual version of the DOM. Real DOM manipulations are expensive.
  - React produces a new virtual DOM and compares it to the old one whenever we make changes or add data. *Diffing* is the name of the comparison procedure; the modifications are then batch-processed and the real DOM is updated with the fewest changes possible without repainting the entire DOM.
  - Reconciliation is the process of creating a virtual DOM, comparing changes to an earlier virtual DOM (diffing), and updating the browser DOM.

- **Folder structure:**
  In public folder, the main file is index.html which contains a div with the id is equal root which is updated by index.js



public/index.html/

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width,
initial-scale=1, shrink-to-fit=no">
  <meta name="theme-color" content="#000000">
  <link rel="manifest" href="%PUBLIC_URL%/manifest.json">
  <link rel="shortcut icon" href="%PUBLIC_URL%/favicon.ico">
  <title>React App</title>
</head>

<body>
  <noscript>
    You need to enable JavaScript to run this app.
  </noscript>
  <div id="root"></div>
</body>
</html>
```

Index.html contains:

```
<div id="root"></div>
```

Which is responsible for UI change.

index.js

```
import { StrictMode } from "react";
import { createRoot } from "react-dom/client";

import App from "./App";

const rootElement = document.getElementById("root");
const root = createRoot(rootElement);

root.render(
 <StrictMode>
   <App />
 </StrictMode>
);
```

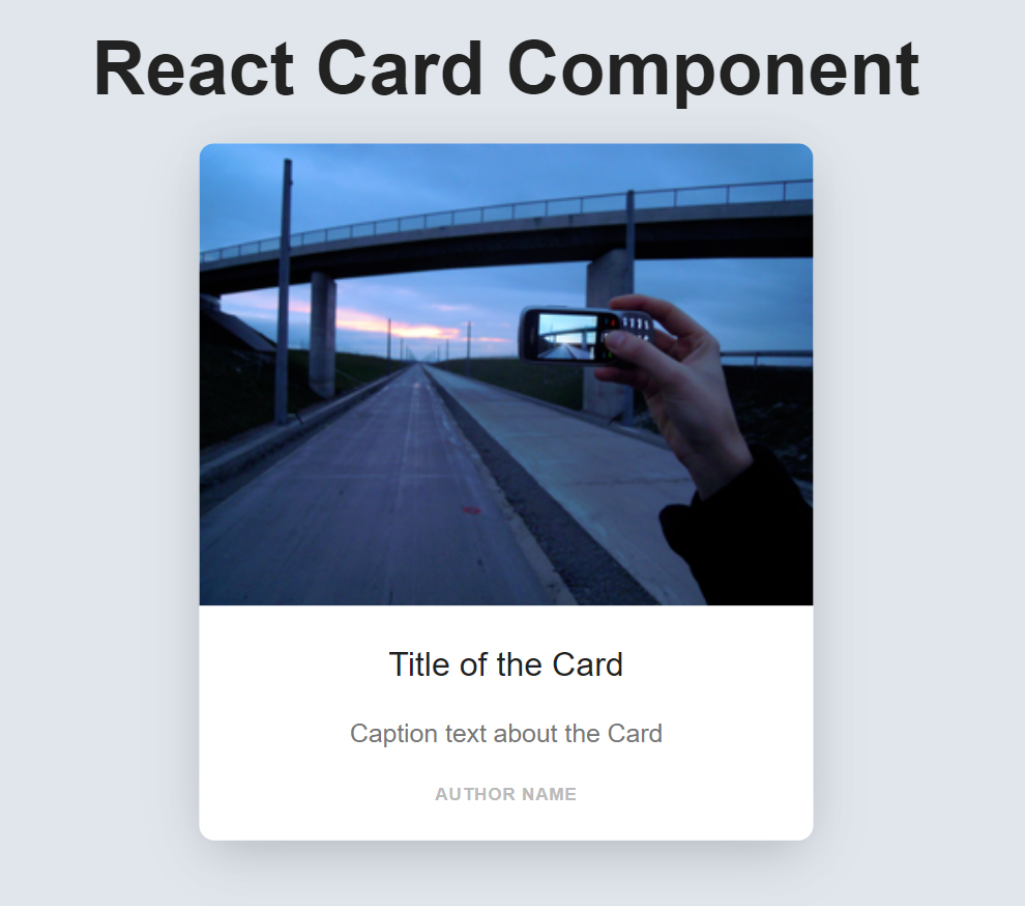
App.js is rendered here as a tag. First it is imported into the index.js and then it is rendered.

Let's make reusable components:
For this App.js

```
import "./styles.css";
export default function App() {
 return (
   <div className="App">
     <div className="header">
       <h1>React Card Component</h1>
     </div>
     <div className="cards">
       <Card />
     </div>
   </div>
 );
}
const Card = () => {
 return (
   <div className="card">
     <img

src="https://upload.wikimedia.org/wikipedia/commons/thumb/b/b6/Image_created_with_a_mobile_phone.png/330px-Image_created_with_a_mobile_phone.png"
       alt="img"
     />
     <div className="card-body">
```

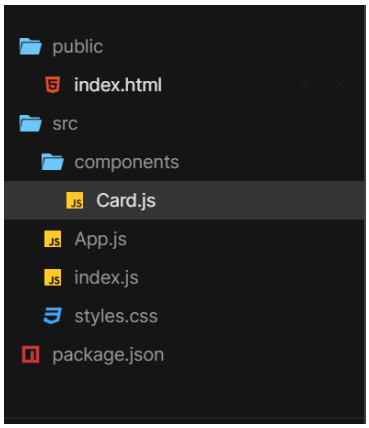
```
        <h2>Title of the Card</h2>
        <p>Caption text about the Card</p>
        <h5>Author name</h5>
      </div>
    </div>
  );
};
```

Whose output is :



Without changing the output we will modularize the code and change the structure of it.

We will create a folder called components and a file in it as Card.js



Inside the Card.js we will put the function which was there in App.js.

Card.js

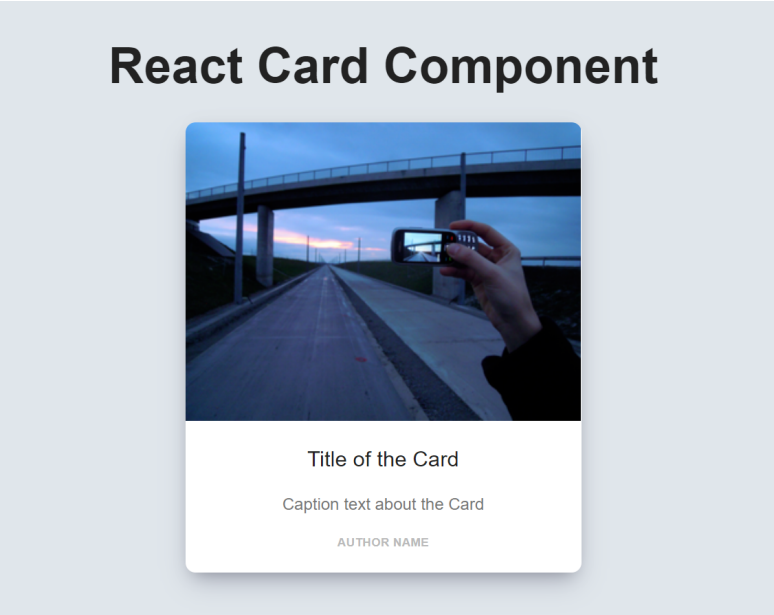
```
const Card = () => {
 return (
   <div className="card">
     <img
src="https://upload.wikimedia.org/wikipedia/commons/thumb/b/b6/Image_created_with_a_mobile_phone.png/330px-Image_created_with_a_mobile_phone.png"
       alt="img"
     />
     <div className="card-body">
       <h2>Title of the Card</h2>
       <p>Caption text about the Card</p>
       <h5>Author name</h5>
     </div>
   </div>
 );
};

export default Card;
```

And in App.js we have to import the Card component and render it as a tag.

App.js:

```
import "./styles.css";
import Card from "./components/Card";
export default function App() {
 return (
   <div className="App">
     <div className="header">
       <h1>React Card Component</h1>
     </div>
     <div className="cards">
       <Card />
     </div>
   </div>
 );
}
```

And here we are getting the same output after applying the appropriate styling to it.



Here

Class Based Components

Topics covered:

- What are Class Based components?
- What is a component constructor?
- What are props in class components?
    - Props to the constructors
    - Components in the component
- What is the State of class components?
    - Creating state object
    - Using State object
    - Changing state object

1. Class-Based Components:

- A class is a particular kind of component in React that enables state management.
- Until Hooks were introduced in React 16.8, the only way to use and maintain states was through a class-based component.
- This is the reason why the only stateful component available at the time was a class.
- Classes let us handle and alter the state in an efficient manner.

  A common class-based component contains the following:

    - For developing JSX, a React component was imported from the React library.
    - Classes can be created using this component imported from the React library.
    - A class component that extends the React Component.
    - To initialize the props, we need a constructor.
    - To paint/create the UI, we need a render method.
    - Export with the same name as the class.
    - The class component in this sample is named CodeFeast, and it creates a header called Hello Everyone!
    - Example:

```
import React, { Component } from 'react'
export default class Demo extends Component {
 render() {
   return (
     <div>Demo</div>
   )
 }
}
```

2. Component Constructor:

- When the component is started, the Constructor() method is the first function executed, making it the obvious place to set up the initial state and other basic settings.
- You should always use the super(props) method before any other method to start the parent's constructor procedure and allow the component to inherit methods from its parent.
- The Constructor() method is invoked with the props as arguments (React.Component).

Example:

```
import React from 'react';
export class Color extends React.Component {
 constructor() {
   super();
   this.state = { color: "red" };
 }
 render() {
   return <h2>The color is {this.state.color}</h2>;
 }
}
```

Output:

The color is red

3. Props:

- Props are similar to function arguments and are passed into the component as properties.
- Like in this Example, <Demo color =" red"/> added a color prop to the class and rendered it through this.props.color (this.props.propName)

Example:

```
class Demo extends React.Component {
 render() {
   return <h2>The color is {this.props.color} Car!</h2>;
 }
}
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Demo color="red"/>);
```

Props in constructors:

- Props should always be supplied to the constructor function of your component if it has one, as well as to the React.Component via the super() function.

Example:

```
class Demo extends React.Component {
 constructor(props) {
    super(props);
}
 render() {
   return <h2>The color is {this.props.color} Car!</h2>;
 }
}
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Demo color="red"/>);
```

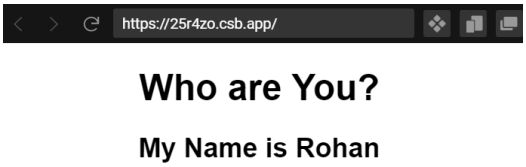
Components in components:

- Components inside other components are referred to as

Example:

```
class Demo extends React.Component {
 render() {
   return <h2>My Name is Rohan</h2>;
 }
}

class Demo1 extends React.Component {
 render() {
   return (
     <div>
     <h1>Who are You?</h1>
     <Demo />
     </div>
   );
 }
}
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Demo1/>);
```

Output:



4. State in class-based component:

- A state object is pre-built into React Class components.
- As you may have seen, the state was used in the component constructor part before.
- Property values for the component's properties are kept in the state object.
- The component re-renders whenever the state object alters.

Create a State object:

In the constructor, the state object is initialized:

```
export class Color extends React.Component {
 constructor(props) {
   super(props);
   this.state = { color: "red" };
 }
 render() {
   return <h2>The color is {this.state.color}</h2>;
 }
}
```

You can have as many attributes as you wish in the state object:

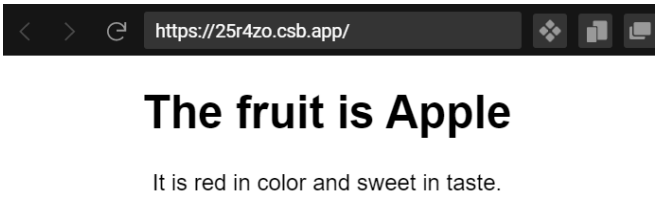
```
import React from "react";
export class Fruit extends React.Component {
 constructor(props) {
   super(props);
   this.state = {
           fruit: "apple",
           color: "red",
           taste: "sweet" };
 }
 render() {
   return <h2>The Fruit</h2>;
 }
}
```

Using State object:

Use the this.state.propertyName syntax to refer to the state object anywhere in the component:

```
import React from "react";
export class Fruit extends React.Component {
 constructor(props) {
   super(props);
   this.state = {
     fruit: "Apple",
     color: "red",
     taste: "sweet"
   };
 }
 render() {
   return (
     <div>
       <h1>The fruit is {this.state.fruit}</h1>
       <p>
         It is {this.state.color} in color and {this.state.taste}
 in taste.
       </p>
     </div>
   );
 }
}
```

Output:



Changing the state object:

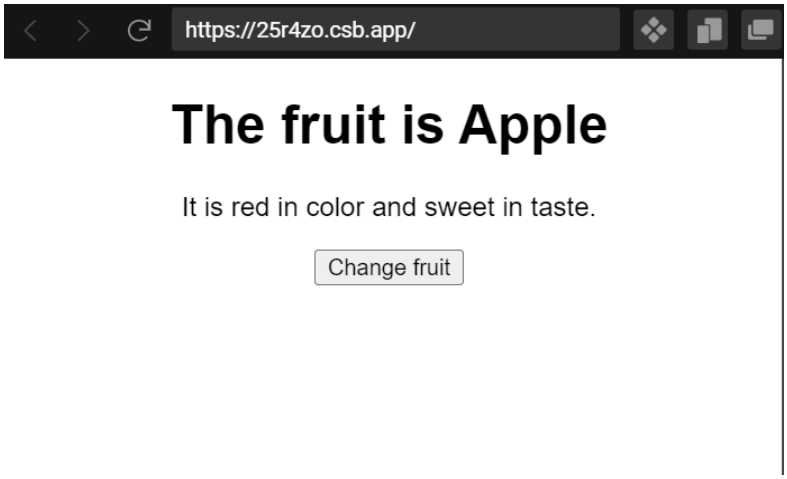
- Use this.setState() to modify a value in the state object.
- The component will re-render when a value in the state object changes, which means that the output will adjust to the new value (s).

Example:

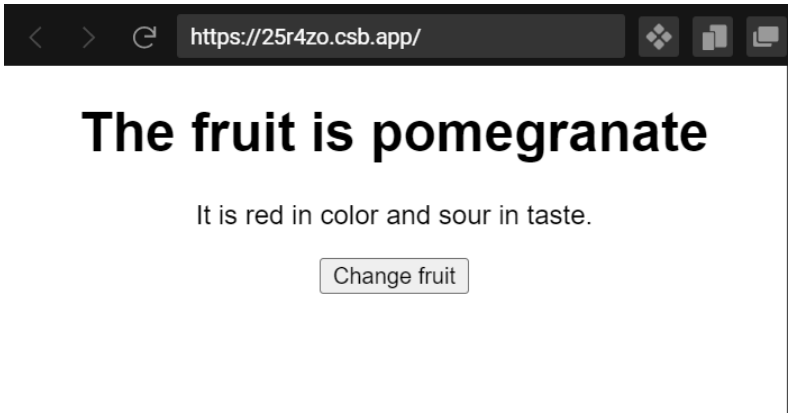
```
import React from "react";
export class Fruit extends React.Component {
 constructor(props) {
   super(props);
   this.state = {
     fruit: "Apple",
     color: "red",
     taste: "sweet"
   };
 }
 changeFruit = () => {
   this.setState({ fruit: "pomegranate", taste: "sour" });
 };
 render() {
   return (
     <div>
       <h1>The fruit is {this.state.fruit}</h1>
       <p>
         It is {this.state.color} in color and {this.state.taste} in
taste.
       </p>
       <button onClick={this.changeFruit}>Change fruit</button>
     </div>
   );
 }
}
```

Output:

Initially before clicking on the Change fruit button.



After clicking on the Change fruit button:



Functional-Based Components

Topics covered:

- What are functional components?
  - Arrow functional components
  - Stateless functional components
- How functional components are different from class components?
- How to use functional components?
  - Syntax of functional components
  - Passing props to functional components
  - State in functional components

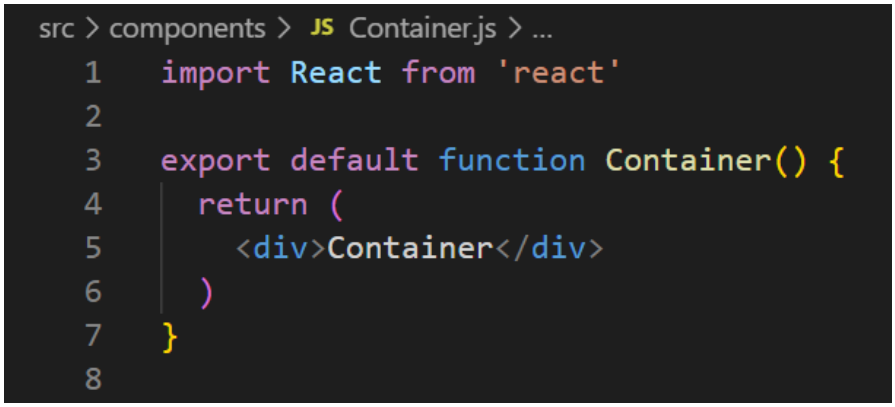
1. **Functional components:**
   A functional component is like a JavaScript function.
   It begins with the term "function", then comes a name enclosed in parenthesis and curly braces.
   For Example

```
Function FunctionName(){
        Return(
            //JSX
            <React.Fragment>
                <h1>Hello React Users</h1>
            </React.Fragment>
        )
    }
```

Functional components did not offer state management before React Hooks' introduction in version 16.5 of React.
Consequently, ***stateless components*** were referred to as functional components.
With the creation of Hooks, we can essentially perform everything that a class component can do, but faster and with fewer lines of code.
A common function-based component contains the following:

- For developing JSX, a React component was loaded from the Reacting library.
- A function declaration should begin with a capital letter, followed by the function name.
- Between parentheses, a parameter is passed; in React, this parameter is called props.
- A return method has JSX in it.
- Export with the same name as the function.
- The ES 6 syntax can also be used to create functional components, such as arrow functions. The function keyword is removed, but everything else is the same.

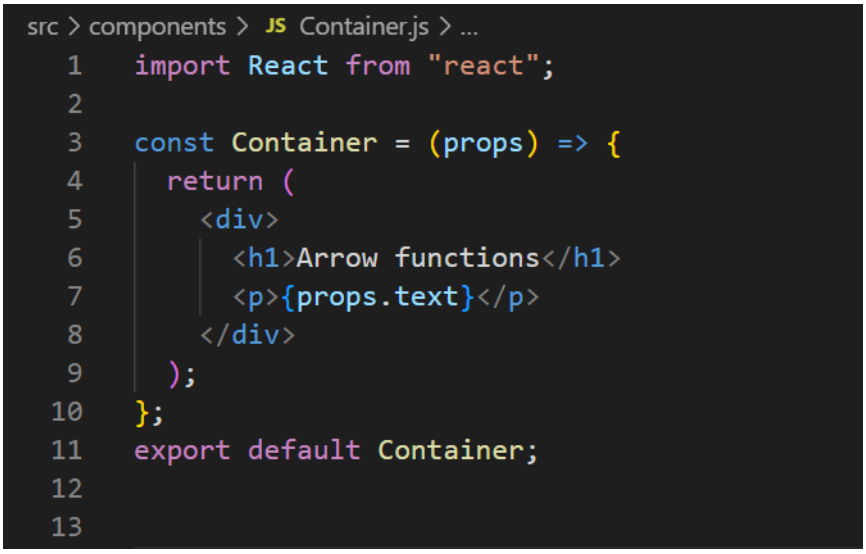
- There is only one container we can return. For instance, in this case, the div container is the one we are returning, and we are unable to return any tags outside of it.
  Example:



Arrow Functional components:

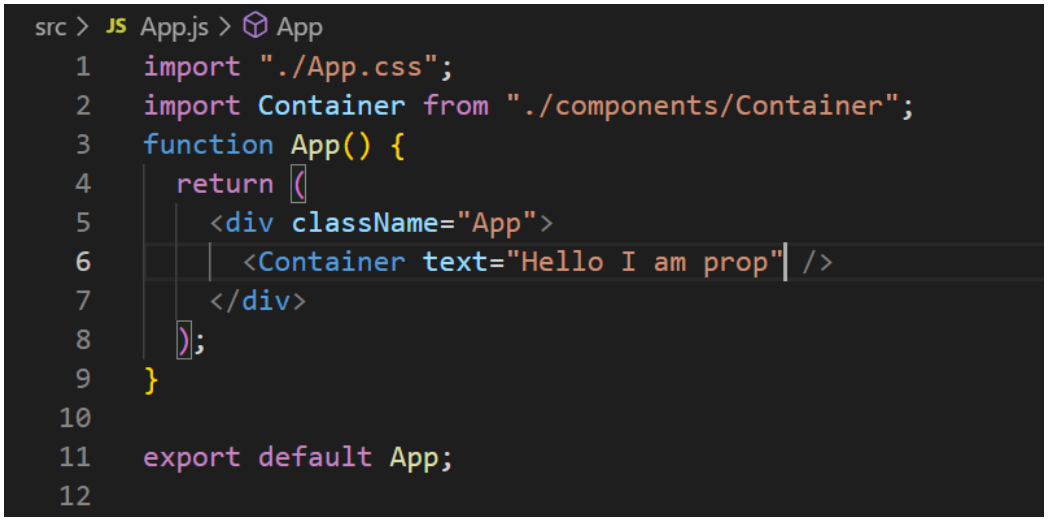
- Over components with particular function keywords, arrow functions are the preferred method for most developers when creating functional components.
- The arrow function makes it easier to design components, and the code is clean and easy to understand.
- It also has the advantage of dealing with this context dynamically, whereas arrow functions deal with it lexically.

Here is an example of a functional component for React that uses the Arrow function:



- Even so, nothing has changed concerning props, thus we can still send props to the arrow functions. We shall export this component at the end so that it can be used in future components.

- The procedure for importing the Container component and employing it to render the data is demonstrated below:



Stateless functional components:

- The purpose of building stateless functional components is to concentrate on the user interface. Stateless functional components lack state and lifecycle management.
- It is simple to create and comprehend by other developers. We also don't have to be concerned about this keyword.
- It enhanced overall performance because we no longer had to worry about managing the state.

2. Functional components vs class components:

- A functional component is a pure JavaScript function that takes props as an argument and returns a React element (JSX). You must extend from React to create a class component. Create a component and a render method that returns a React element.
- In functional components, no render method is used. It must have a render() method that returns JSX (which is syntactically similar to HTML)
- When the function is returned, functional components no longer exist and operate from top to bottom. The class component is created, and several life cycle methods are maintained alive and executed, and invoked depending on the phase of the class component.
- Functional components are sometimes known as stateless components because they only accept data and show it in some manner, and they are primarily responsible for UI rendering. On the other hand, class-based components use logic and state, they are also called stateful components.
- Functional components cannot use React lifecycle methods. Within class components, React lifecycle methods can be used.

- Hooks may easily be utilized to create functional components stateful.
  example: const [name, setName] = React.useState(‘ ‘)
- The syntax for hook implementation inside a class component must be distinct.
  example: constructor(props) {

  super(props);

  this.state = {name: ‘ ‘}

  }

- Constructors are not used in functional components. Constructors are used because the state must be stored in class-based components.

3. **How to use functional components:**
   a. **Syntax of functional component:**
   A functional component is a simple function that returns a valid React element, as previously stated.

```
1   import React from "react";
2   import "./App.css";
3
4   function App() {
5     return (
6       <div className="App">
7         <h1>My First Functional Component</h1>
8       </div>
9     );
10  }
11
12  export default App;
13
```

   This is a very fundamental functional part that displays static text on the screen.
   Note: The ES6 arrow function can also be used to create functional components.

   b. **Passing props in functional components:**
   React's functional components are pure javascript functions. It accepts an object called props (which stands for properties) as an argument and produces JSX.

   Let's break this down with some code.

   Inside src, create a folder called components, i.e. /src/components. This is merely a practice that all react developers use; we will write all of our components in this folder and then import them into App.js.

Let's make a file called Person.js in the components folder:

src > components > JS Person.js > ...

```
1   import React from "react";
2   import "./Person.css";
3
4   const Person = (props) => {
5     return (
6       <div className="person">
7         <h2>Name: {props.name}</h2>
8         <h2>Age: {props.age}</h2>
9       </div>
10    );
11  };
12
13  export default Person;
14
```

And App.js seems to be as follows:

src > JS App.js > default

```
1   import React from "react";
2   import "./App.css";
3   import Person from "./components/Person";
4
5   function App() {
6     return (
7       <div className="App">
8         <Person name="David" age={20} />
9       </div>
10    );
11  }
12
13  export default App;
```

c. State in functional components:

useState is a React hook for the state (explain with an example)

React Hooks are functions that allow function components to access React's state and lifecycle attributes. Hooks allow us to manipulate the state of our functional components without converting them to class components.

src > JS App.js > default

```
1   import React, { useState } from 'react'
2   import './App.css';
3
4   function App() {
5
6     const [counter, setCounter] = useState(0)
7
8     const clickHandler = () => {
9       const updatedCounter = counter + 1;
10      setCounter(updatedCounter);
11    }
12
13    return (
14      <div className="App">
15        <div>Counter Value: {counter}</div>
16        <button onClick={clickHandler}>
17          Increase Counter
18        </button>
19      </div>
20    );
21  }
22
23  export default App;
24
```

To clarify what we've said above, we're utilizing a state variable named counter, with an initial value of 0.
When we press the "Increase Counter" button, the current counter value is increased by one, and the current state is updated using the setCounter method.

When the state is altered or updated, the UI is refreshed or the component is re-rendered, we see the revised counter value on the screen.

Note that a state might be a text, number, array, object, or boolean, whereas props is an object.

Lifecycle methods in React

Topics covered:

- What is the life cycle of a component?
- Mounting
    - constructor()
    - getDrivedStateFromProps()
    - render()
    - componentDidMount()
- Update
    - getDerivedStateFromProps()
    - shouldComponentUpdate()
    - render()
    - getSnapshotBeforeUpdate()
    - componentDidUpdate()
- Unmounting
    - componentWillUnmount()

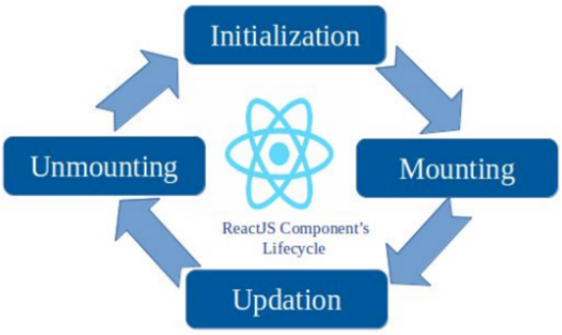
1. **Lifecycle of a component**:

The three major phases of each component's lifecycle in React can be used to monitor and modify each component.

It consists of three stages:

- Mounting (Inserting element in DOM)
- Updating (Modifying element in DOM)
- Unmounting (Removing element from DOM)

These represent the beginning, development, and ending of a component, respectively.



2. Mounting

When a React component is generated and added to the DOM, we can view it on the user interface or in the browser;

it goes through the following processes, listed here in the order they occur:

- Constructor.
- static getDerivedStateFromProps()
- render()
- componentDidMount()

constructor:

When the component is started, the Constructor() method is the first function that is executed, making it the obvious place to set up the initial state and other basic settings.

You should always use the super(props) method before any other method in order to start the parent's constructor procedure and allow the component to inherit methods from its parent. The Constructor() method is invoked with the props as arguments (React.Component).

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

class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = {favoritecolor: "red"};
  }
  render() {
    return (
      <h1>My Favorite Color is {this.state.favoritecolor}</h1>
    );
  }
}

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

getDerivedStateFromProps():

The second method that is called, both on the initial mount and future updates, is getDerivedStateFromProps().

- It is called just before invoking the render method.
- If the state needs to be updated, it must return an object; otherwise, it must return null.
- It is not frequently used and is only used when state derivation is necessary.

```
class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = {favoritecolor: "red"};
  }
  static getDerivedStateFromProps(props, state) {
    return {favoritecolor: props.favcol };
  }
  render() {
    return (
      <h1>My Favorite Color is {this.state.favoritecolor}</h1>
    );
  }
}
```

Output:

Render method

The third method is called render, and it is the only one that is absolutely necessary to render our class component.

It is called anytime there is a change in state or props, and it is largely used for printing the JSX in DOM.

Class Header extends Component{

render(

```
            //JSX
                <React.Fragment>
                    <h1>Hello JavaScript Users </h1>
                </React.Fragment>
        )
}
```

```
class Header extends React.Component {
  render() {
    return (
      <h1>This is the content of the Header component</h1>
    );
  }
}

ReactDOM.render(<Header />, document.getElementById('root'));
```

componentDidMount():

The final lifecycle method in the mounting stage, componentDidMount(), is called right away when a component is put into the DOM.

It is mostly used for making API calls and is only called once.

```
import React from "react";

class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = { favoritecolor: "red" };
  }
  componentDidMount() {
    setTimeout(() => {
      this.setState({ favoritecolor: "yellow" });
    }, 1000);
  }
  render() {
    return <h1>My Favorite Color is {this.state.favoritecolor}</h1>;
  }
}
export default Header;
```

This will be the output before 1000 milliseconds:

My Favorite Color is red

This will be the output after 1000 millisecond:

My Favorite Color is yellow

3. Updating:

The next stage in the lifecycle occurs when a component is updated.
Every time a component's state or props change, the component is updated.
When a component is changed, React's five built-in methods are called in the following order:

- static getDerivedStateFromProps()
- shouldComponentUpdate()

- render()
- getSnapshotBeforeUpdate()
- componentDidUpdate()

The ***render()*** method is essential and is always called; the others are optional and are only called if they are defined.

getDerivedStateFromProps:

- The getDerivedStateFromProps function is also invoked during updates. When a component is updated, this is the first method that is called.
- This is still the correct location to set the state object based on the initial props.
- Two procedures that are used frequently during the Mounting stage are static getDerivedStateFromProps and *render()*.

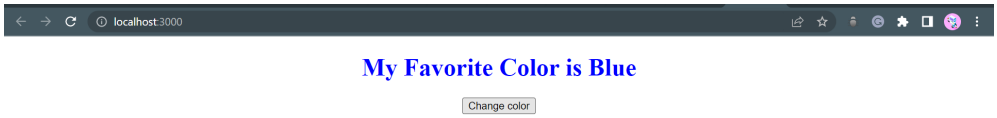
The button in the following example changes the favorite color to green, however because the getDerivedStateFromProps() function is invoked, which updates the state with the color from the favcol attribute, the favorite color remains blue:

src > components > JS Header.js > ...

```
1   import React from "react";
2   class Header extends React.Component {
3     constructor(props) {
4       super(props);
5       this.state = { favoritecolor: "red" };
6     }
7     static getDerivedStateFromProps(props, state) {
8       return { favoritecolor: props.favcol };
9     }
10    changeColor = () => {
11      this.setState({ favoritecolor: "Green" });
12    };
13    render() {
14      return (
15        <div>
16          <h1 style={{ color: this.state.favoritecolor }}>
17            My Favorite Color is {this.state.favoritecolor}
18          </h1>
19          <button type="button" onClick={this.changeColor}>
20            Change color
21          </button>
22        </div>
23      );
24    }
25  }
26
27  export default Header;
28
```

```
src > JS App.js > ...
1  import "./App.css";
2  import Header from "./components/Header";
3  function App() {
4    return (
5      <div className="App">
6      <Header favcol="Blue" />
7      </div>
8      );
9    }
10
11   export default App;
```

Output:



shouldComponentUpdate:

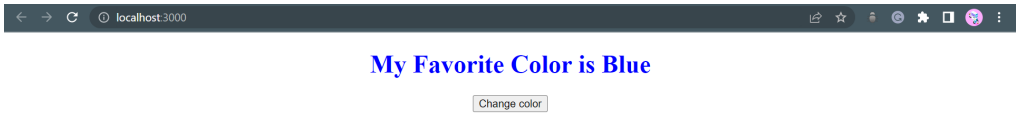
- The shouldComponentUpdate() method returns a Boolean value indicating whether React should continue with the rendering or not.
- True is the default value.
- *shouldComponentUpdate()* is used to inform React whether the current change in state or props has no impact on a component's output.
- Re-rendering occurs automatically by default whenever a state changes.
- Similar to *componentDidMount()*, *componentDidUpdate()* is the final method to be called during the update phase. It is only called once, and when it is, it indicates that the DOM has been changed.
- After the DOM has been updated, *ComponentDidUpdate()* is typically used to operate on DOM elements.
- The following example illustrates what transpires when the shouldComponentUpdate() function returns false:

src > components > JS Header.js > ...

```
1   import React from "react";
2   //shouldComponentUpdate
3   class Header extends React.Component {
4     constructor(props) {
5       super(props);
6       this.state = { favoritecolor: "Blue" };
7     }
8     shouldComponentUpdate() {
9       return false;
10    }
11    changeColor = () => {
12      this.setState({ favoritecolor: "Green" });
13    };
14    render() {
15      return (
16        <div>
17          <h1 style={{ color: this.state.favoritecolor }}>
18            My Favorite Color is {this.state.favoritecolor}
19          </h1>
20          <button type="button" onClick={this.changeColor}>
21            Change color
22          </button>
23        </div>
24      );
25    }
26  }
27
28  export default Header;
29
```

In this case, we send the value to shouldComponentUpdate, which returns false.

So, if we click the change color button in the output, there will be no change in the component because the value returned by shouldComponentUpdate is false.



If we return the true value to shouldComponentUpdate, the component will be updated when we click the change color button.

src > components > JS Header.js > ...

```
import React from "react";
//shouldComponentUpdate
class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = { favoritecolor: "Blue" };
  }
  shouldComponentUpdate() {
    return true;
  }
  changeColor = () => {
    this.setState({ favoritecolor: "Green" });
  };
  render() {
    return (
      <div>
        <h1 style={{ color: this.state.favoritecolor }}>
          My Favorite Color is {this.state.favoritecolor}
        </h1>
        <button type="button" onClick={this.changeColor}>
          Change color
        </button>
      </div>
    );
  }
}

export default Header;

```

Output:

render():

When a component is updated, it must re-render the HTML to the DOM with the new changes, which calls the render() method.

The button in the example below switches the favorite color to Green:

src > components > JS Header.js > ...

```
import React from "react";

class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = { favoritecolor: "Blue" };
  }
  changeColor = () => {
    this.setState({ favoritecolor: "Green" });
  };
  render() {
    return (
      <div>
        <h1 style={{ color: this.state.favoritecolor }}>
          My Favorite Color is {this.state.favoritecolor}
        </h1>
        <button type="button" onClick={this.changeColor}>
          Change color
        </button>
      </div>
    );
  }
}

export default Header;

```

Output:

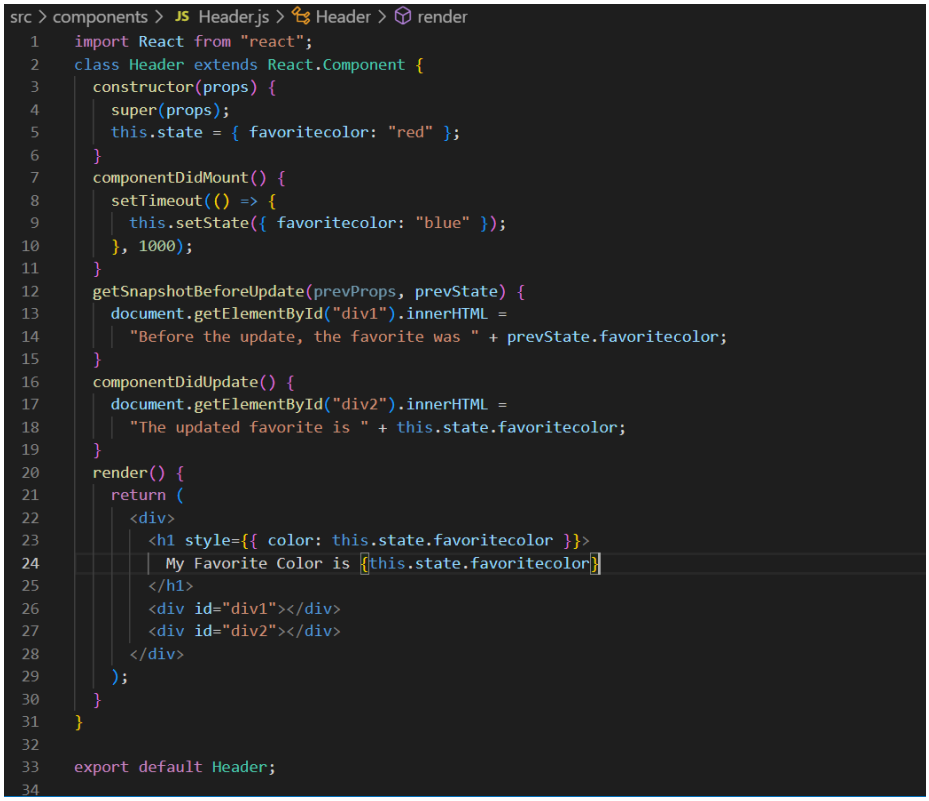
getSnapshotBeforeUpdate():

- The getSnapshotBeforeUpdate() method gives you access to the props and state from before the update, so you can examine the values from before the update even after it has occurred.
- The componentDidUpdate() function must be present in addition to the getSnapshotBeforeUpdate() method in order to avoid an error.

Although the example below appears complex, all it accomplishes is the following:

- The preferred color "red" is displayed when the component is mounting.
- After mounting the component, a timer modifies the state, and after one second, the preferred color switches to "yellow."
- Since this component includes a getSnapshotBeforeUpdate() method, it is called when the update phase is triggered. This method writes a message to the empty DIV1 element.
- The empty DIV2 element is then written with the following message by the componentDidUpdate() method:

To see what the state object looked like before the change, use the getSnapshotBeforeUpdate() method:



Output:

My Favorite Color is red

My Favorite Color is blue

Before the update, the favorite was red
The updated favorite is blue

componentDidUpdate():

- After the component has been updated in the DOM, the componentDidUpdate function is called.
- The following example may appear difficult, however it only does the following:
- When the component is mounted, the color "red" is used to represent it.
- A timer changes the state and the color to "blue" after the component is mounted.
- This action initiates the update phase, and because this component includes a componentDidUpdate function, this method is invoked and a message is written to the empty DIV element:

src > components > JS Header.js > ...

```
import React from "react";
class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = { favoritecolor: "red" };
  }
  componentDidMount() {
    setTimeout(() => {
      this.setState({ favoritecolor: "blue" });
    }, 1000);
  }
  componentDidUpdate() {
    document.getElementById("mydiv").innerHTML =
      "The updated favorite is " + this.state.favoritecolor;
  }
  render() {
    return (
      <div>
        <h1 style={{ color: this.state.favoritecolor }}>
          My Favorite Color is {this.state.favoritecolor}
        </h1>
        <div id="mydiv"></div>
      </div>
    );
  }
}

export default Header;


```

Output:

My Favorite Color is red

My Favorite Color is blue

The updated favorite is blue

4. **Unmounting:**

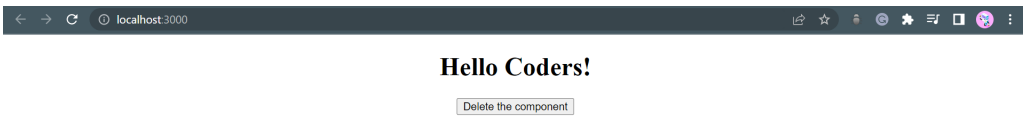
The final phase of a component's lifecycle, unmounting, marks the end of the component's existence in the DOM.

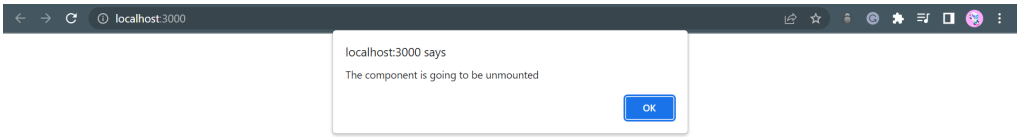
- componentWillUnmount() is the main method it contains.
- Before a component is unmounted and destroyed, componentWillUnmount() is called.
- It is mostly used to clean up our code, such as cancelling API calls or removing any subscriptions generated by componentDidMount ().
- Here is an example of unmounting:

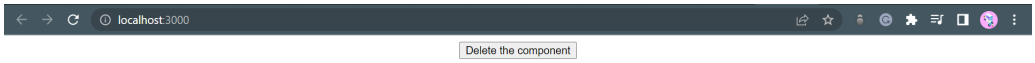
src > components > JS Container.js > ...

```
1   import React from "react";
2
3   class Container extends React.Component {
4     state = { display: true };
5     delete = () => {
6       this.setState({ display: false });
7     };
8     render() {
9       let comp;
10      if (this.state.display) {
11        comp = <Child />;
12      }
13      return (
14        <div className="App">
15          {comp}
16          <button onClick={this.delete}>Delete the component</button>
17        </div>
18      );
19    }
20  }
21  class Child extends React.Component {
22    // Defining the componentWillUnmount method
23    componentWillUnmount() {
24      alert("The component is going to be unmounted");
25    }
26    render() {
27      return <h1>Hello Coders!</h1>;
28    }
29  }
30
31  export default Container;
32
```

Here is the Output:




localhost:3000
localhost:3000 says
The component is going to be unmounted
OK


localhost:3000
Delete the component

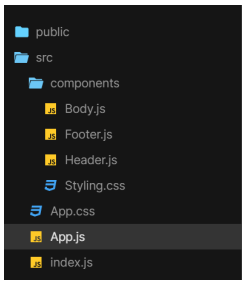
Build custom Component

Topics covered:

- Component 1
- Component 2
- Component 3

1. Component 1:

- The first component we will create is the header component.
    - In the header component we will create a navbar.
    - And in the navbar we will create a search bar and nav items like Home, About and Posts. (not responsive).



We create a folder named components and inside the components we create three (.js) files which are the custom components we can create.

Header.js:

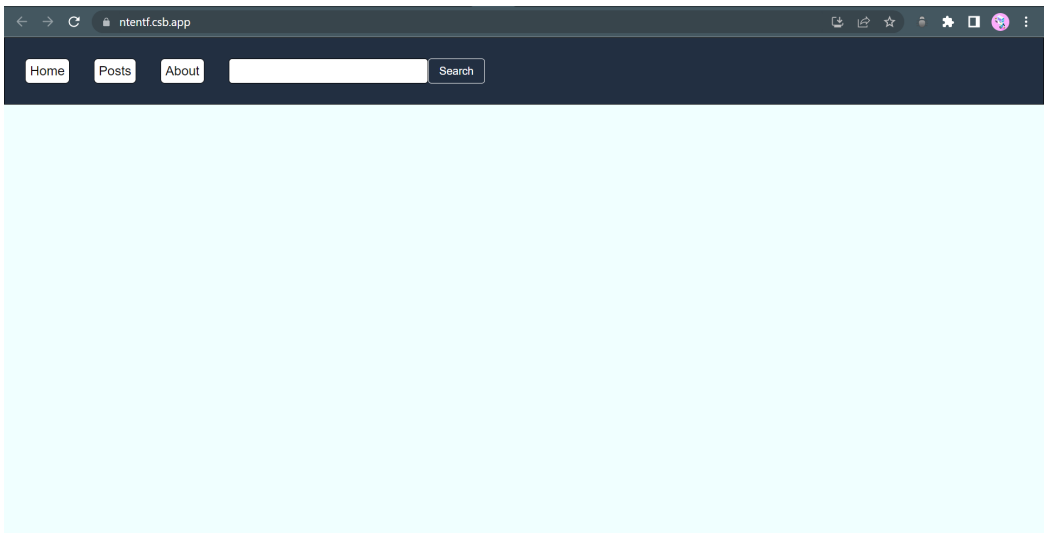
```
import React, { Component } from "react";
import "./Styling.css";
export default class Header extends Component {
  render() {
    return (
      <div className="Header">
        <div className="NavItems">
          <a href="#" className="HeaderItem">
            Home
          </a>
          <a href="#" className="HeaderItem">
            Posts
          </a>
          <a href="#" className="HeaderItem">
            About
          </a>
        </div>
        <input type="text" className="SearchField" />
```

```
        <input type="button" value="Search"
classNamе="SearchBtn" />
      </div>
    );
  }
}
```

Import it in App.js:

```
import "./App.css";
import Header from "./components/Header";
function App() {
 return (
   <div className="App">
     <Header />
   </div>
 );
}
export default App;
```

And the output is:



2. Component 2:

- The second component we create is the body component.
- In this component we just simply add a paragraph tag in the component which has Body written inside.
- And Body is written on the center of the Body component.

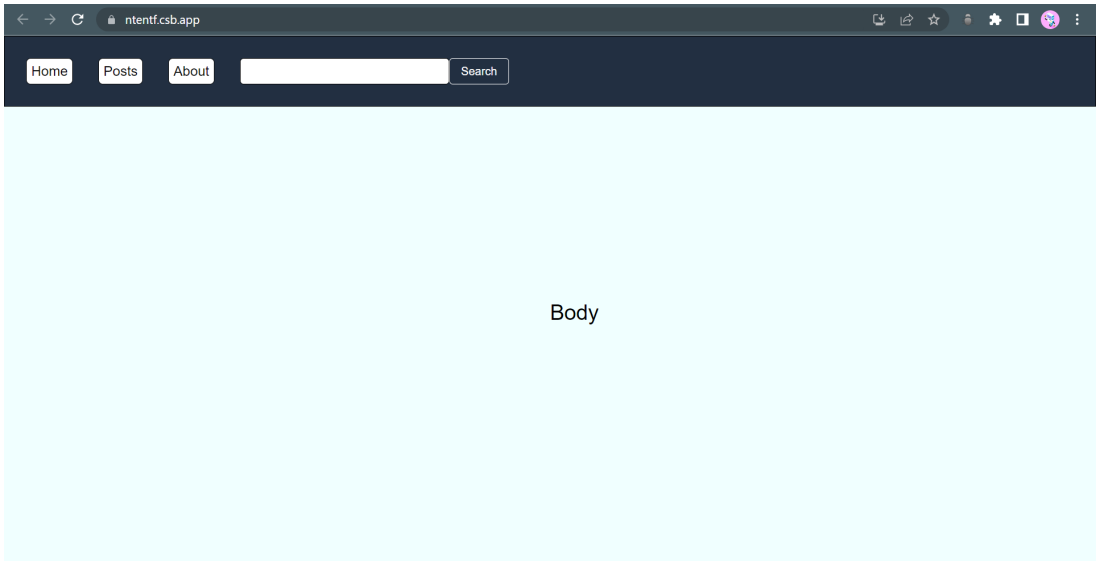
Body.js

```
import React, { Component } from "react";
import "./Styling.css";
export default class Body extends Component {
  render() {
    return (
      <div className="Body">
        <p>Body</p>
      </div>
    );
  }
}
```

Imported Body.js in App.js:

```
import "./App.css";
import Body from "./components/Body";
import Header from "./components/Header";
function App() {
 return (
   <div className="App">
     <Header />
     <Body/>
   </div>
 );
}

export default App;
```

And the Output is:



3. Component 3:

- The third component is the Footer component.
- In this component we add Footer written inside the paragraph tag similar to the 2nd component.
- And similar to the 2nd component it also should be in the center.

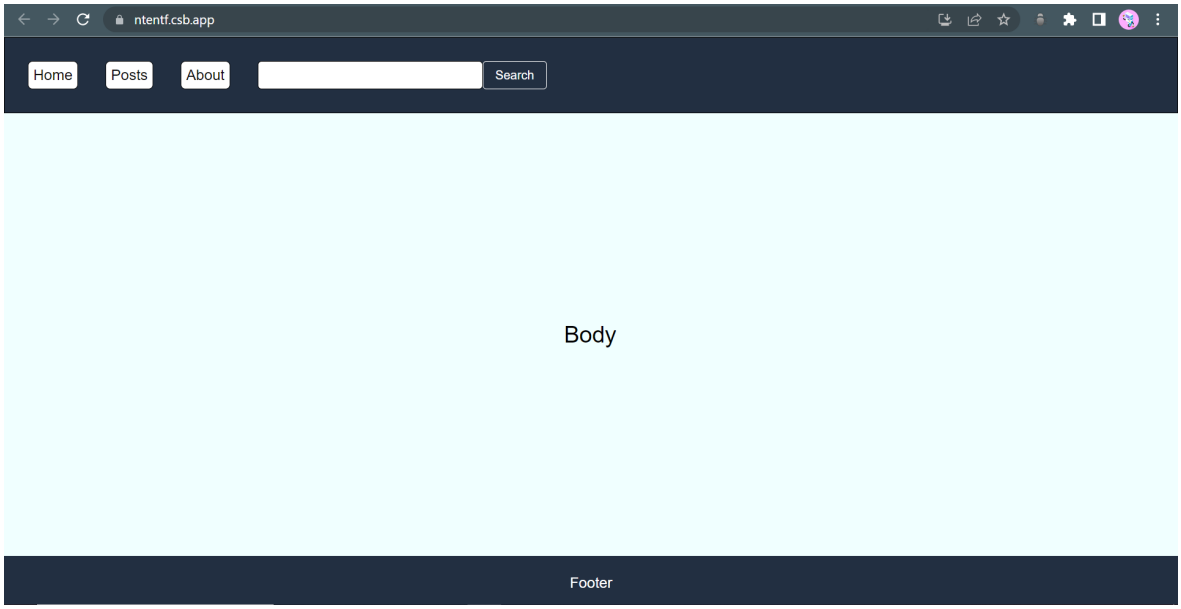
Footer.js:

```
import React, { Component } from "react";
import "./Styling.css";
export default class Footer extends Component {
 render() {
   return (
     <div className="Footer">
       <p className="Footer">Footer</p>
     </div>
   );
 }
}
```

Imported Footer.js in App.js:

```
import "./App.css";
import Body from "./components/Body";
import Footer from "./components/Footer";
import Header from "./components/Header";
function App() {
 return (
   <div className="App">
     <Header />
     <Body />
     <Footer/>
   </div>
 );
}

export default App;
```

And the final output



[Output Link]

Props in React Components

Topics covered:

- What are Props?
- How to use props?
    - Without Destructuring
        - Analyzing props
        - Extracting data
    - With Destructuring
    - Default Props

1. **Props**:
    - Props are similar to function arguments and are passed into the component as properties.
    - Props, an abbreviation for properties, transfer data between React components. The data flow within React's components is unidirectional (from parent to child only).
    - Props come in handy when you want the data flow in your app to be dynamic.

Example:

Add “taste” attribute in myElement:

```
const myElement = <Fruit taste='sweet'/>
```

The argument is given to the component as a props object:

Use the taste attribute in the component:

```
function Fruit(props) {
  return <h2>I am {props.taste} in taste!</h2>;
}
```

2. **Uses of Props:**
    - Props are another method for transferring data between components as parameters.
    - Props must be passed as an argument to your function to be used.
    - Comparable to calling your standard JavaScript functions with parameters

Without destructuring:

```
export default function FavColor(props) {
  const name = props.name;
  const color = props.color;
  return (
    <div>
      <h1>My name is {name}.</h1>
      <p>My favorite color is {color}.</p>
    </div>
  );
}
```

I'll now break down everything that just happened step by step:

Step 1: Introduce props as an argument.
This is what the function FavColor(props) in the code above does for us. As a result, you can use props in the component of your React app.

Step 2: Declare variables for props (s) **(Analyzing props)**

```
const name = props.name;
const color = props.color;
```

These variables, as you can see above, are distinct from conventional variables since their data relates to props.
If you do not wish to create variables for your props, you can just feed them into your template as follows:

```
      <h1>My name is {props.name}.</h1>
```

Step 3: In the JSX template, use a variable(s).
Now that you've defined your variables, you can put them wherever you want in your code.

```
return (
    <div>
      <h1>My name is {name}.</h1>
      <p>My favorite color is {color}.</p>
    </div>
  );
```

Step 4: Provide data on the App component's properties. **(Extracting Data)**
We've learned how to make our props, so the next step is to send data to them. We have previously imported the FavColor component, which is currently visible in the browser:

You can add default data to your properties so they don't appear empty when you declare them. You'll see how to do it in the final segment.

Recall that the current state of the App component:

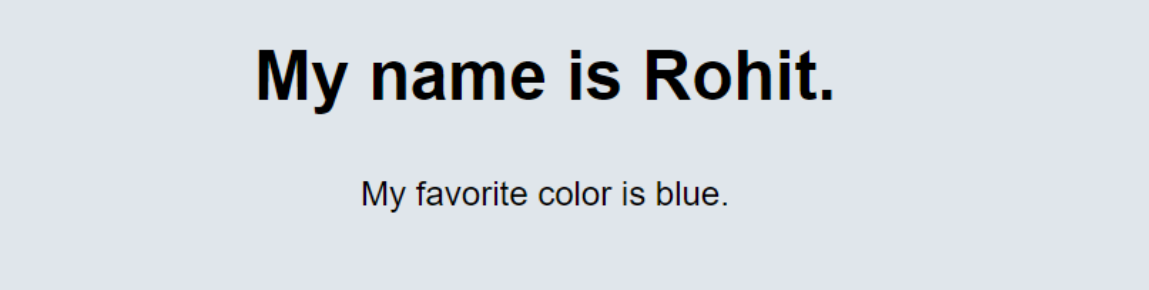
```
function App() {
  return (
    <div className="App">
    <FavColor/>
    </div>
    );
  }

  export default App;
```

You must be wondering just to whom the data would be sent. You enter the data in the form of attributes to achieve this. It appears as follows:

```
function App() {
  return (
    <div className="App">
    <FavColor name='Rohit' color='blue'/>
    </div>
    );
  }

  export default App;
```

What has changed? From "FavColor" to <FavColor name="Rohit" color="Blue"/> in this example. Since those characteristics are linked to the properties defined in the FavColor component, this won't result in an error for you.

You should see the following in your browser:



NOTE: The variable name is not the prop itself. If I had made a variable like this: **const myPropName = props.name** and used it in my template like this: **<h1>My name is {myPropName}.</h1>**
And then if I did this: **<FavColor name="Rohit" color="Blue"/>**, the code will still work correctly. The name attribute is obtained from props.name rather than the variable name that contains the prop.
Using the logic given in the FavColor component, you can now create data dynamically for any component. You may declare an unlimited number of props.

With Destructuring:
Except for the method for declaring the props, the code for this section is identical to that of the previous section.
In the preceding part, we stated our props as follows:

```
const name = props.name;
const color = props.color;
```

However, we do not have to do this with destructuring. Simply follow these steps:

```
export function FavColor({ name, color }) {
  return (
    <div>
      <h1>My name is {name}.</h1>
      <p>My favorite color is {color}.</p>
    </div>
  );
}
```

The first line of code contains the difference. We destructured and supplied the variables as the function's argument rather than the props.
Other than that, nothing has changed.
You can also pass in functions and even data from objects, so don't think you're limited to just using single variables as your props data.

Default Props:
You can specify a default value if you don't want your props data to be empty when you create them. This is how to accomplish it:

```
function FavColor({ name, color }) {
  return (
    <div>
      <h1>My name is {name}.</h1>
      <p>My favorite color is {color}.</p>
    </div>
  );
}
FavColor.defaultProps={
    name: "User",
    color:"transparent"
}
export default FavColor;
```

We declared default values for our props right before the component was exported at the very end of the code. To start, we used the component's name and the built-in defaultProps that appear when you create a React app, separated by a dot and a period.

Now, instead of being blank whenever we import this component, those values will be the initial values. The default values are overridden when data is passed to the child component, as we did in the previous sections.

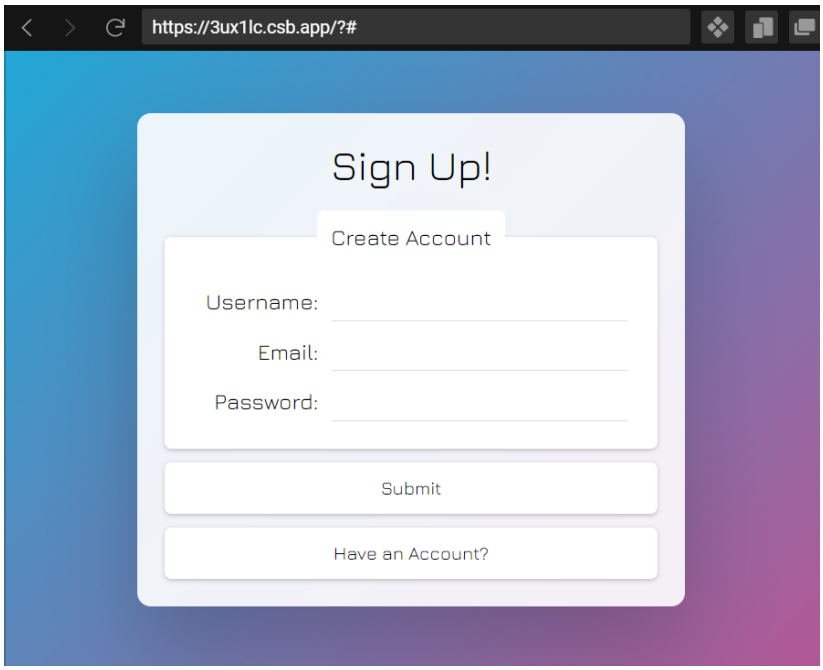
Output:

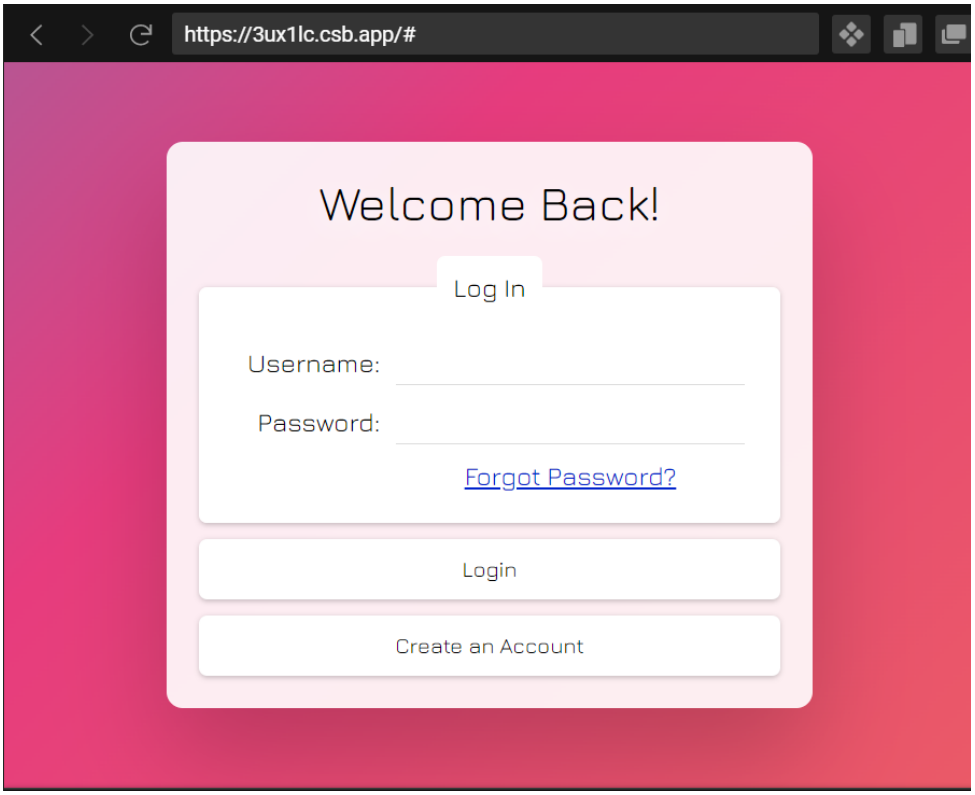
Assignment 2

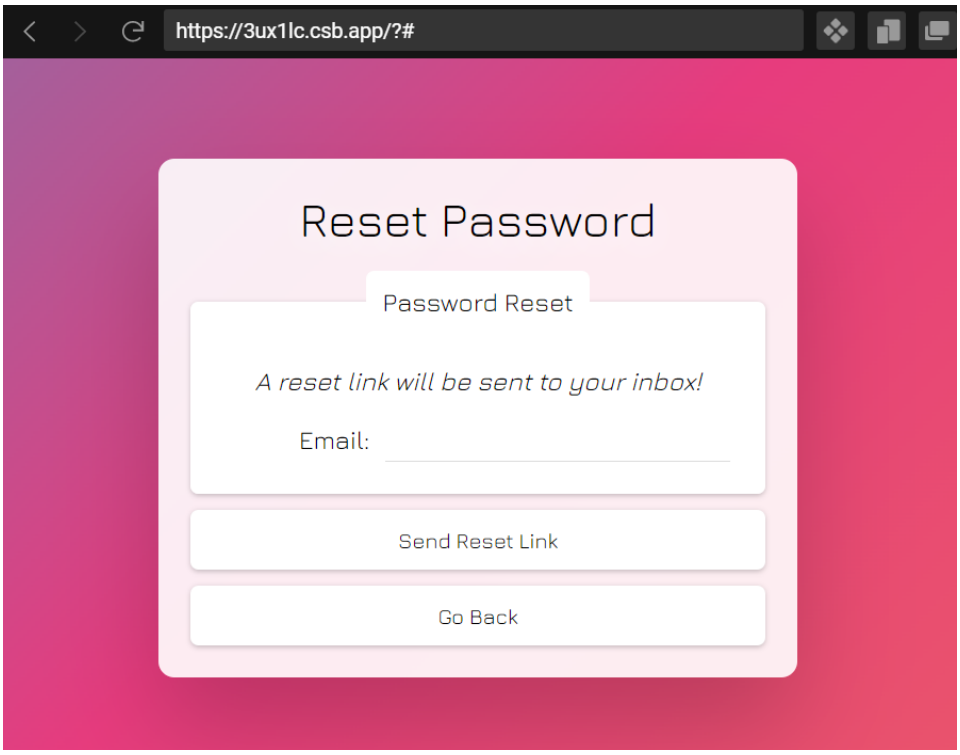
Using Conditional rendering in JSX, Create a signIn and signup page. Use Class-based components and initialize the state and you can use switch case/ ternary operator/ logical operator for conditional rendering.

Details:

1. Two input fields should be displayed that take username/email and password as input.
2. A button with innerText “signin” or “signup” should be rendered.
3. Set the default state as currentView: “Signup” and give three input fields to take email, username, and password as input.
4. And a button below with innerText “Already Have an Account” to switch to the signin page and have state currentView: “LogIn”.
5. “Signin” Page only takes a username and password as input and has a login button.
6. Below the login button, there must be a link to “Lost your password? ” and have state currentView: “PWReset”.
7. Put styling as given below:
    - Add animation in the background.
    - Use the font “Jura, Arial” and import it in the CSS file as:
        - @import url("https://fonts.googleapis.com/css?family=Jura:400");
    - Try to make the site responsive by using @media query.




https://3ux1lc.csb.app/#
Welcome Back!
Log In
Username:
Password:
Forgot Password?
Login
Create an Account


https://3ux1lc.csb.app/?#
Reset Password
Password Reset
A reset link will be sent to your inbox!
Email:
Send Reset Link
Go Back

Solution

State in React Components

Topics covered:

- What is the state of component?
- How to use state in class based components?
  - ➔ Syntax of setState
  - ➔ Use function to update state
- How to use state in functional component?
  - ➔ useState Hooks
- Difference between state and props.

1. **State of component**:
   - State is the most complicated aspect of React, and it is something that both new and veteran developers struggle with. We will therefore examine all the fundamentals of state in React.
   - A state object is already present in React components.
   - Property values for the component's properties are kept in the state object.
   - The component re-renders whenever the state object alters.

   Example:

   The constructor initializes the state object:

   ```
   this.state = {name: "sony",rollno: "202126" };
   ```

   Add a constructor method parameter specifying the state object:

```
export class Details extends Component {
  constructor(props) {
    super(props);
    this.state = { name: "Avni",rollno: "202126"};
  }
render() {
    return (
          <div>
             <p>
              Student Name: {this.state.name} <br /> Roll no:
              {this.state.rollno}
             </p>
           </div>
    );
  }
```

```
}
```

Output:

Student Name: Avni
Roll no:202126

2. **Use state in class-based component:**
   - An internal state object exists in React Class components.
   - Perhaps you noticed that we used state in the component constructor section previously.
   - The state object is where you save the component's property values.
   - The component re-renders as soon as the state object is modified.

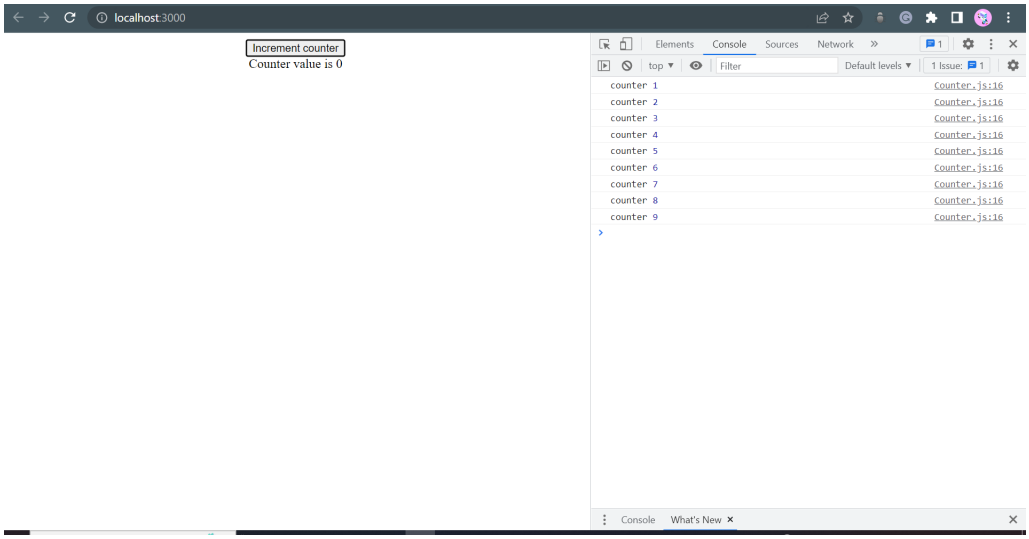
Example:

```
import React from "react";

class Counter extends React.Component {
 constructor(props) {
   super(props);

   this.state = {
     counter: 0
   };

   this.handleClick = this.handleClick.bind(this);
 }

 handleClick() {
     this.state.counter = this.state.counter + 1;

     console.log("counter", this.state.counter);
 }

 render() {
   const { counter } = this.state;
```

```
    return (
      <div>
        <button onClick={this.handleClick}>
         Increment counter</button>
        <div>Counter value is {counter}</div>
      </div>
    );
  }
}

const rootElement = document.getElementById("root");
ReactDOM.render(<Counter />, rootElement);
```

Examine what we're doing here:

- We first call super inside the constructor function and pass it props. After that, we defined the state as an object with the property of counter.
- In order to ensure that the handleClick method receives the appropriate context for this, we are also binding this's context to it.
- The counter is then updated and logged to the console inside the handleClick method.
- We also return the JSX that we wish to render on the UI inside the render method.



As you can see in the console, the counter is correctly updating, but it is not updating in the user interface.
This is because, inside the handleClick method, we are immediately updating the state as:

```
this.state.counter = this.state.counter + 1;
```

React does not re-render the component as a result (and explicitly updating state is likewise bad practice).

Directly updating or changing state in React is never a good idea because it will break your application. Additionally, if you make a direct state change, your component won't be re-rendered on state change.

Syntax of setState

- React provides us with a setState function that enables us to modify the state's value in order to make a change.

The syntax for the setState function is as follows:

```
setState(updater, [callback])
```

- *Updater* may be an object or a function.
- The optional function *callback* is executed after the state has been successfully modified.

Note: The component and all of its children are automatically rerendered when setState is called on them. As previously demonstrated with the renderContent function, we don't need to manually render again.

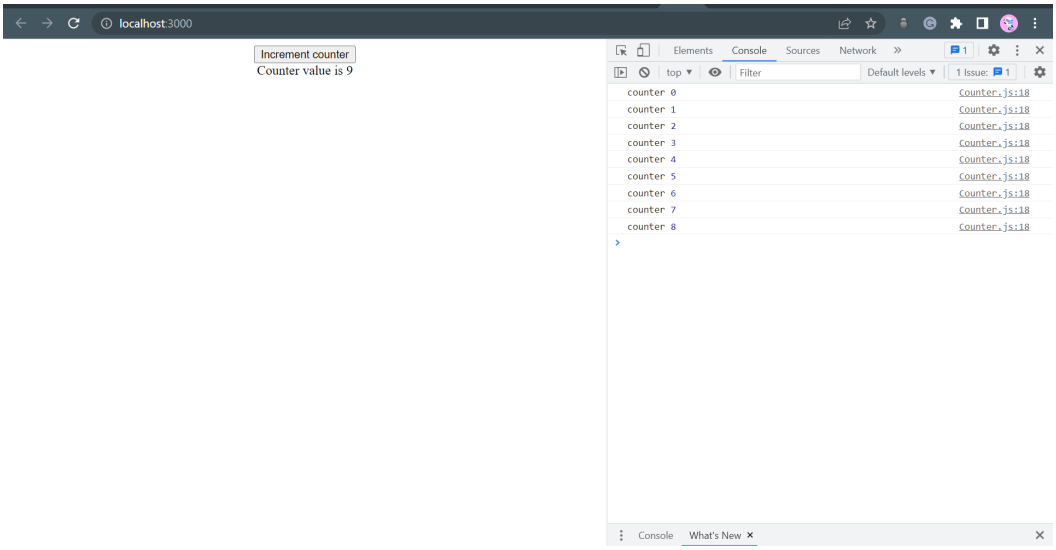
Use function to update state:

- The modified handleClick function appears to be as follows:

```
handleClick() {
  this.setState((pre) => {
    return { counter: pre.counter + 1 };
  });
  console.log("counter", this.state.counter);
}
```

In this case, we send a function as the first argument to the setState function, and we return a new state object with counter incremented by one based on the previous value of counter.

In the preceding code, we're using the arrow function, but any standard function would work.



- We're getting the updated value of the counter on the UI, as you can see. However, even if we inserted console.log after the this.setState method, we still get the previous counter value in the console.

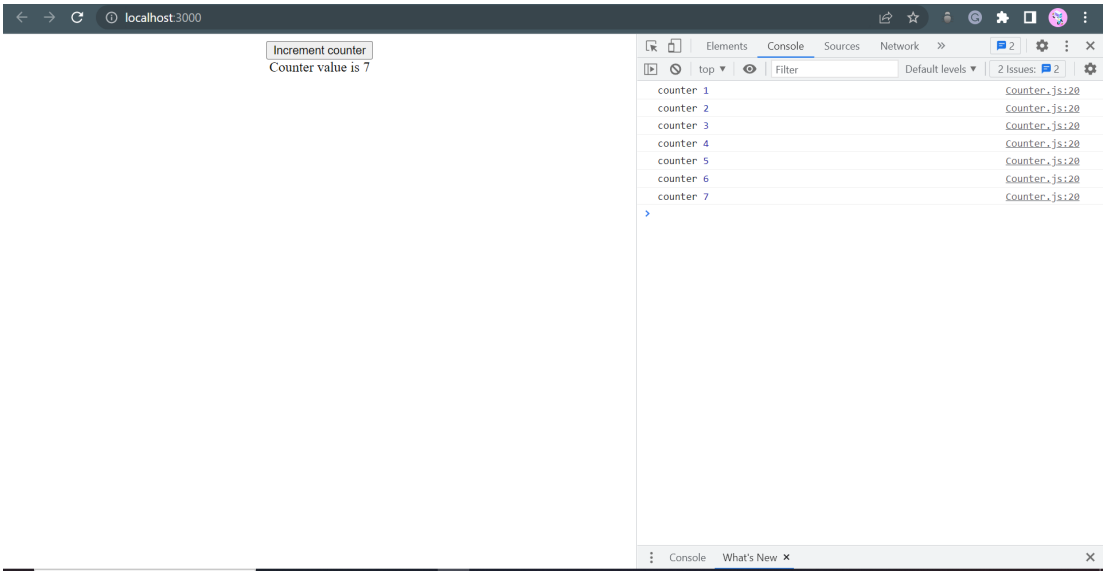
- This is due to the asynchronous nature of the setState function.

- This indicates that even though we called setState to increment the counter value by one, it did not happen right away.

- This is due to the fact that when we execute the setState function, the entire component is re-rendered - therefore React needs to examine what needs to be modified using the Virtual DOM algorithm and then do other checks for an efficient UI update.

You can supply a function as the second argument to the setState call, which will be performed once the state is modified, if you wish to quickly obtain the updated value of the state following the setState call.
Like this:

```
handleClick() {
   this.setState(
     (pre) => {
       return { counter: pre.counter + 1 };
     },
     () => {
       console.log("counter", this.state.counter);
     }
   );
 }
```



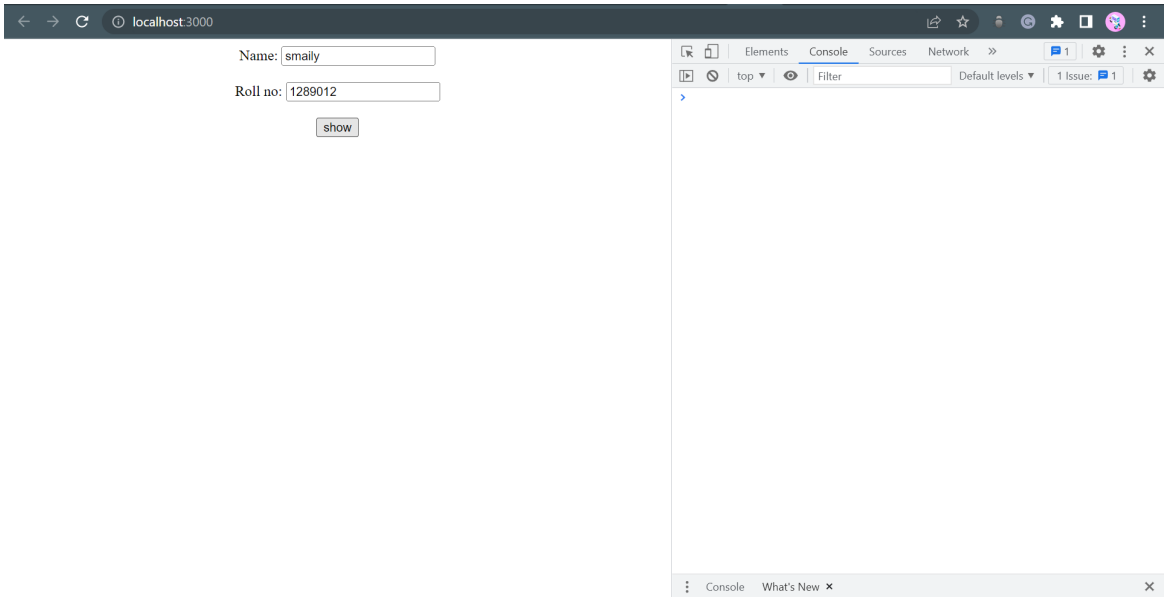
- We are sending two arguments in this case while calling the setState function. The first is a function that creates a new state, and the second is a callback function that is invoked whenever the state is altered. In the callback function, we are only logging the modified counter value to the console.

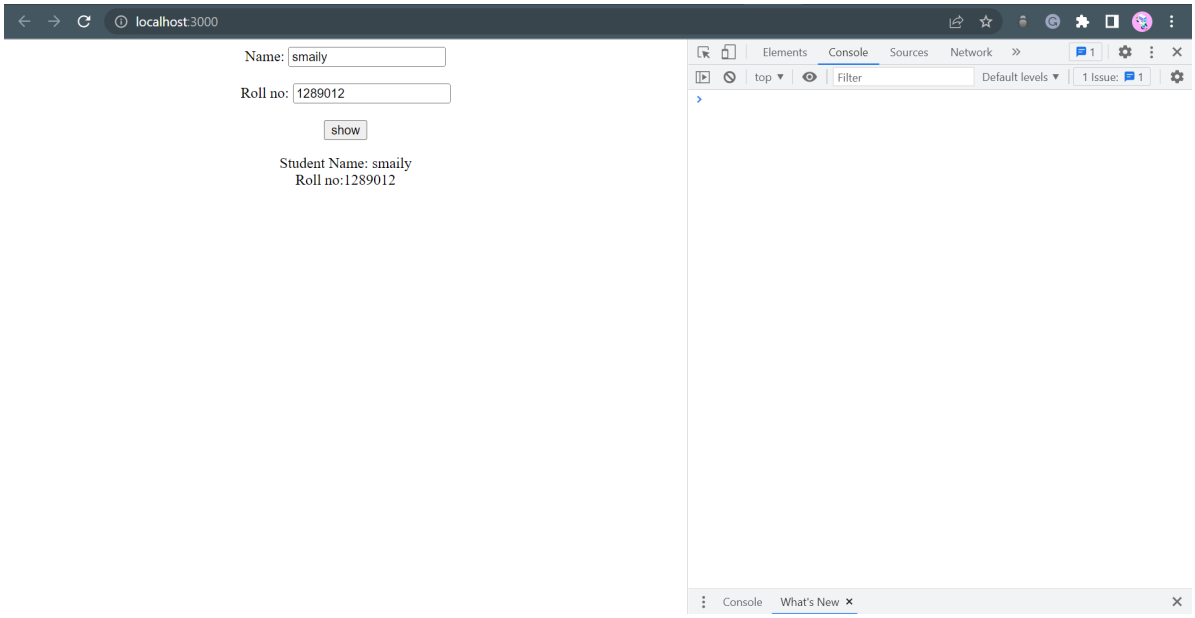
Example:

```
export class Details extends Component {
  constructor(props) {
    super(props);
    this.state = { name: "", roll_no: "", show: false };
  }
  changeHandle1 = (e) => {
    this.setState({ name: e.target.value });
  };
  changeHandle2 = (e) => {
    this.setState({ roll_no: e.target.value });
  };

  render() {
    return (
      <div>
        <label>Name: </label>
        <input
          type="text"
          value={this.state.name}
          onChange={this.changeHandle1}
        />
        <br />
        <br />
        <label>Roll no: </label>
        <input
```

```
          type="text"
          value={this.state.roll_no}
          onChange={this.changeHandle2}
        />
        <br />
        <br />
        <button
          onClick={() => {
            this.setState((pre) => {
              return { show: !pre.show };
            });
          }}
        >
          show
        </button>
        {this.state.show && (
          <p>
            Student Name: {this.state.name} <br /> Roll no:
            {this.state.roll_no}
          </p>
        )}
      </div>
    );
  }
}
```

Output:





Use state in functional component:

- With the exception of not having state and lifecycle methods, functional components are similar to class components. This is the reason they are sometimes referred to as stateless functional components.

- Hooks were added to React in version 16.8.0. Additionally, they have entirely altered the way we write code for React. We can leverage state and lifecycle methods inside functional components by using React Hooks.

- Functional components with additional state and lifecycle methods are known as react hooks.

  useState hook in react:

- We must apply the *useState* hook in order to define state using React Hooks.

- The initial value of the state is a parameter that the *useState* hook accepts.

- State is always an object in components that use classes. But when using useState, you can give any value—such as a number, string, boolean, object, array, null, and so on—as the initial value.

- The first value of the array that the useState hook delivers is the state's current value. The function we will use to update the state in a manner similar to the setState method is represented by the second value.

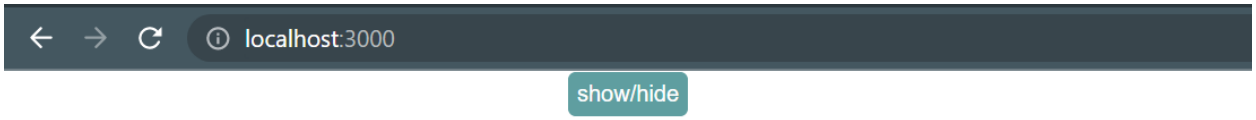
Let's look at a class-based component that uses the state object. Using hooks, we'll turn it into a functional component:

```
import React from "react";
export default class ShowHide extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      show: false,
    };
  }
  showFun = () => {
    this.setState((prev) => {
      return { show: !prev.show };
    });
  };
  render() {
    return (
      <div>
        {this.state.show && <div className="block">block</div>}
        <button onClick={this.showFun}>show/hide</button>
      </div>
    );
  }
}
```

Lets convert the above code in functional useState hooks:

```
import React, { useState } from "react";
export default function ShowHide() {
  const [show, setShow] = useState(false);

  return (
    <div>
      {show && <div className="block">block</div>}
      <button
        onClick={() => {
          setShow((prev) => !prev);
        }}
      >
        show/hide
      </button>
    </div>
  );
}
```

Output:



Let's examine the code shown above.

- We must import the *useState* hook like we did in the first line in order to use it.
- Within the App component, we are utilizing destructuring syntax and supplying *false* as the starting value when we use useState. The *show* and *setShow* variables were used to store the array values returned by useState.
- Prefixing the name of the function used to update the state with the set keyword, as in *setShow*, is a typical practice.
- The *setShow* function is called by defining an inline function and sending the updated *Show* value when the increment button is pressed.

Note that we used the sowh value, which we already had, to change the show (boolean value) by using the formula setShow((pre)=>!pre).
There is no need to relocate the code into a different function because the inline on click handler only contains one statement. However, you have the option to do that if the handler's code becomes complicated.

Difference between state and props:

Let's review and examine the key distinctions between props and state, then:

- While state allows components to create and maintain their own data, props allow them to receive data from outside the component.
- Data is passed using props, whereas state is used to manage data.
- Data from props is read-only and cannot be changed by an external component sending it.
- State data is private but can be altered by its own component (cannot be accessed from outside).
- The only way to transmit a prop is from a parent component to a child (unidirectional flow).
- SetState () method should be used to modify state.

Mapping in React

Topics covered:

- What are array methods?
  - map()
- How to use a map()?
  - Array
  - Array of object
- Mapping an array without using the map()

1. **Array methods**:
   - There are numerous array methods in JavaScript.
   - The .map() array method in React is one of the most helpful.
   - By applying a function to each element of the array using the .map() method, a new array is created as a consequence.
   - Map() in React can be used to create lists.

   Syntax: Map()

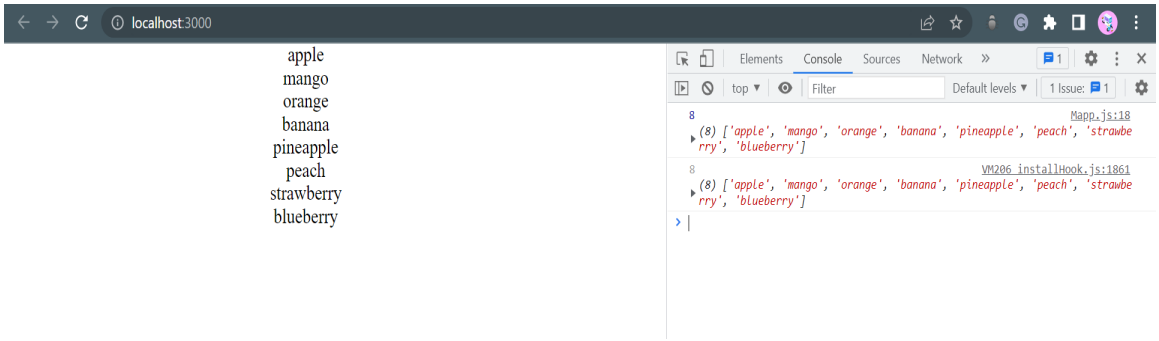
   Array.map((item)=> {item}

2. **How to use map():**

- **Array**

  Let's see how to use a map() method with the help of the example:

```
export default function Mapp() {
  var fruits = [
    "apple",
    "mango",
    "orange",
    "banana",
    "pineapple",
    "peach",
    "strawberry",
    "blueberry",
  ];

  return (
    <div>
      {console.log(fruits.length, fruits)}
      {fruits.map((fruit, key) => <p key={key}>{fruit}</p>)}
    </div>
  );
}
```



Here's how it works:

- We take the fruits array sent down as fruits and place it within a div.
- Create a new <p> for every fruit in the array by using our map() to traverse over each one. Keep in mind that the map() function accepts a function to apply to each element of the array. In this instance, the function only returns a <p> tag.

- **An array of objects:**
  - ➔ A map is a sort of data collection that stores data in the form of key-value pairs. The value recorded in the map must be mapped to the key. The map function in JavaScript can be used on any array. In a single line of code, we use the map function to map every element of the array to the custom components. This eliminates the need to repeatedly refer to components and their properties as array items.
  - ➔ You may quickly present similarly grouped data to your user using map(). The array will update in your component if it changes elsewhere else. You may conveniently show information using mapping rather than manually adding a new HTML element for each entry.

Employees.js: An array containing the employee's name and employee Id as key-value pairs.

```
var employees = [
    {
      name: "Deepak",
      empId: "123",
    },
    {
      name: "Yash",
      empId: "124",
    },
    {
      name: "Raj",
      empId: "125",
```

```
    },
    {
      name: "Rohan",
      empId: "126",
    },
    {
      name: "Puneet",
      empId: "127",
    },
    {
      name: "Vivek",
      empId: "128",
    },
    {
      name: "Aman",
      empId: "129",
    },
  ];
  export default employees;
```

Info.js: Username and employee Id as props in the Info component.

```
import React from 'react';

function Info (props){
  return (
    <div key={key} style={{ margin: 20, textAlign: "left" }}>
        <p>Name: {props.name}</p>
        <p>Employee Id:{props.empId}</p>
        <hr />
    </div>
    )
}
export default Info;
```

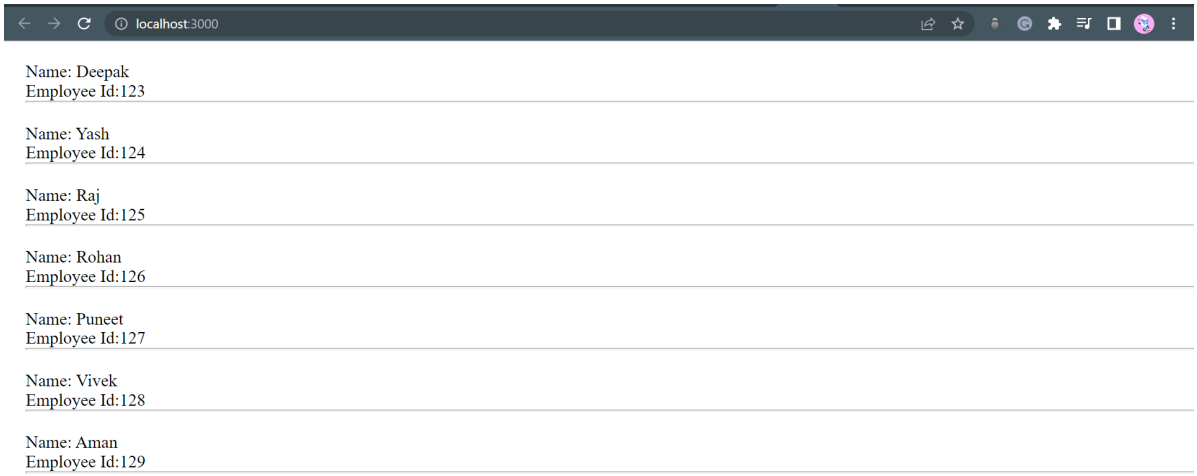
Example: Using the map function to map data from the *Employees.js* file to a custom Info component.

App.js: In the App.js file, import the Employees array and the Info component. Map each element of the Employees arrays to the Info component using the map function.

```
import React from 'react';
import './App.css';
import Users from './list';
import Info from './Info';

function App() {

return (
    <div style={{margin:'20px'}} >
    {Users.map((e)=>{
```

```
    return (
    <Info name={e.name} empId={e.rollNo}/>
    );})}
    </div>
);
}
export default App;
```

Output:



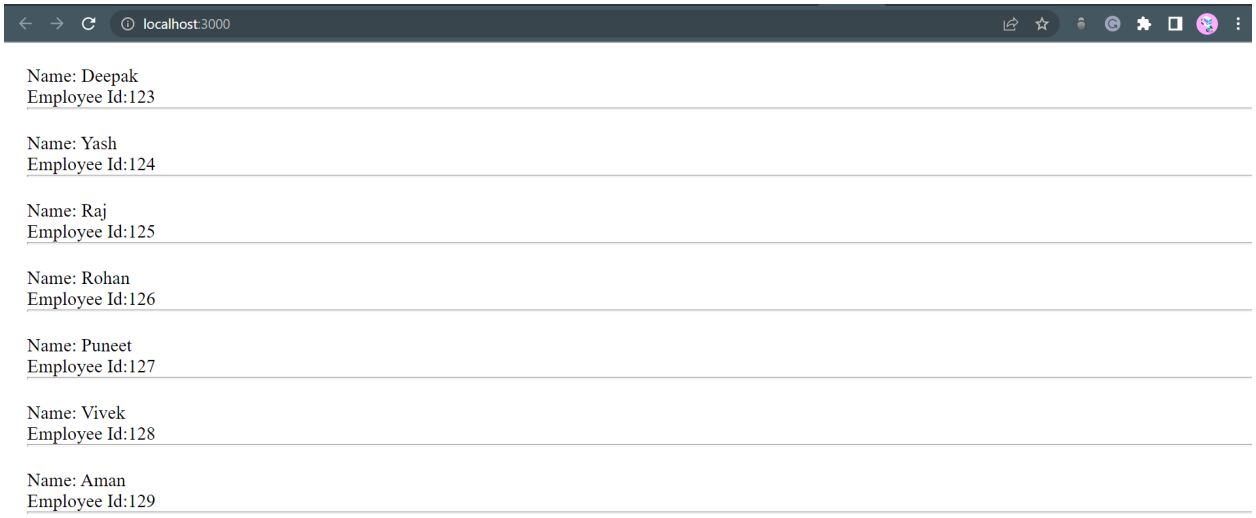
3. **Mapping an array without using the map():**

   Example: Using the map function to map data from the Employees.js file to the Info component.

```
import React from "react";
import "./App.css";
import Users from "./list";
import Info from "./Info";

function App() {
  return (
    <div style={{ margin: "20px" }}>
      <Info name={Users[0].name} empId={Users[0].empId} />
      <Info name={Users[1].name} empId={Users[1].empId} />
      <Info name={Users[2].name} empId={Users[2].empId} />
      <Info name={Users[3].name} empId={Users[3].empId} />
      <Info name={Users[4].name} empId={Users[4].empId} />
      <Info name={Users[5].name} empId={Users[5].empId} />
      <Info name={Users[6].name} empId={Users[6].empId} />
    </div>
  );
}
```

```
export default App;
```

Output:



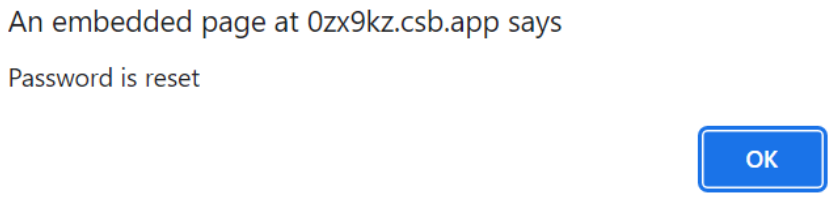
Both examples produce the same result, but utilizing the map method in the App.js file makes mapping array data to components easier and requires less code.

Assignment 3

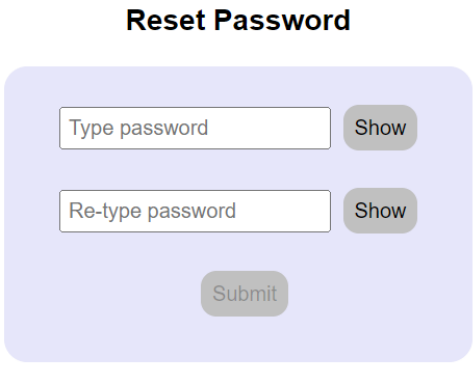
Create a reset password form with password hide and show functionality

Details:

1. Create a functional component and use react hooks.
2. Create two input fields of type password and re-type Password.
3. Create a submit button that is disabled and only enabled after typing the exact and correct password in both the password input fields by using useState.
4. Create a show and hide button along with each input field and add an onClick handler that shows and hides data of the input field by creating a state using react hooks (useState).
5. And on clicking submit button add an alert that says:



6. UI can be of your choice similar to:



Solution

Minor Assignment

(Todo List)

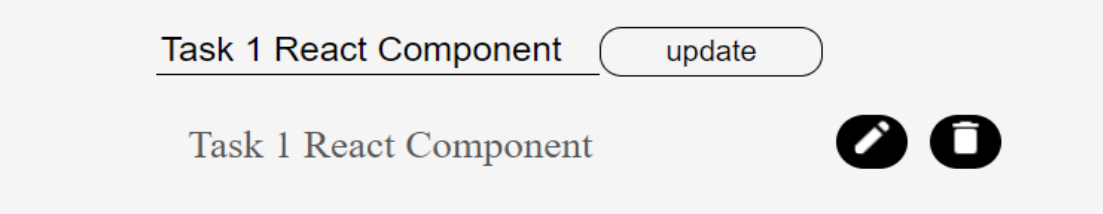
Problem Statement

You have to create a react app and a class based component called Todolist.js in which you have to create a form with conditional rendering.

Details:

1. Create a class based component and initialize a constructor with a super method inside it. (initialize states with different types according to need eg: this.state = { todos: [ ], value: " ", editing: " ", … }).
2. Create a form in which we have an input field and a button named as **Add task.** (with a condition in which "this.state.editing === false").

3. Create another form with input field and button named as **Update task.** (both forms are either in conditional rendering or in ternary operator).
4. Add edit and delete button to the list **"myList"** with map in state of todos.



5. Create a array of list of tasks named as **mylist** which maps the updated state of **todos.** (**hint:** this.state.todos.map((todo)=>{ … }))
6. Create different arrow functions for adding different functionalities. (**hints:** onAddTask, onChange, onDelete,....).

Process for Submission

Please upload your assignment files via this Google Form before the deadline 10th Feb 2023.

skill academy
by testbook

Minor Assignment

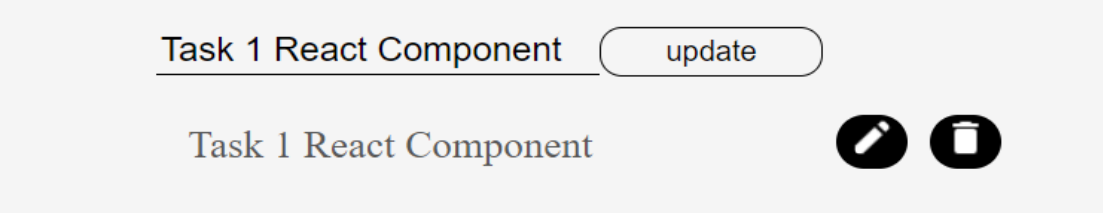
(Todo List)

You have to create an react app and a class based component called Todolist.js in which you have to create a form with conditional rendering.

Details:

1. Create a class based component and initialize a constructor with super method inside it. (initialize states with different types according to need eg: this.state = { todos: [ ], value: “ ”, editing: “ ”, … }).
2. Create a form in which we have an input field and a button named as **Add task.** (with a condition in which “this.state.editing === false”).

3. Create another form with input field and button named as **Update task.** (both forms are either in conditional rendering or in ternary operator).
4. Add edit and delete button to the list **“myList”** with map in state of todos.



5. Create a array of list of tasks named as **mylist** which maps the updated state of **todos.** (**hint:** this.state.todos.map((todo)=>{ … }))
6. Create different arrow functions for adding different functionalities. (**hints:** onAddTask, onChange, onDelete,....).

solution

React Hooks

Topics covered:

- What is the lifecycle of functional components?
- What are React Hooks?
    - useState Hook
    - useEffect Hook
- How to create custom hooks?

1. **Lifecycle of Functional components:**
    - Class-based components in React have access to a wide range of lifecycle functions, whereas functional components do not.
    - However, with the addition of Hooks, we now have one method that enables us to imitate the behavior of some of the class component lifecycle functions.

    We'll discover in-depth information regarding hooks.

All of this is possible because of the ***useState*** and ***useEffect*** hooks, two unique React capabilities that enable setting the initial state and utilizing lifecycle events in functional components. By skillfully utilizing these two hooks in your pure JavaScript routines, it is currently possible to simulate the performance of practically every supported lifecycle method.

2. **React Hooks:**

➔ React Hooks are simple JavaScript functions that can be used to separate reusable components from functional components. Hooks can be declarative and manage side effects.

➔ We can "*hook*" into React features like state and lifecycle methods using hooks.

Hooks follow three rules:

- ◆ Hooks can only be accessed from within *React function components*.
- ◆ Hooks can only be called at the component's top level.
- ◆ Hooks are not conditional.

➔ Note: Hooks cannot be used in class-based components.

React includes several standard built-in hooks:

- ❖ ***useState****:* To control states. Returns a stateful value as well as an updater function.
- ❖ ***useEffect***: To manage side effects such as API calls, subscriptions, timers, mutations, and other things.
- ❖ ***useContext****:* To return the context's current value.
- ❖ *useReducer:* An alternative to useState for complex state management.

- ***useMemo****:* This method returns a memoized value that can be used to improve performance.
- ***useRef:*** This method returns a ref object with the .current property. The ref object can be changed. It is primarily used to gain immediate access to a child component.

➔ **useState:**

- We can keep track of the state in a function component with the React *useState* Hook.
- The state generally describes data or parameters that applications need to track.

It must first be imported into our component before we can utilize the useState Hook.

```
1   import { useState } from "react";
```

Because useState is a named export, we destruct it from reactjs.

useState is called in our function component to initialize our state.
useState takes a starting state and returns two values:

- The present state.
- A function for modifying the state.

Example:
Initialize state at the function component's top:

```
1   import { useState } from "react";
2
3   function FavoriteColor() {
4     const [color, setColor] = useState("");
5   }
```

- Again, we are destructuring the values returned by ***useState***.
- The first value, color, represents our current situation.
- The function used to update our state is setColor, the second value.
- These are variables that can be named whatever you choose.
- Finally, we make the initial state an empty string: useState("")

Now, wherever we want in our component, we may include our state.

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

function FavoriteColor() {
  const [color, setColor] = useState("red");

  return <h1>My favorite color is {color}!</h1>;
}

export default FavoriteColor;

```

Output:

My favorite color is red!

Modify State:

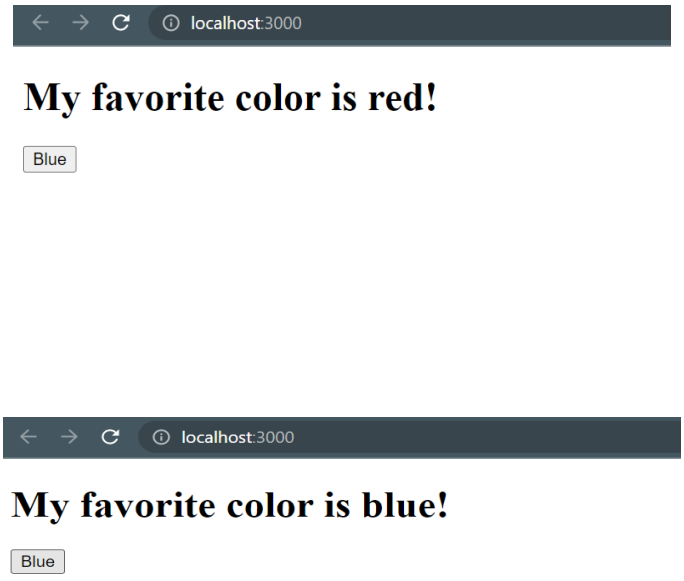
- We utilize our state updater function to update our state.
- Never should we update state directly.

We use our state updater function to update our state.

We should never update state directly.

```jsx
import { useState } from "react";
function FavoriteColor() {
  const [color, setColor] = useState("red");

  return (
    <div>
      <h1>My favorite color is {color}!</h1>
      <button type="button" onClick={() => setColor("blue")}>
        Blue
      </button>
    </div>
  );
}

export default FavoriteColor;
```

Output:



➔ **useEffect:**

- To perform side effects in your components, useEffect Hook is used.
- Fetching data, directly updating the DOM, and timers are examples of side effects.
- useEffect accepts two parameters. The second parameter is not required.

```
useEffect(<function>, <dependency>)
```

Let us see an example to understand what useEffect can do,

```
import React, { useState, useEffect } from "react";
export default function FunctionHooks() {
  const [value, setValue] = useState(0);
  useEffect(() => {
    document.title = `current value is ${value}`;
    console.log(document.title);
  });
  return (
    <div>
      <h1>current value is {value}</h1>
      <button onClick={() => setValue(value + 1)}>Add by
1</button>
    </div>
  );
}
```

- Every render includes useEffect. That is, when the value changes, a render occurs, which then causes another effect to occur.
- This is not what we desire. There are several methods for controlling when side effects occur.
- The second parameter, which accepts an array, should always be included. In this array, we can optionally pass dependencies to useEffect.

When dependency is not passed:

```
useEffect(() => {
  //executes on every render
});
```

When an empty Array is passed:

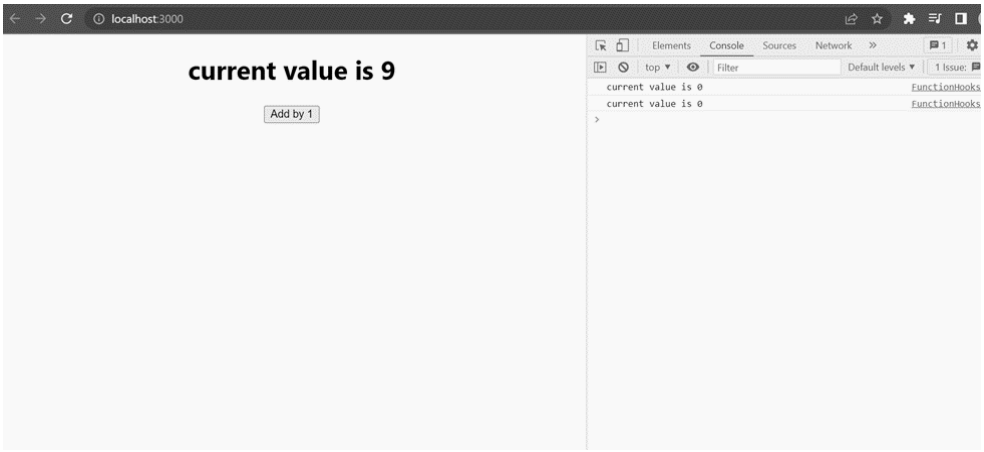
```
useEffect(() => {
  //Executes only on the first render
}, []);
```

When props and states are passed:

```
useEffect(() => {
  //Executes on the first render
  //When any dependency value changes
}, [prop, state]);
```

Let's see what will happen when we pass an empty are in dependencies:

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

export default function FunctionHooks() {
  const [value, setValue] = useState(0);
  useEffect(() => {
    document.title = `current value is ${value}`;
    console.log(document.title);
  }, []);

  return (
    <div>
      <h1>current value is {value}</h1>
      <button onClick={() => setValue(value + 1)}>Add by
1</button>
    </div>
  );
}
```



And what will happen if we pass value as a dependency;

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

export default function FunctionHooks() {
  const [value, setValue] = useState(0);
  useEffect(() => {
    document.title = `current value is ${value}`;
    console.log(document.title);
  }, [value]);

  return (
    <div>
      <h1>current value is {value}</h1>
      <button onClick={() => setValue(value + 1)}>Add by
1</button>
    </div>
  );
}
```

3. **Custom Hooks:**
    - Hooks are functions that can be reused.
    - When you have component logic that needs to be shared by several components, we can extract it to a custom Hook.

Create a Hook:

- The following code gets and displays data from our Home component.
- To get data, we'll use the JSONPlaceholder service.
- This service is ideal for testing apps when no data is available.
- See the JavaScript Fetch API section for further information.
- Use the JSONPlaceholder service to retrieve fictitious "todo" items and show their titles on the page:

- Because the fetch logic may be used in other components, we will extract it into a new Hook.
- Fetch logic should be transferred to a new file and used as a custom hook:

useFetch.js:

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

const useFetch = (url) => {
  const [data, setData] = useState(null);

  useEffect(() => {
    fetch(url)
      .then((res) => res.json())
      .then((data) => setData(data));
  }, [url]);

  return [data];
};

export default useFetch;
```

Home.js

```
import useFetch from "./useFetch";

const Home = () => {
  const [data] = useFetch("https://jsonplaceholder.typicode.com/todos");

  return (
    <div>
      {data &&
        data.map((item) => {
          return <p key={item.id}>{item.title}</p>;
        })}
    </div>
  );
};
export default Home;
```

- The entire logic required to fetch our data is contained in a function named useFetch that we have built in a new file called useFetch.js.
- The hard-coded URL was deleted, and its place was taken by a url variable that could be provided to the unique Hook.
- Finally, we are sending data back to our Hook.
- We are importing our useFetch Hook into index.js and using it just like any other Hook. Here is where we provide the URL to use to retrieve the data.
- Now, any component can utilize this custom Hook to fetch data from any URL.

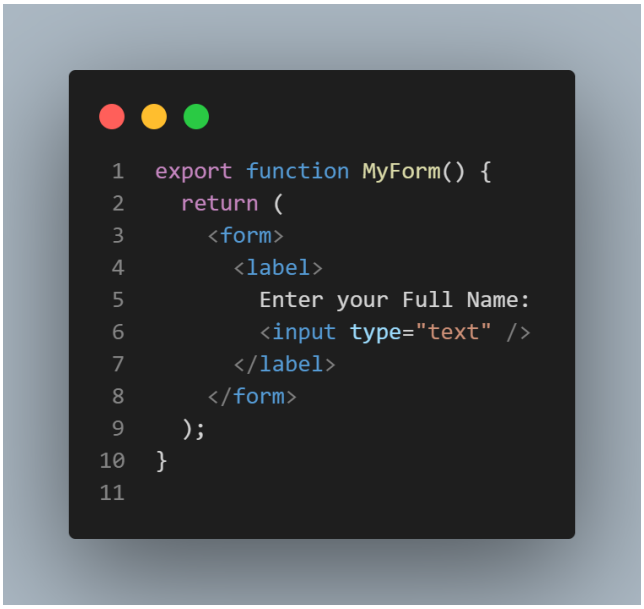
React Forms

Topics covered:

- How to add Forms in react?
  - Types of inputs
  - Adding a label to an input
  - To provide an initial point value for an input
- How to handle React forms?
  - Submitting form
  - Handling multiple inputs
- Control input with state variable
- RegEx in React
  - RegEx
  - Example

1. **Add Form in react component:**

   React allows you to add a form just like any other element:



```
export function MyForm() {
  return (
    <form>
      <label>
        Enter your Full Name:
        <input type="text" />
      </label>
    </form>
  );
}
```

   - The form will submit as expected, and the page will refresh.
   - However, with React, this is usually not what we want to happen.
   - In order to let React manage the form, we want to avoid this default behavior.

a. Type of inputs:

Render an input component in order to see the input. It will by default be a text input. For a checkbox, a radio button, or any other input type, you can pass type=" checkbox," type=" radio," or another input type.

```
export default function MyForm() {
  return (
    <div>
      <label>
        Text input: <input name="myInput" />
      </label>
      <hr />
      <label>
        Checkbox: <input type="checkbox" name="myCheckbox" />
      </label>
      <hr />
      <p>
        Radio buttons:
        <label>
          <input type="radio" name="myRadio" value="option1" />
          Option 1
        </label>
        <label>
          <input type="radio" name="myRadio" value="option2" />
          Option 2
        </label>
        <label>
          <input type="radio" name="myRadio" value="option3" />
          Option 3
        </label>
      </p>
    </div>
  );
}
```

Text input: Some initial value

Checkbox:

Radio buttons:
- Option 1
- Option 2
- Option 3

b. Adding a label to an input:

- Every <input> tag should be placed inside a <label> tag. This informs the browser that this label is linked to that input. The browser will immediately focus the input when the user clicks the label. It's also important for accessibility: when the user focuses on the related input, a screen reader will announce the label caption.
- If you can't nest <input> within a <label>, connect them by supplying the same ID to <input id> and <label htmlFor>. Use useId to generate such an ID to avoid conflicts between several instances of the same component.

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

export default function Form() {
  const ageInputId = useId();
  return (
    <div>
      <label>
        Your first name:
        <input name="firstName" />
      </label>
      <hr />
      <label htmlFor={ageInputId}>Your age:</label>
      <input id={ageInputId} name="age" type="number" />
    </div>
  );
}
```

Your first name:

Your age:

c. To provide an initial point value for an input:

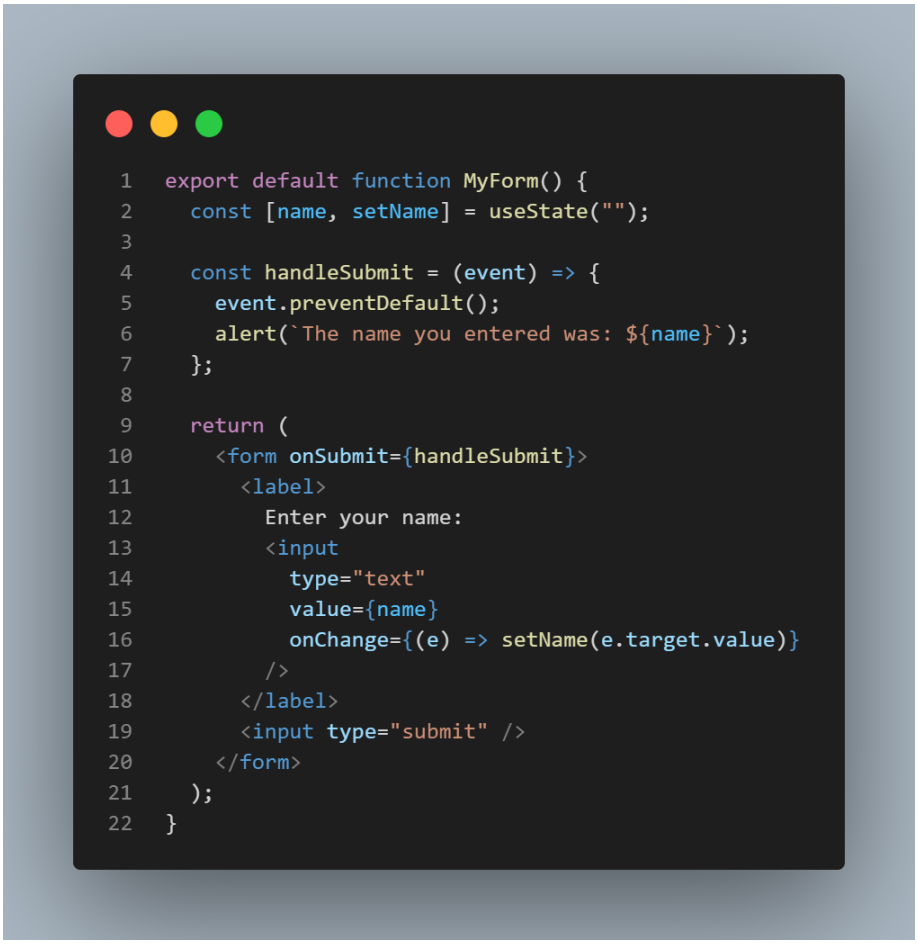
- Any input can have its initial value specified as an option. For text inputs, use it as the defaultValue string. Instead, checkboxes and radio buttons should use the defaultChecked boolean to specify the initial value.

```
export default function MyForm() {
  return (
    <div>
      <label>
        Text input: <input name="myInput" defaultValue="Some initial value" />
      </label>
      <hr />
      <label>
        Checkbox:
        <input type="checkbox" name="myCheckbox" defaultChecked={true} />
      </label>
      <hr />
      <p>
        Radio buttons:
        <label>
          <input type="radio" name="myRadio" value="option1" />
          Option 1
        </label>
        <label>
          <input
            type="radio"
            name="myRadio"
            value="option2"
            defaultChecked={true}
          />
          Option 2
        </label>
        <label>
          <input type="radio" name="myRadio" value="option3" />
          Option 3
        </label>
      </p>
    </div>
  );
}
```

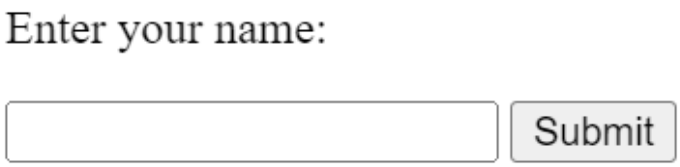
2. **Handle React Forms:**
    - Handling forms refers to how you handle data when it changes or is submitted.
    - Form data is typically handled by the DOM in HTML.
    - In React, components handle form data frequently.
    - When data is handled by components, it is all kept in the component state.
    - Changes can be controlled by adding event handlers to the onChange attribute.
    - The useState Hook can be used to keep track of each input's value and offer a "single source of truth" for the entire application.

    a. **Submitting form:**

    You can control the submit action by adding an event handler to the form's onSubmit attribute:



```jsx
export default function MyForm() {
  const [name, setName] = useState("");

  const handleSubmit = (event) => {
    event.preventDefault();
    alert(`The name you entered was: ${name}`);
  };

  return (
    <form onSubmit={handleSubmit}>
      <label>
        Enter your name:
        <input
          type="text"
          value={name}
          onChange={(e) => setName(e.target.value)}
        />
      </label>
      <input type="submit" />
    </form>
  );
}
```



b. Handling multiple inputs:

<input> should be enclosed in a <form> with a <button type="submit">. Your form's onSubmit event handler will be called. The browser will automatically send the form's data to the current URL and reload the page. Calling e.preventDefault() will allow you to alter that behavior. Use new FormData(e.target) to read the form data.

```jsx
export default function MyForm() {
  function handleSubmit(e) {
    // Prevent the browser from reloading the page
    e.preventDefault();

    // Read the form data
    const form = e.target;
    const formData = new FormData(form);

    // You can pass formData as a fetch body directly:
    fetch("/some-api", { method: form.method, body: formData });

    // Or you can work with it as a plain object:
    const formJson = Object.fromEntries(formData.entries());
    console.log(formJson);
  }

  return (
    <form method="post" onSubmit={handleSubmit}>
      <label>
        Text input: <input name="myInput" defaultValue="Some initial value" />
      </label>
      <hr />
      <label>
        Checkbox:{" "}
        <input type="checkbox" name="myCheckbox" defaultChecked={true} />
      </label>
      <hr />
      <p>
        Radio buttons:
        <label>
          <input type="radio" name="myRadio" value="option1" /> Option 1
        </label>
        <label>
          <input
            type="radio"
            name="myRadio"
            value="option2"
            defaultChecked={true}
          />{" "}
          Option 2
        </label>
        <label>
          <input type="radio" name="myRadio" value="option3" /> Option 3
        </label>
      </p>
      <hr />
      <button type="reset">Reset form</button>
      <button type="submit">Submit form</button>
    </form>
  );
}
```

Text input: Some initial value

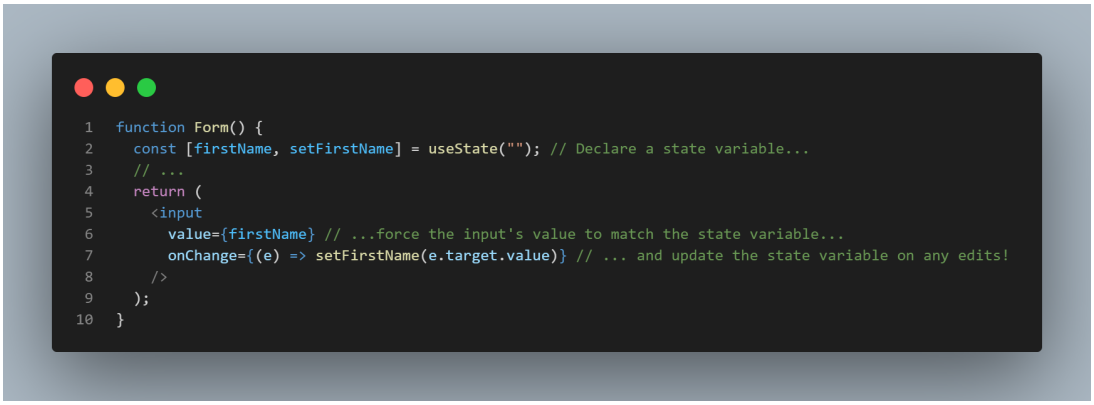
Checkbox: ☑

Radio buttons:

- ○ Option 1
- ◉ Option 2
- ○ Option 3

Reset form | Submit form

3. **Control input with state variable**
    - An uncontrolled input is <input />. Even if you specify an initial value, such as <input defaultValue="Initial text" />, your JSX will just specify the initial value. It has no say over what the value should be right now.
    - Pass the value prop to a controlled input to render it (or checked for checkboxes and radios). React will always force the input to have the value you gave. In most cases, you'll control an input by declaring a state variable:



```
function Form() {
  const [firstName, setFirstName] = useState(""); // Declare a state variable...
  // ...
  return (
    <input
      value={firstName} // ...force the input's value to match the state variable...
      onChange={(e) => setFirstName(e.target.value)} // ... and update the state variable on any edits!
    />
  );
}
```

If you needed state anyhow, for example, to re-render your UI on every edit, a controlled input makes sense.

```
function Form() {
  const [firstName, setFirstName] = useState('');
  return (
    <>
      <label>
        First name:
        <input value={firstName} onChange={e => setFirstName(e.target.value)} />
      </label>
      {firstName !== '' && <p>Your name is {firstName}.</p>}
      ...

```

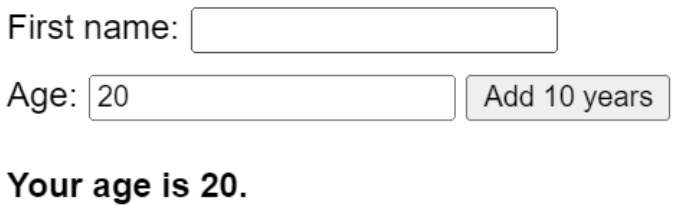
It's also useful if you want to provide multiple ways to change the input state (such as by clicking a button):

```jsx
function Form() {
  // ...
  const [age, setAge] = useState('');
  const ageAsNumber = Number(age);
  return (
    <>
      <label>
        Age:
        <input
          value={age}
          onChange={e => setAge(e.target.value)}
          type="number"
        />
        <button onClick={() => setAge(ageAsNumber + 10)}>
          Add 10 years
        </button>
```

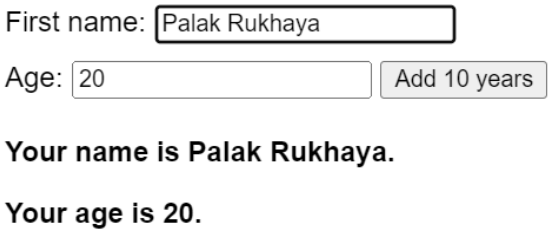
Controlled components should not be given undefined or null values. If the initial value must be empty (as in the firstName field below), set your state variable to an empty string (' ').

```jsx
import { useState } from "react";

export default function Form() {
  const [firstName, setFirstName] = useState("");
  const [age, setAge] = useState("20");
  const ageAsNumber = Number(age);
  return (
    <div>
      <label>
        First name:
        <input
          value={firstName}
          onChange={(e) => setFirstName(e.target.value)}
        />
      </label>
      <label>
        Age:
        <input
          value={age}
          onChange={(e) => setAge(e.target.value)}
          type="number"
        />
        <button onClick={() => setAge(ageAsNumber + 10)}>Add 10 years</button>
      </label>
      {firstName !== "" && <p>Your name is {firstName}.</p>}
      {ageAsNumber > 0 && <p>Your age is {ageAsNumber}.</p>}
    </div>
  );
}
```

Initial Output:



Output after applying changes:



4. **RegEx in React (Regular Expression):**
   a. **RegEx(or Regular Expressions):**
      - A string is examined to see if it contains the provided search pattern using a RegEx, or regular expression, which is a series of characters that defines a search pattern.
      - Additionally, it is used to validate strings that contain email, passwords, and other data.
   b. **Example:**
      In this example, we'll create a React application for authentication that asks the user for their email and password and determines whether or not they've been validated.

      For our application's email and password validation, we have Regex.js, which contains all the regular expressions.

Regex.js:

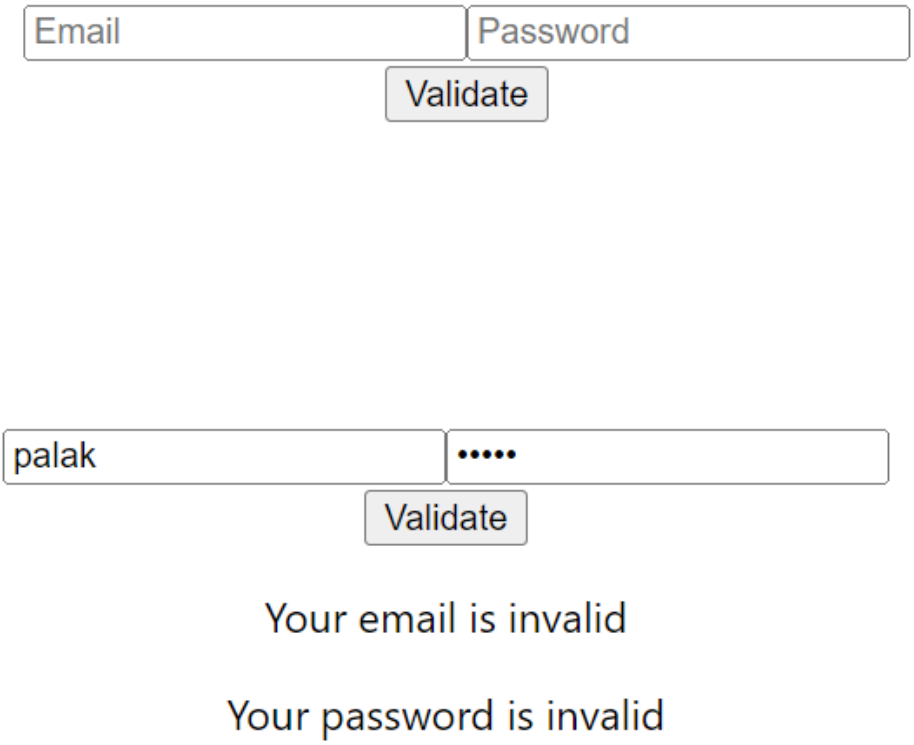
```
export const validEmail = new RegExp(
  "^[a-zA-Z0-9._:$!%-]+@[a-zA-Z0-9.-]+.[a-zA-Z]$"
);
export const validPassword = new RegExp("^(?=.*?[A-Za-z])(?=.*?[0-9]).{6,}$");

```

App.js

```
import React, { useState } from 'react';
import { validEmail, validPassword } from './regex.js';

const App = () => {
   const [email, setEmail] = useState('');
   const [password, setPassword] = useState('');
   const [emailErr, setEmailErr] = useState(false);
   const [pwdError, setPwdError] = useState(false);
   const validate = () => {
      if (!validEmail.test(email)) {
         setEmailErr(true);
      }
      if (!validPassword.test(password)) {
         setPwdError(true);
      }
   };
   return (
      <div>
         <input
            type="email"
            placeholder="Email"
            value={email}
            onChange={(e) => setEmail(e.target.value)}
         />
         <input
            type="password"
            placeholder="Password"
            value={password}
            onChange={(e) => setPassword(e.target.value)}
         />
         <div>
            <button onClick={validate}>Validate
         </div>
         {emailErr && <p>Your email is invalid</p>}
         {pwdError && <p>Your password is invalid</p>}
      </div>
   );
};
export default App;
```

When the user clicks the Validate button in the above example, the email and password are validated and the result is displayed.

Output:
This will result in the following outcome:



APIs and Events

Topics covered:

- What are APIs?
    - Protocols of APIs
    - API requests
- How to add APIs in react?
    - Using Fetch
    - Using Axios
- Difference between Axios and fetch

1. **APIs**:

   The API defines commands, functions, protocols, and objects that programmers can use to build software or interact with external systems.
   - APIs enable various unrelated software products to integrate and communicate with other applications and data.
   - APIs also enable developers to add features and functionality to software by using the APIs of other developers.
   - A wide variety of APIs is used in today's workplace, mobile, and web software.
   - However, not all APIs are created equal. Developers can work with a variety of API types, protocols, and architectures to meet the specific demands of various applications and enterprises.

   a. **Protocols of APIs:**

      APIs transmit instructions and data, which requires defined protocols and architectures—the guidelines, frameworks, and limitations that control an API's operation. REST, RPC, and SOAP are the three major categories of API protocols or architectures used today. These may all be referred to as "formats," each with distinctive properties and trade-offs that are used for various objectives.
      - **REST:** The most popular technique for developing APIs is the *REpresentational State Transfer* (REST) architecture.
        REST is based on a client/server model that separates both the front and back ends of the API and allows for a great deal of freedom in development and implementation.

Because REST is stateless, the API retains no data or status between requests. For slow or non-time-sensitive APIs, REST enables caching, which caches answers.

REST APIs, also known as **RESTful APIs**, can communicate directly or via intermediary systems such as API gateways and load balancers.

- **RPC:** The *Remote Procedure Call* (RPC) protocol is a straightforward method for sending and receiving several

  The remote procedure call (RPC) protocol is a straightforward method for sending and receiving several. RPC APIs perform executable operations or processes, whereas REST APIs primarily exchange data or resources like documents.

  RPC can use two alternative languages for coding: JSON and XML; these APIs are known as JSON-RPC and XML-RPC, respectively.
- **SOAP:** The World Wide Web Consortium created the *simple object access protocol (SOAP)*, a messaging standard that is widely used to establish web APIs, typically with XML.

  Numerous internet-based communication protocols, including HTTP, SMTP, and TCP/IP, are supported by SOAP.

  Developers can easily add features and functionality to SOAP APIs since SOAP is flexible and independent of writing styles.

  The SOAP method specifies the features and modules that go into a SOAP message, the supported communication protocol(s), how SOAP messages are constructed, and how the SOAP message is handled.

b. API request:

The HTTP request types GET, POST, PUT, PATCH, and DELETE are the most frequently used ones. In other words, these operations are CRUD (create, read, update, and delete).

- **GET**: A GET request reads/retrieves data from a web server. If the data is successfully retrieved from the server, GET returns an HTTP status code of 200 (OK).
- **POST**: A POST request is used to transmit data to the server (file, form data, etc.). It returns an HTTP status code of 201 upon successful creation.

- **PUT**: To update data on the server, submit a PUT request. It passes data that is contained in the body payload in place of the complete content at a specific position. It will create one if there are no resources that match the request.
- **PATCH**: A PATCH request alters a portion of the data, whereas a PUT request modifies the entire request. Only the content you want to update will be replaced.
- **DELETE**: A DELETE request is used to remove data from a specific location on a server.

2. **Add APIs in React:**
    a. **Using Fetch:**
        - A built-in JavaScript method for obtaining resources from a server or API endpoint is fetch(). It is comparable to XMLHttpRequest, except the fetch API is more powerful and flexible.
        - It introduces topics like CORS and the HTTP Origin header semantics, which it replaces with distinct definitions elsewhere.
        - The fetch() API method always requires a mandatory argument, which is the path or URL of the resource to be retrieved. Whether the request is successful or not, it delivers a promise pointing to the response. As the second argument, you can optionally send in an init options object.

    Parameters for fetch():

        - **resource**
          This is the route to the resource you wish to retrieve; it may be a request object or a straight link to the resource path.
        - **init**
          This is an object that contains any special parameters or login information you want to include with your fetch() call. Some of the choices that could be included in the init object include the following:
            - **method:** This is where the HTTP request method, such as GET, POST, etc., is specified.
            - **headers:** You can use this to include any headers to your request that are typically contained in an object or an object literal.

- **body:** You can add a Blob, BufferSource, FormData, URLSearchParams, USVString, or ReadableStream object as the body of your request by providing it here.
- **mode:** You can use this to indicate the request mode, such as cors, no-cors, or same-origin.
- **credentials:** This option must be provided if you think about sending cookies automatically for the current domain. It allows you to define the request credentials you want to use for the request.

SYNTAX PRINCIPLES FOR USING THE FETCH() API

Look at the code below to see how easy it is to build a basic fetch request:

```
1
2  fetch("https://api.github.com/users/hacktivist123/repos")
3    .then((response) => response.json())
4    .then((data) => console.log(data));
```

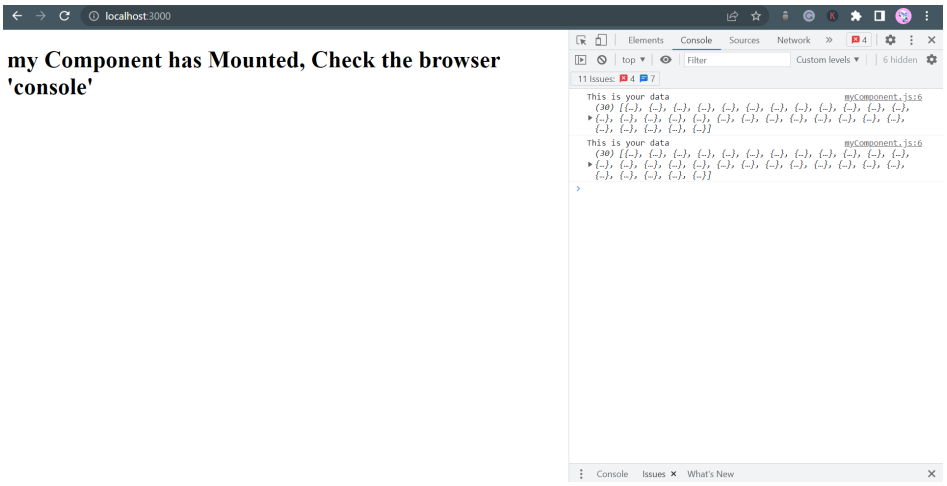
The above code retrieves data from a URL that delivers it as JSON, prints it to the console, and does so. The path to the resource you wish to fetch is typically the only argument required for the simplest version of using fetch(), which then returns a promise with the result of the fetch request. An object, that is, is the reaction.

Instead of the actual JSON, the response is merely a standard HTTP response. We would need to use the response's json() function to convert the response to actual JSON in order to extract the response's JSON body content.

```
1  import React from "react";
2
3  class ApiComponent extends React.Component {
4    componentDidMount() {
5      const apiUrl = "https://api.github.com/users/hacktivist123/repos";
6      fetch(apiUrl)
7        .then((response) => response.json())
8        .then((data) => console.log("This is your data", data));
9    }
10   render() {
11     return <h1>my Component has Mounted, Check the browser 'console' </h1>;
12   }
13 }
14 export default ApiComponent;
```

The code above creates a very straightforward class component that, once the React component has done mounting, sends a fetch request to the API URL and logs the results into the browser console.

The resource's path, which is stored in a variable called apiUrl, is sent to the retrieve() method. Following the successful completion of the retrieve request, a promise containing a response object is returned. Then, we use the json() method to extract the response's JSON body content, and finally, we log the promise's final value into the console.



A Class Component to a Function Component:

- useState and useEffect are hooks that are used to keep local states in function components.
- useEffect is a function that is used to execute functions after a component has been rendered (to "perform side effects"). useEffect can be restricted to circumstances in which a specific collection of values changes. These values are known as 'dependencies'.
- useEffects combines the functions of componentDidMount, componentDidUpdate, and componentWillUpdate.
- These two hooks essentially provide all of the conveniences formerly provided by class states and lifecycle methods.
- So, instead of a class component, let's refactor the App to a function component.

useEffect and useState

- useState is a hook that is used to keep local states in function components.
- useEffect is a function that is used to execute functions after a component has been rendered (to "perform side effects"). useEffect can be restricted to circumstances in which a specific collection of values changes. These values are known as 'dependencies'.
- useEffects combines the functions of componentDidMount, componentDidUpdate, and componentWillUpdate.
- These two hooks essentially provide all of the conveniences formerly provided by class states and lifecycle methods.
- So, instead of a class component, let's refactor the App to a function component.
- A state-based local array is managed by the useState.
- On component render, the useEffect will send a network request. When that fetch resolves, the setState function will be used to set the server response to the local state. The component will then render as a result, updating the DOM with the new data.

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

export default function ApiComponent() {
  const [data, setData] = useState([]);
  const myStyle = {
    listStyle: "none",
    marginRight: "auto",
    marginLeft: "auto",
    border: " 0.1px  solid black",
    width: 400,
    textAlign: "justify",
  };

  useEffect(() => {
    const apiUrl = "https://api.github.com/users/hacktivist123/repos";
    fetch(apiUrl)
      .then((response) => response.json())
      .then((data) => {
        setData(data);
        console.log(data);
      });
  }, []);
  return (
    <div style={{ textAlign: "center" }}>
      <h1>Available Repositories</h1>
      <ul style={myStyle}>
        {data.map((d) => (
          <li style={{ padding: 5 }}>
            {d.name} {d.id}
          </li>
        ))}
      </ul>
    </div>
  );
}

```

Available Repositories

Akintayoshedrack.me 111079785
Algorithm-Challenges 126191710
Angular-Blog 137686365
Angular-Features 137689252
awesome-app-building-tutorials 96143889
Awesome-Cloud-Foundry 355944223
awesome-cloud-native 355944822
awesome-code-review 188363277
awesome-documentation-tools 96142505
awesome-meanstack 96143920
awesome-nextjs 96140554
awesome-opensource-documents 303984891
base16-iterm2 235361951
berkshire-deck-demo 264793845
black-speakers-in-tech 160467559
Blunt 266178711
clean-code-javascript 262692557
cloudfoundry-nodejs-app 297975380
cloudfoundry-nodejs-tutorial-pt-3 304160778

b. Using Axios to Access APIs:

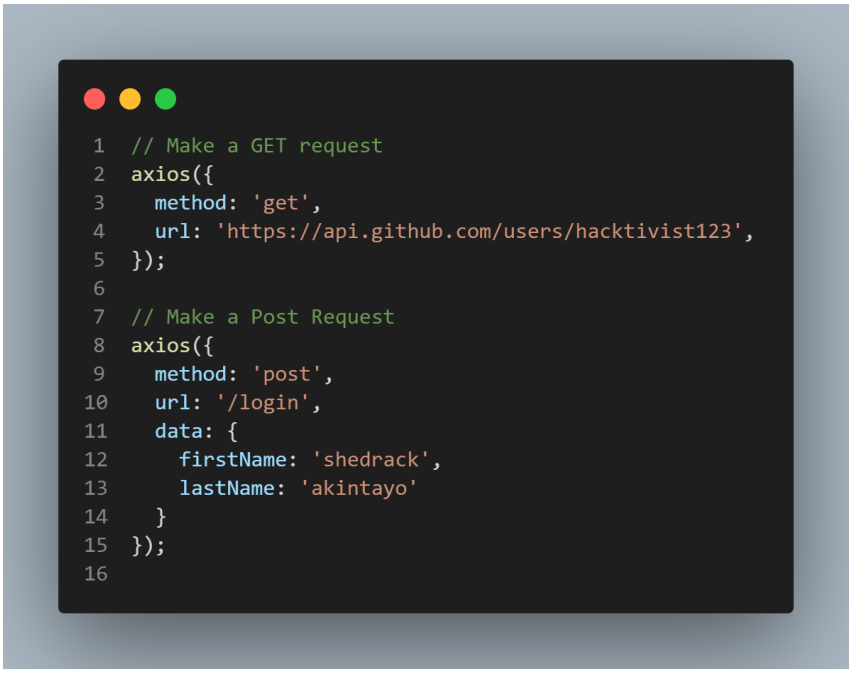
Axios is a basic promise-based HTTP client for sites, browser and nodes.js. Because Axios is promise-based, we can use async and await to write more understandable and asynchronous code. Axios provides the ability to intercept and cancel requests, as well as a built-in functionality that protects against cross-site request forgery.

AXIOS FEATURES

- Interception of requests and responses.
- streamlined handling of errors.
- defense against XSRF.
- assistance with upload progress.
- Response delay.
- being able to refuse requests.
- support for legacy browsers
- automatic transformation of JSON data.

REQUEST MAKING WITH AXIOS #

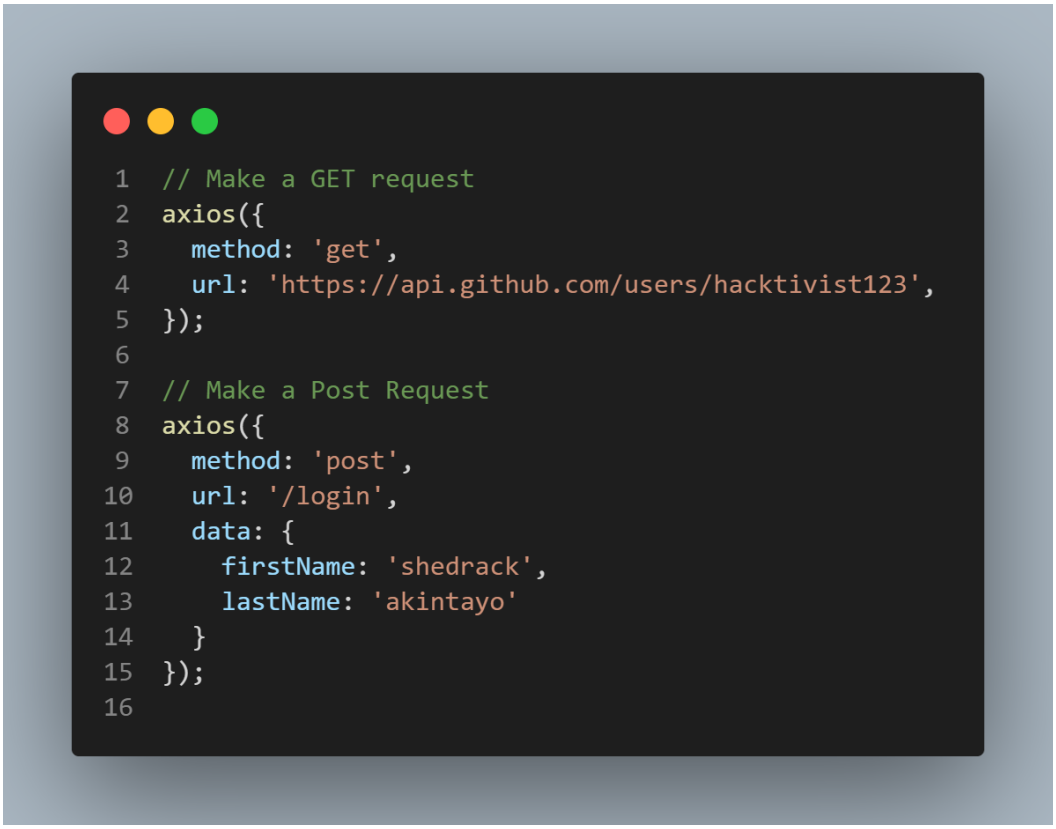
Making HTTP requests in Axios is simple. The code below demonstrates how to send an HTTP request.



```
// Make a GET request
axios({
  method: 'get',
  url: 'https://api.github.com/users/hacktivist123',
});

// Make a Post Request
axios({
  method: 'post',
  url: '/login',
  data: {
    firstName: 'shedrack',
    lastName: 'akintayo'
  }
});
```

- The code above demonstrates the fundamentals of making a GET and POST HTTP request with Axios.
- Axios also has a collection of shorthand methods for performing certain HTTP requests. The procedures are as follows:
    - axios.request(config)
    - axios.get(url[, config])
    - axios.delete(url[, config])
    - axios.head(url[, config])
    - axios.options(url[, config])
    - axios.post(url[, data[, config]])
    - axios.put(url[, data[, config]])
    - axios.patch(url[, data[, config]])

  For example, if we wish to make a request similar to the example code above but using the abbreviated methods, we can do so as follows:



```
// Make a GET request
axios({
  method: 'get',
  url: 'https://api.github.com/users/hacktivist123',
});

// Make a Post Request
axios({
  method: 'post',
  url: '/login',
  data: {
    firstName: 'shedrack',
    lastName: 'akintayo'
  }
});

```

In the code above, we make the same request as before, but this time using the abbreviated method. Axios adds flexibility and makes HTTP requests more readable.

Let's Use Axios Client to Consume A REST API

In this part, we will simply replace the fetch() method in our existing React application with Axios. All that remains is to install Axios and then use it in our App.js file to make the HTTP call to the GitHub API.

Let's now install Axios in our React project by doing one of the following:

Using NPM:

```
PS C:\Users\hp\Desktop\newproject> npm install axios
```

Using yarn:

```
PS C:\Users\hp\Desktop\newproject> yarn  axios
```

- After the installation is finished, we must import axios into our App.js. To the top of our App.js file, we'll add the following line:

```
1  import axios from "axios";
```

- After adding the piece of code to our App.js, we only need to insert the following code within our useEffect():

```
1  useEffect(() => {
2    const apiUrl = "https://api.github.com/users/hacktivist123/repos";
3    axios.get(apiUrl).then((repos) => {
4      const allRepos = repos.data;
5      setData(allRepos);
6    });
7  }, [setData]);
```

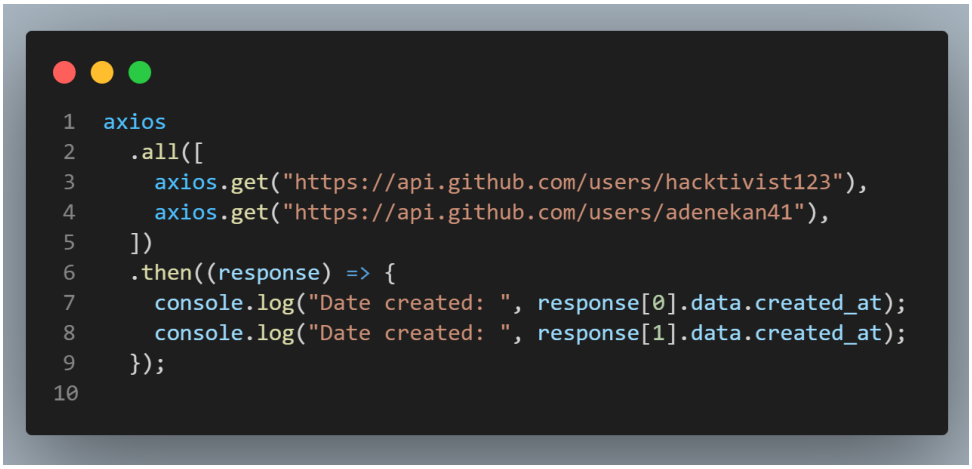
```jsx
import React from "react";
import { useEffect, useState } from "react";
import axios from "axios";
export default function ApiComponent() {
  const [data, setData] = useState([]);
  const myStyle = {
    listStyle: "none",
    marginRight: "auto",
    marginLeft: "auto",
    border: " 0.1px  solid black",
    width: 400,
    textAlign: "justify",
  };
  useEffect(() => {
    const apiUrl = "https://api.github.com/users/hacktivist123/repos";
    axios.get(apiUrl).then((repos) => {
      const allRepos = repos.data;
      setData(allRepos);
    });
  }, [setData]);
  return (
    <div style={{ textAlign: "center" }}>
      <h1>Available Repositories</h1>
      <ul style={myStyle}>
        {data.map((d) => (
          <li key={d.id} style={{ padding: 5 }}>
            {d.name}
          </li>
        ))}
      </ul>
    </div>
  );
}
```

You may have noticed that the retrieve API has been replaced by the Axios shortcut method axios. To make a get request to the API, use get.
If we performed everything right, our app should still seem the same:

3. **Difference between Axios and fetch:**
    - **Syntax Fundamental concepts:** Both Fetch and Axios have relatively basic request syntaxes. But Axios has an advantage because it automatically transforms a response to JSON, therefore when we use Axios, we bypass the step of converting the response to JSON, whereas Fetch() requires us to do so. Finally, Axios shorthand methods allow us to simplify certain HTTP Requests.
    - **Compatibility with Web Browsers:** One of the numerous reasons why developers prefer Axios to Fetch is that Axios is supported by all major browsers and versions, whereas Fetch is only supported by Chrome 42+, Firefox 39+, Edge 14+, and Safari 10.1+.
    - **Response Timeout Management:** Setting a timeout for responses is simple with Axios by using the timeout option within the request object. However, this is not so simple in Fetch. Fetch has a comparable capability that uses the AbortController() interface, but it takes longer to implement and can be confusing.
    - **HTTP Request Interception:** Axios gives developers the ability to intercept HTTP requests. When we need to alter HTTP requests from our application to the server, we need HTTP interceptors. Interceptors enable us to do so without writing additional code.

- **Making Several Requests Simultaneously:**Axios allows us to send numerous HTTP requests using the axios all() methods. With the promise, fetch() gives the same functionality. We can perform numerous fetch() queries within the all() method.

  For example, we can use the axios.all() method to make several queries to the GitHub API, as shown below:



```
axios
  .all([
    axios.get("https://api.github.com/users/hacktivist123"),
    axios.get("https://api.github.com/users/adenekan41"),
  ])
  .then((response) => {
    console.log("Date created: ", response[0].data.created_at);
    console.log("Date created: ", response[1].data.created_at);
  });

```

  The code above executes parallel queries to an array of arguments and returns the response data; in our example, it will report the created at object from each API response to the console.

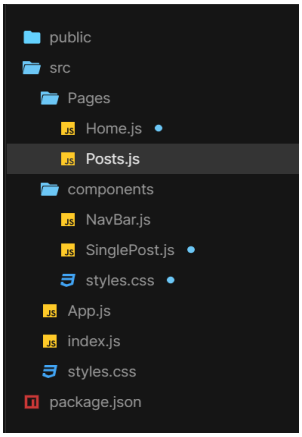
Minor Assignment

Problem Statement

Using functional components create a website that has multiple posts fetching data from API (dummyapi.io) by using Axios API and using routers for adding pages.

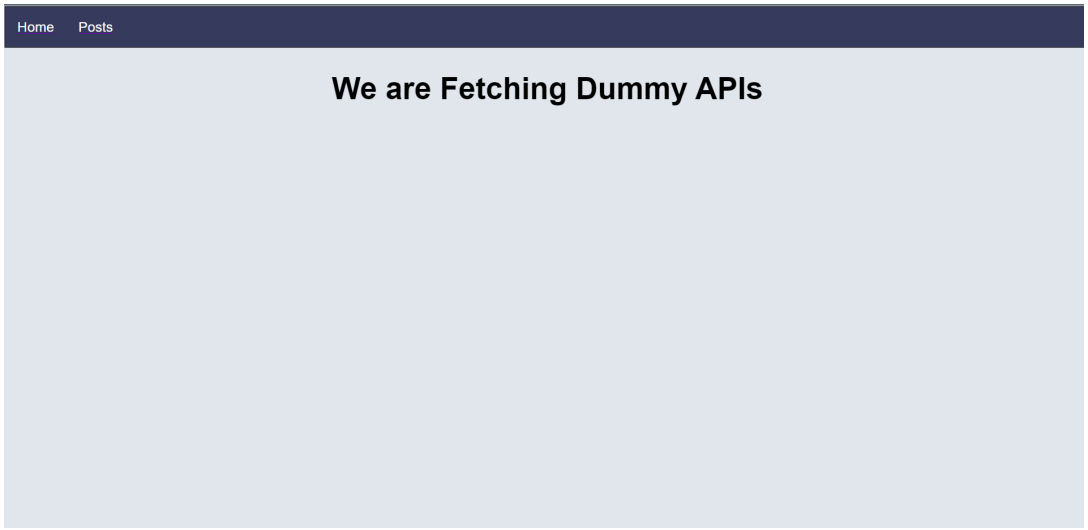
Details:

1. The folder structure of the project should be like this:

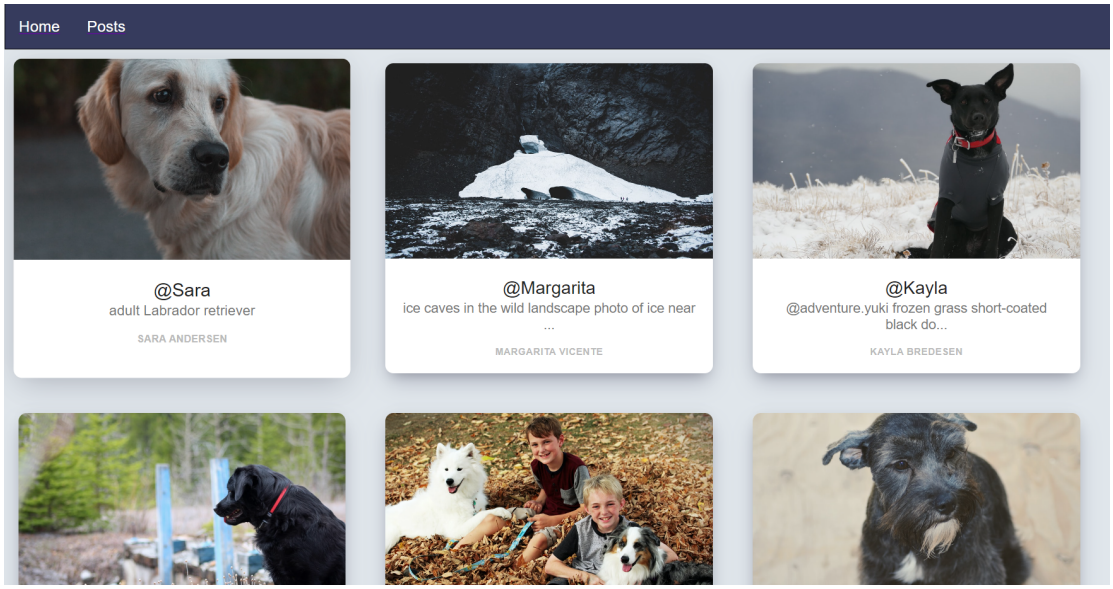


1. Create a project and add 2 folders pages and components.
2. In pages add the home and posts pages and in components, NavBar and singlePost component.
3. Add react-router-dom package in the project: **npm install react-router-dom**
4. Create routes in app.js and add a navbar in its initial path '/'.
5. Add the Axios package in the node modules: **npm install axios**
6. You need to fetch data from dummyapi.io and display it in the form of posts. And use singlePost to display multiple posts by applying mapping on the element.
   (**Hint:** use the hovercard from the first assignment for the post and make it dynamic).

   Home page:


Home
Posts
We are Fetching Dummy APIs

Posts Page:

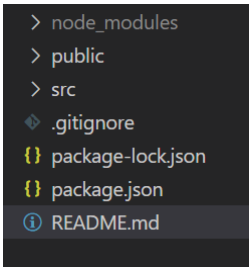

Home
Posts
@Sara
adult Labrador retriever
SARA ANDERSEN
@Margarita
ice caves in the wild landscape photo of ice near ...
MARGARITA VICENTE
@Kayla
@adventure.yuki frozen grass short-coated black do...
KAYLA BREDESEN

Process for Submission

Please upload your assignment files via this Google Form before the deadline 21st Feb 2023.

APIs Assignment Solution

Create a React App by using npx and you will see a file structure like this:



Open src
Create components directory

- Create a component named “Card.js”.
- Create an arrow function named “Card”.
- Create a .css file named “Card.css”.
- Add CSS by Applying External Css in Card.css.
- Add classNames as “cards” or “card-body”.
- Import it inside the App.js
- And render it inside the App function.
- Add hover property to the card.
  (on hover size of the card must increase or add transition of your choice).
  (Try to use camelCasing as It's a good Practice.)

Card: create a basic structure of a card using JSX and add className in each component accordingly.

```jsx
const Card = () => {
  return (
    <div className="card">
      <img
        src="https://upload.wikimedia.org/wikipedia/commons/thumb/b/b6/Image_created_with_a_mobile_phone.png/330px-Image_created_with_a_mobile_phone.png"
        alt="img"
      />
      <div className="card-body">
        <h2>Title of the Card</h2>
        <p>Caption text about the Card</p>
        <h5>Author name</h5>
      </div>
    </div>
  );
};

```

App: Import a Card component and call it in App.js

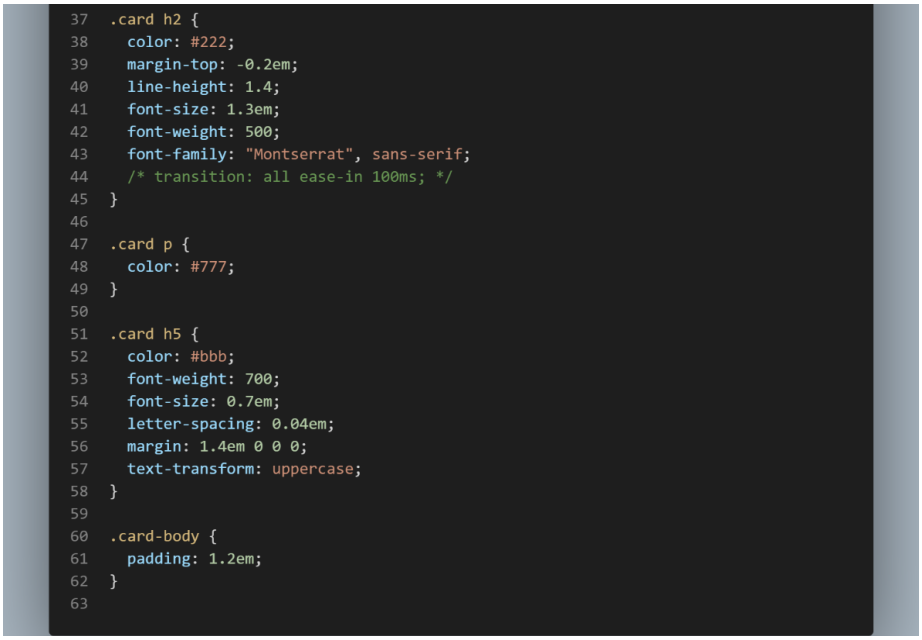
```
import "./styles.css";

export default function App() {
  return (
    <div className="App">
      <div className="header">
        <h1>React Card Component</h1>
      </div>
      <div className="cards">
        <Card />
      </div>
    </div>
  );
}
```

styles.css: Add different styling using different classNames

```
.App {
  font-family: sans-serif;
  text-align: center;
}
body {
  background: hsla(210, 20%, 90%, 1);
  padding: 2em 0;
  line-height: 1.6;
  display: flex;
  justify-content: center;
  min-height: 100vh;
  align-items: center;
  font-family: "Open Sans", sans-serif;
}
```

```
15
16  h1,
17  h2,
18  h3,
19  h5 {
20    margin: 0;
21  }
22
23  .header {
24    text-align: center;
25  }
26
27  .header h1 {
28    font-family: "Montserrat", sans-serif;
29    font-size: 3em;
30    margin-bottom: 0.2em;
31    line-height: 1.2;
32    color: #222;
33  }
```

```
1  .header h3 {
2    font-weight: 400;
3    color: #555;
4    width: 30em;
5  }
6
7  .cards {
8    display: flex;
9    align-items: flex-start;
10   justify-content: center;
11 }
12
13 .card {
14   background: #fff;
15   width: 24em;
16   border-radius: 0.6em;
17   margin: 1em;
18   overflow: hidden;
19   cursor: pointer;
20   box-shadow: 0 13px 27px -5px hsla(240, 30.1%, 28%, 0.25),
21     0 8px 16px -8px hsla(0, 0%, 0%, 0.3), 0 -6px 16px -6px hsla(0, 0%, 0%, 0.03);
22   transition: all ease 200ms;
23 }
24
25 .card:hover {
26   transform: scale(1.03);
27   box-shadow: 0 13px 40px -5px hsla(240, 30.1%, 28%, 0.12),
28     0 8px 32px -8px hsla(0, 0%, 0%, 0.14),
29     0 -6px 32px -6px hsla(0, 0%, 0%, 0.02);
30 }
31
32 .card img {
33   width: 100%;
34   object-fit: cover;
35 }
36
```



```
.card h2 {
  color: #222;
  margin-top: -0.2em;
  line-height: 1.4;
  font-size: 1.3em;
  font-weight: 500;
  font-family: "Montserrat", sans-serif;
  /* transition: all ease-in 100ms; */
}

.card p {
  color: #777;
}

.card h5 {
  color: #bbb;
  font-weight: 700;
  font-size: 0.7em;
  letter-spacing: 0.04em;
  margin: 1.4em 0 0 0;
  text-transform: uppercase;
}

.card-body {
  padding: 1.2em;
}

```

Routers and Lists

Topics covered:

- What are React routers?
    - Installing React Router
- Types of routers
- Nesting (routers inside routers)
- Lists and keys in ReactJs

1. **React routers:**

   React Router is a standard library for React routing. It allows navigating between views of different components in a React Application, changes the browser URL, and keeps the UI in sync with the URL.

   To understand how the React Router works, let's build a simple React application. The application will have three parts: the **home** component, the **about** a component, and the **contact** component. To move between these components, we will use React Router.

- **Installing React Router:**

  You may install React Router in your React application using npm. To install the Router in your React application, follow the steps below:

  Step 1: Navigate to your project's directory by using the command:

  cd project-dir-name

  Step 2: Run the following command to install the React Router:

  npm install react-router-dom

  Add react-router-dom components to your React application after installing it.

  Adding React Router Components: The major React Router components are:
  BrowserRouter is a router solution that makes use of the HTML5 history API (pushState, replaceState, and the popstate event) to keep your UI in sync with the URL. All of the other components are stored in the parent component.
  Routes: This is a new component introduced in v6 as well as an improvement to the component.
  The following are the main advantages of Routes vs Switches:
  Instead of being traversed in order,

  - relative s and s

- Routes are picked based on the best match.

Route: A route is a conditionally displayed component that displays some UI when its path matches the current URL.

Link: The link component is used to create linkages to different routes and to provide application navigation. It functions similarly to the HTML anchor tag.

To include React Router components in your application, enter your project directory in your preferred editor and navigate to the app.js file. Now, in app.js, paste the code below.

```
import {
    BrowserRouter as Router,
    Routes,
    Route,
    Link
} from 'react-router-dom';
```

Note: BrowserRouter is also known as Router.

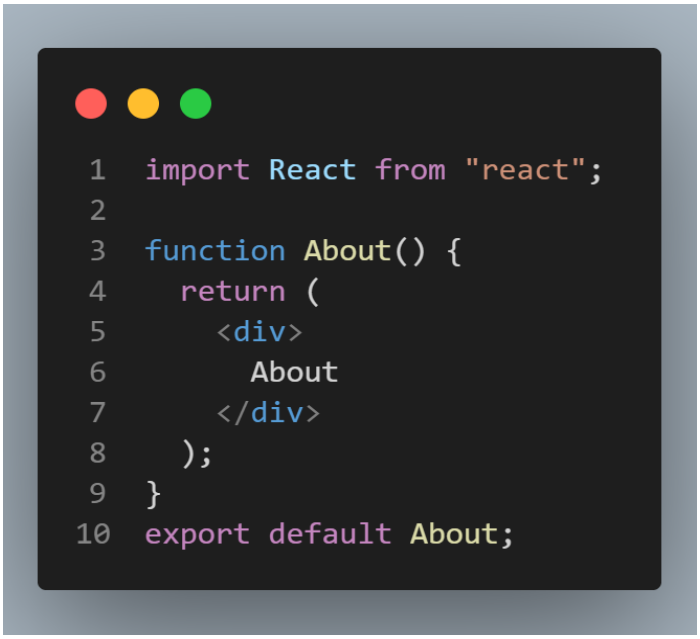
Using React Router: Before we begin using React Router, we'll need to develop a few components in our react application. Create a component folder inside the src folder in your project directory, and then add three files to the component folder: home.js, about.js, and contact.js.

Let's put some code into our three components:

Home.js:

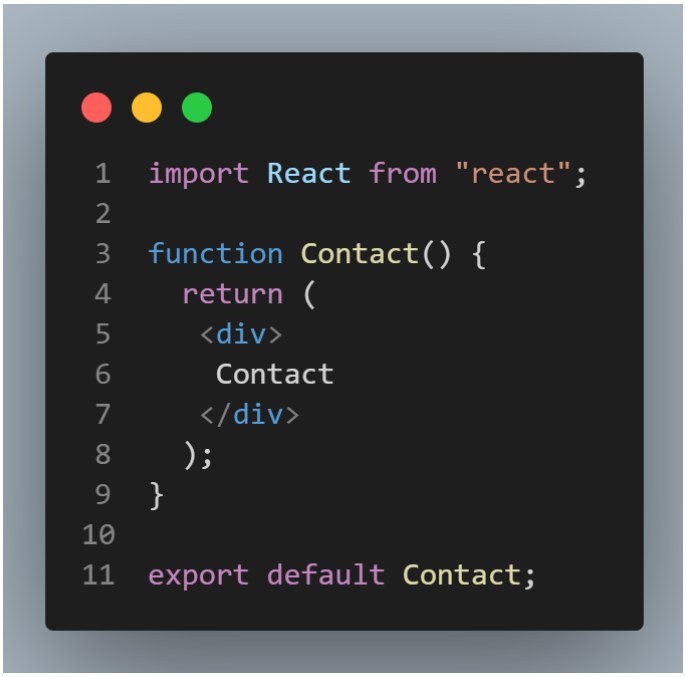
```
import React from "react";

function Home() {
  return <h1>Welcome to Home!!</h1>;
}

export default Home;
```

About.js:



```jsx
import React from "react";

function About() {
  return (
    <div>
      About
    </div>
  );
}
export default About;
```

Contact.js:



```jsx
import React from "react";

function Contact() {
  return (
   <div>
    Contact
   </div>
  );
}

export default Contact;
```

Now, let's add some React Router components to the app:

Add BrowserRouter, aliased as Router, to your appjs file to wrap all the other components.

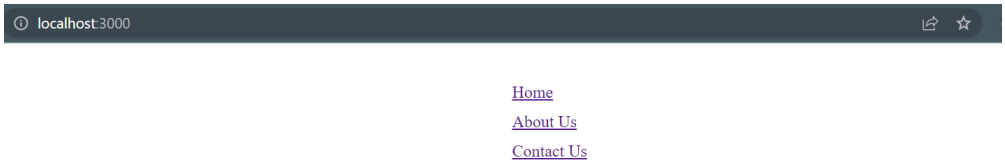
BrowserRouter is a parent component that can only have one child.

```
class App extends Component {
  render() {
    return (
      <Router>
        <div className="App"></div>
      </Router>
    );
  }
}
```

Let us now make links to our components. The to prop is used by the link component to indicate the place to which the links should navigate.

```
<div className="App">
  <ul>
    <li>
      <Link to="/">Home</Link>
    </li>
    <li>
      <Link to="/about">About Us</Link>
    </li>
    <li>
      <Link to="/contact">Contact Us</Link>
    </li>
  </ul>
</div>
```

Run your program on the local host now, then click the links you made. The url will adjust in accordance with the value entered for the Link component's properties.



You can now access the various components by clicking on the links. Your application's UI and URL are kept in sync via React Router.

2. **Types of Router:**

React Router offers three different types of routers based on the portion of the URL that the router will use to monitor the content that the user is attempting to view:

- Memory Router
- Browser Router
- Hash Router

Memory Router: This type of router retains updated URLs in memory rather than in user browsers. The user is unable to use the back and forward buttons on the browser since it saves the URL history in memory and doesn't read or write to the address bar. Your browser's URL is left unchanged. For testing and non-browser environments like React Native, it is incredibly helpful.

Syntax:

```
import { MemoryRouter as Router } from "react-router-dom";
```

Browser Router: The browser router makes advantage of the pushState, replaceState, and popState APIs from the HTML 5 history standard to keep your user interface (UI) in sync with the URL. It routes as a normal URL in the browser and expects that the server handles all request URLs (for example, /, /about) and leads to root index.html. It supports legacy browsers that do not implement HTML 5 pushState API by accepting forceRefresh props.

Syntax:

```
import { BrowserRouter as Router } from "react-router-dom";
```

Hash Router: Client-side hash routing is used by hash router. To keep your UI in sync with the URL, it uses the hash component of the URL (i.e. window.location.hash). The server won't handle the hash portion of the URL; instead, it will always send index.html in response to requests and ignore the hash value. The server does not need to be configured in order to handle routes. The purpose of it is to accommodate older browsers, which often do not implement HTML pushState API. When using legacy browsers or when there is no server logic to handle client-side processing, it is quite helpful. The react-router-dom team does not advise using this route.
Syntax:

```
import { HashRouter as Router } from "react-router-dom";
```

3. **Nesting:**
    - Routes can be rendered anywhere in the app, including in child elements, as they are standard React components.
    - Because code-splitting a React Router app is the same as code-splitting any other React project, this is useful when it comes time to divide your app into various bundles.

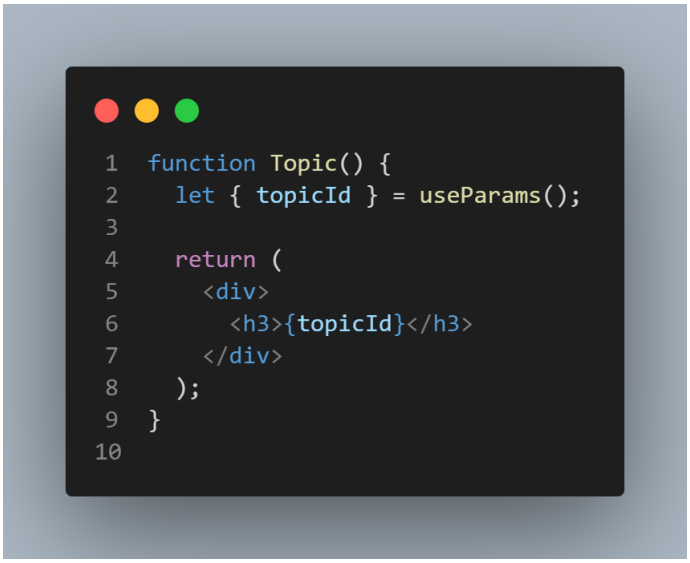
```
import React from "react";
import {
  BrowserRouter as Router,
  Switch,
  Route,
  Link,
  useParams,
  useRouteMatch,
} from "react-router-dom";

```

```jsx
export default function NestingExample() {
  return (
    <Router>
      <div>
        <ul>
          <li>
            <Link to="/">Home</Link>
          </li>
          <li>
            <Link to="/topics">Topics</Link>
          </li>
        </ul>
        <hr />
        <Switch>
          <Route exact path="/">
            <Home />
          </Route>
          <Route path="/topics">
            <Topics />
          </Route>
        </Switch>
      </div>
    </Router>
  );
}
```

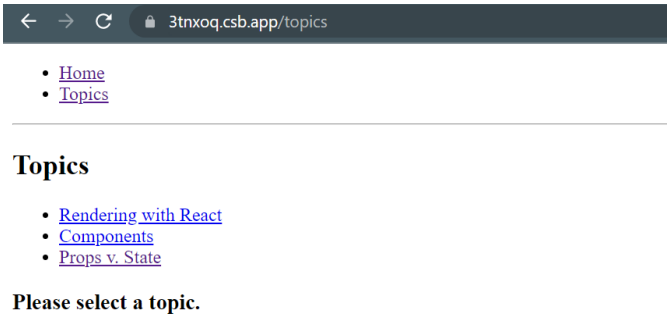
- The 'path' allows us to create relative <Route> pathways to the parent route, whereas the 'url' enables us to create relative links.

```jsx
function Topics() {
  let { path, url } = useRouteMatch();

  return (
    <div>
      <h2>Topics</h2>
      <ul>
        <li>
          <Link to={`${url}/rendering`}>Rendering with React</Link>
        </li>
        <li>
          <Link to={`${url}/components`}>Components</Link>
        </li>
        <li>
          <Link to={`${url}/props-v-state`}>Props v. State</Link>
        </li>
      </ul>

      <Switch>
        <Route exact path={path}>
          <h3>Please select a topic.</h3>
        </Route>
        <Route path={`${path}/:topicId`}>
          <Topic />
        </Route>
      </Switch>
    </div>
  );
}
```

- This component was produced by a route with the path '/topics/:topicId'. The URL's ':topicId' component denotes a placeholder that can be obtained by calling 'useParams()'.



```
1  function Topic() {
2    let { topicId } = useParams();
3
4    return (
5      <div>
6        <h3>{topicId}</h3>
7      </div>
8    );
9  }
10
```



4. **Lists and keys:**

   a. **Rendering List:**

- Data rendering from arrays

  Assume you have a content list.

```
<ul>
  <li>Creola Katherine Johnson: mathematician</li>
  <li>Mario José Molina-Pasquel Henríquez: chemist</li>
  <li>Mohammad Abdus Salam: physicist</li>
  <li>Percy Lavon Julian: chemist</li>
  <li>Subrahmanyan Chandrasekhar: astrophysicist</li>
</ul>
```

The only distinction between those list items is their content, or data. When creating interfaces, you may frequently need to display multiple instances of the same component using different data: from lists of comments to galleries of profile photographs. As these cases, you can save the data in JavaScript objects and arrays and then use techniques like map() and filter() to generate lists of components.

Here's a quick illustration of how to generate a list of objects from an array:

1.) Put the data in an array:

```
const people = [
  "Creola Katherine Johnson: mathematician",
  "Mario José Molina-Pasquel Henríquez: chemist",
  "Mohammad Abdus Salam: physicist",
  "Percy Lavon Julian: chemist",
  "Subrahmanyan Chandrasekhar: astrophysicist",
];
```

2.) Map the members of the persons array to a new array of JSX nodes, listItems:

```
const listItems = people.map((person) => <li>{person}</li>);
```

3.) Return listItems wrapped in an <ul> from your component:

```
return <ul>{listItems}</ul>;
```

4.) The end result is as follows:

```
const people = [
  'Creola Katherine Johnson: mathematician',
  'Mario José Molina-Pasquel Henríquez: chemist',
  'Mohammad Abdus Salam: physicist',
  'Percy Lavon Julian: chemist',
  'Subrahmanyan Chandrasekhar: astrophysicist'
];

export default function List() {
  const listItems = people.map(person =>
    <li>{person}</li>
  );
  return <ul>{listItems}</ul>;
}
```

- Creola Katherine Johnson: mathematician
- Mario José Molina-Pasquel Henríquez: chemist
- Mohammad Abdus Salam: physicist
- Percy Lavon Julian: chemist
- Subrahmanyan Chandrasekhar: astrophysicist

- Filtering objects in arrays

  This data can be further organized.

```
const people = [{
  id: 0,
  name: 'Creola Katherine Johnson',
  profession: 'mathematician',
}, {
  id: 1,
  name: 'Mario José Molina-Pasquel Henríquez',
  profession: 'chemist',
}, {
  id: 2,
  name: 'Mohammad Abdus Salam',
  profession: 'physicist',
}, {
  name: 'Percy Lavon Julian',
  profession: 'chemist',
}, {
  name: 'Subrahmanyan Chandrasekhar',
  profession: 'astrophysicist',
}];
```

Assume you wish to only see those whose occupation is chemist. To return only those individuals, utilize JavaScript's filter() function. This method accepts an array of items, runs them through a "test" (a function that returns true or false), and returns a new array containing only the things that passed the test (returned true).

You only want articles with the occupation chemist. The "test" function is(person) => person. profession === 'chemist'. Here's how to do it:

1. You can create a new array of people who are simply "chemists" by using person.profession == 'chemist':

```
const chemists = people.filter(person =>
  person.profession === 'chemist'
);
```

2. Map the chemists next:

```
const listItems = chemists.map(person =>
  <li>
    <img
      src={getImageUrl(person)}
      alt={person.name}
    />
    <p>
      <b>{person.name}:</b>
      {' ' + person.profession + ' '}
      known for {person.accomplishment}
    </p>
  </li>
);
```

3. Return your component's listItems as a final step:

```
return <ul>{listItems}</ul>;
```

4. The end result is as follows:

```
import { people } from './data.js';

export default function List() {
  const chemists = people.filter(person =>
    person.profession === 'chemist'
  );
  const listItems = chemists.map(person =>
    <li>
      <p>
        <b>{person.name}:</b>
        {' ' + person.profession + ' '}
        known for {person.accomplishment}
      </p>
    </li>
  );
  return <ul>{listItems}</ul>;
}
```

Mario José Molina-Pasquel Henríquez: chemist known for discovery of Arctic ozone hole

Percy Lavon Julian: chemist known for pioneering cortisone drugs, steroids and birth control pills

The code will execute when you run it in your create-react-app, but you will be informed that the list items do not have a "key" specified.

```
Warning: Each child in a list should have a unique “key” prop.
```

```
Warning: Each child in a list should have a unique "key" prop.

Check the render method of `List`. See https://reactjs.org/link/warning-keys for more information.
    at li
    at List
```

b. Key:

- React uses keys to keep track of elements. In this manner, only the modified or deleted item will be re-rendered and not the full list.
- Each sibling needs their own set of keys. They can, however, be globally duplicated.
- Typically, each item's unique ID serves as the key. The array index might be used as a key as a last resort.

Example:

```jsx
import React from "react";

export default function List() {
  const list = [
    { id: 0, name: "Rohan" },
    { id: 1, name: "Hitasha" },
    { id: 3, name: "Simran" },
    { id: 4, name: "Paarth" },
  ];
  return (
    <div>
      <h1>List of Names & IDs</h1>
      <ol style={{ listStyle: "upper-roman" }}>
        {list.map((p) => (
          <li key={p.id}>
            Name: {p.name} <br />
            Id: {p.id}
          </li>
        ))}
      </ol>
    </div>
  );
}
```

Output: