3. CSS Module

CSS Module is another way of adding styles to your application. It is a **CSS file** where all class names and **animation** names are scoped locally by default. It is available only for the component which imports it, means any styling you add can never be applied to other components without your permission, and you never need to worry about name conflicts. You can create CSS Module with the **.module.css** extension like a **myStyles.module.css** name.

Example

App.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import styles from './myStyles.module.css';

class App extends React.Component {
  render() {
    return (
      <div>
      <h1 className={styles.mystyle}>Hello JavaTpoint</h1>
      <p className={styles.parastyle}>It provides great CS tutorials.</p>
      </div>
    );
  }
}
export default App;
```

myStyles.module.css

```
.mystyle {
  background-color: #cdc0b0;
  color: Red;
  padding: 10px;
  font-family: Arial;
  text-align: center;
}

.parastyle{
  color: Green;
  font-family: Arial;
  font-size: 35px;
  text-align: center;
}
```

2. CSS Stylesheet

You can write styling in a separate file for your React application, and save the file with a .css extension. Now, you can **import** this file in your application.

Example

App.js

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

class App extends React.Component {
  render() {
    return (
      <div>
      <h1>Hello JavaTpoint</h1>
      <p>Here, you can find all CS tutorials.</p>
      </div>
    );
  }
}
export default App;
```

App.css

```
body {
  background-color: #008080;
  color: yellow;
  padding: 40px;
  font-family: Arial;
  text-align: center;
}
```

Index.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <meta name="viewport"
      content="width=device-width, initial-scale=1" />
    <title>React App</title>
  </head>
  <body>
    <div id="app"></div>
  </body>
</html>
```

Multiple Inputs

What if we add another input to the mix? Instead of just a first name field, we add a last name field as a second text input field:

```javascript
import React from "react";
function Form() {
  const [state, setState] = React.useState({
    firstName: "",
    lastName: ""
  })
  return (
    <form>
      <label>
        First name
        <input
          type="text"
          name="firstName"
          value={state.firstName}
          onChange={handleChange}
        />
      </label>
      <label>
        Last name
        <input
          type="text"
          name="lastName"
          value={state.lastName}
          onChange={handleChange}
        />
      </label>
    </form>
  );
}
```

There are a couple of significant changes that have been made, in addition to the new input field. A new `lastName` string has been added to `state` to store the data from this input, and each of the input elements have a new `name` prop. These `name` props will show up in the DOM as a `name` attributes on the input HTML elements. We'll consume them in an adjustment to the handler code:

```javascript
function handleChange(evt) {
  const value = evt.target.value;
  setState({
    ...state,
    [evt.target.name]: value
  });
}
```

In addition to getting the `value` from the event target, we get the `name` of that target as well. *This is the essential point for handling multiple input fields with one handler.* We funnel all changes through that one handler but then distinguish *which* input the change is coming from using the `name`.

This example is using `[evt.target.name]`, with the name in square brackets, to create a dynamic key name in the object. Because the form `name` props match the `state` property keys, the `firstName` input will set the `firstName` state and the `lastName` input will separately set the `lastName` state.

Also note that, because we are using a single `state` object that contains multiple properties, we're spreading (`...state`) the existing state back into the new state value, merging it manually, when calling `setState`. This is required when using `React.useState` in the solution.

React Forms

Forms are an integral part of any modern web application. It allows the users to interact with the application as well as gather information from the users. Forms can perform many tasks that depend on the nature of your business requirements and logic such as authentication of the user, adding user, searching, filtering, booking, ordering, etc. A form can contain text fields, buttons, checkbox, radio button, etc.

Creating Form

React offers a stateful, reactive approach to build a form. The component rather than the DOM usually handles the React form. In React, the form is usually implemented by using controlled components.

There are mainly two types of form input in React.

1. Uncontrolled component
2. Controlled component

Uncontrolled component

The uncontrolled input is similar to the traditional HTML form inputs. The DOM itself handles the form data. Here, the HTML elements maintain their own state that will be updated when the input value changes. To write an uncontrolled component, you need to use a ref to get form values from the DOM. In other words, there is no need to write an event handler for every state update. You can use a ref to access the input field value of the form from the DOM.

Example

In this example, the code accepts a field **username** and **company name** in an uncontrolled component.

```
import React, { Component } from 'react';
class App extends React.Component {
 constructor(props) {
    super(props);
    this.updateSubmit = this.updateSubmit.bind(this);
    this.input = React.createRef();
 }
 updateSubmit(event) {
    alert('You have entered the UserName and CompanyName successfully.');
    event.preventDefault();
 }
 render() {
  return (
    <form onSubmit={this.updateSubmit}>
      <h1>Uncontrolled Form Example</h1>
      <label>Name:
         <input type="text" ref={this.input} />
      </label>
      <label>
         CompanyName:
         <input type="text" ref={this.input} />
      </label>
      <input type="submit" value="Submit" />
    </form>
  );
 }
}
export default App;
```

Output

When you execute the above code, you will see the following screen.

After filling the data in the field, you get the message that can be seen in the below screen.