Day 10

Styling In React

@oluwakemi Oluwadahunsi

Introduction

Styling is an essential part of web development, playing an important role in how users perceive and interact with your application.

In React, there are **multiple** approaches to styling components, each offering unique advantages based on the complexity and scale of your project.

React's **component-based architecture** encourages **encapsulation**, making it crucial to adopt a styling approach that enhances maintainability, reusability, and readability.

React provides several options to handle component styling, ranging from traditional methods like **CSS stylesheets** to modern solutions like CSS-in-JS libraries and utility-first frameworks.

Each approach offers flexibility and empowers developers to create consistent, modular, and visually appealing UIs.

@oluwakemi Oluwadahunsi

Different Ways to Style React Components

There are several common approaches to styling in React, and each serves different needs depending on the project size, complexity, and developer preferences.

1. Inline Styles

You can apply styles **directly to the elements** using the **style** attribute in **JSX**. Inline styles in React are defined as objects, with CSS properties written in **camelCase**.

For Example:

@oluwakemi Oluwadahunsi

```jsx
function InlineStyleExample() {
  return (
    <div>
      <p style={{ color: "blue", fontSize: "18px", margin: "10px 0" }}>
        This is a paragraph with inline styles.
      </p>
      <button
        style={{
          backgroundColor: "green",
          color: "white",
          padding: "10px 20px",
          borderRadius: "5px",
        }}
      >
        Inline Styled Button
      </button>
    </div>
  );
}
export default InlineStyleExample;
```

styles desclared directly in jsx elements

styles desclared directly in jsx elements

Inline styling can also be defined using JavaScript variables to store the styles and then accessed in JSX. This approach can make the code cleaner and more reusable, especially when you need to apply the same styles to multiple elements.

@oluwakemi Oluwadahunsi

```jsx
function InlineStyleExample() {
  // Define the styles as JavaScript objects
  const paragraphStyle = {
    color: "blue",
    fontSize: "18px",
    margin: "10px 0",
  };
  const buttonStyle = {
    backgroundColor: "green",
    color: "white",
    padding: "10px 20px",
    borderRadius: "5px",
    border: "none",
    cursor: "pointer",
  };

  return (
    <div>
      <p style={paragraphStyle}>
        This is a paragraph with inline styles declared in a variable.
      </p>
      <button style={buttonStyle}>Inline Styled Button</button>
    </div>
  );
}
export default InlineStyleExample;
```

styles stored in “paragraphStyle” variable

styles stored in “buttonStyle” variable

variables accessed inline

Inline styles pros:

- Easy to use for simple styles.
- Directly tied to the component, reducing scope confusion.
- Avoids issues with class name conflicts.

Inline styles cons:

- Does not support pseudo-classes like :hover, :active, or media queries easily.
- Can lead to bloated and repetitive code in larger applications.
- No native support for CSS features like keyframes or animations.

@oluwakemi Oluwadahunsi

2. Regular CSS Stylesheets

The most straightforward approach to styling in React is to use regular **.css** stylesheets. You import the CSS file into your component and apply classes using the className attribute.

Example:

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

function RegularStyledComponent() {
  return (
    <div className="container">This is a regular CSS styled component</div>
  );
}
export default RegularStyledComponent;
```

css class selector, set with the attribute “className” not “class”

```css
/* styles.css */
.container {
  background-color: lightblue;
  padding: 10px;
  font-size: 18px;
}
```

container class used for setting styles to the “div” element

@oluwakemi Oluwadahunsi

Using css stylesheet pros:

- Simple and familiar to anyone with basic CSS knowledge.

- Allows the full power of CSS, including media queries, animations, and pseudo-classes.

Cons:

- Global nature of CSS can lead to class name collisions, especially in larger projects.

- Difficult to manage and maintain in large-scale applications.

@oluwakemi Oluwadahunsi

3. CSS Modules

CSS Modules are a popular solution for scoped and modular CSS in React applications. They allow styles to be **scoped locally to a component**, which means you can avoid conflicts and reuse class names across different components.

Example:

```jsx
import styles from "./Button.module.css"

function RegularStyledComponent() {
  return (
    <div className={styles.button}>This is a regular CSS styled component</div>
  );
}
export default RegularStyledComponent;
```

"styles" is used to access the button selector from the Button.module.css file

```css
/* Buttpn.module.css */
.button {
  background-color: lightblue;
  padding: 10px;
  font-size: 18px;
}
```

styles declared for the button element in the Button.module.css file

Styles are defined in an external **.css** file and imported into the component.

@oluwakemi Oluwadahunsi

Module.css pros:

- Locally scoped by default, avoiding global class name collisions.
- Easy to use and integrate with existing CSS knowledge.
- Enables modularity in larger applications.

Cons:

- Slightly more complex setup, as you need to configure Webpack or use a React framework like Create React App that supports CSS Modules out of the box.
- Not as flexible as some other approaches (e.g., Styled Components) when it comes to dynamic styling.

@oluwakemi Oluwadahunsi

4. Styled Components

Styled Components is a **CSS-in-JS** library that allows you to write actual CSS code inside your JavaScript, scoped to individual components.

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

const Button = styled.button`
  background-color: ${(props) => (props.primary ? "blue" : "gray")};
  color: white;
  padding: 10px;
  border: none;
  font-size: 18px;

  &:hover {
    background-color: ${(props) => (props.primary ? "darkblue" : "darkgray")};
  }
`;

function App() {
  return (
    <>
      <Button primary>Primary Button</Button>
      <Button>Secondary Button</Button>
    </>
  );
}
```

@oluwakemi Oluwadahunsi

In the example above:

- The **background-color: ${(props) => (props.primary ? "blue" : "gray")};** sets the background-color dynamically based on the **primary prop**. If **primary** is **true**, the background color is set to "blue"; otherwise, it is "gray".

- The background color changes to a darker shade based on the primary prop when the button is hovered.

- **Button primary** renders the Button component with the **primary prop** set to **true**. The **background-color** will be "**blue**" and change to "**darkblue**" on hover.

Styled component pros:

- Styles are scoped to the component, avoiding conflicts.
- Dynamic styling based on props and state.
- Full support for CSS features like media queries, animations, and pseudo-classes.
- Styled Components result in fewer class name conflicts, as the generated class names are unique.

Cons:

- Requires learning a new syntax and approach to styling.
- Can lead to performance concerns in large applications due to runtime generation of styles.
- Less familiar for teams used to traditional CSS.

@oluwakemi Oluwadahunsi

5. Emotion

Emotion is another **CSS-in-JS** library that is similar to Styled Components but with slightly different design goals.

Emotion provides two ways to style components: **the styled API** (similar to Styled Components) and **the css API**, which allows you to write styles inline but still benefit from CSS-in-JS features.

```jsx
/** @jsxImportSource @emotion/react */
import { css } from '@emotion/react';

function EmotionStyledComponent() {
  return (
    <div
      css={css`
        background-color: lightblue;
        padding: 10px;
        font-size: 18px;
      `}
    >
      This is an Emotion styled component
    </div>
  );
}
```

@oluwakemi Oluwadahunsi

In the example above:

- The code uses **Emotion**, a CSS-in-JS library, to apply styles directly within the React component. The **@jsxImportSource @emotion/react** directive tells React to use Emotion for processing JSX and applying styles.
- The **css** function from Emotion is used to define a set of CSS rules. These rules are written inside **template literals (` `)** and specify the styles for the component. In this example, the **background color** is set to **light blue**, padding is set to 10px, and font size is set to 18px.
- The css function's output is passed to the css prop of the element. This prop applies the defined styles to the component, resulting in the being rendered with a light blue background, 10px padding, and 18px font size.

@oluwakemi Oluwadahunsi

Using Emotion pros:

- Powerful and flexible, supports both styled components and inline styles.
- Full control over CSS with JavaScript.
- Works well with theming and dynamic styling.

Cons:

- Slightly higher learning curve.
- Like Styled Components, can have performance overhead in large apps.

@oluwakemi Oluwadahunsi

6. Preprocessors

Preprocessors like Sass and Less are tools that allow you to write more powerful and flexible CSS by providing features that regular CSS doesn't have, such as variables, nesting, mixins, and functions.

These can be used alongside React to enhance the styling capabilities of your components.

Key Features of Preprocessors in React:

1. Variables: Preprocessors let you define variables for things like colors, fonts, and sizes, making it easy to maintain consistent styles across your application.

Example (Sass):

```sass
$primary-color: #3498db; // reusable variable declaration

.button {
  background-color: $primary-color;
  padding: 10px;
}
```

2. Nesting: You can nest CSS selectors, which mirrors the structure of your HTML or JSX, leading to cleaner and more readable styles.

Example (Sass):

```sass
.container {
  .button {
    background-color: blue;
    &:hover {
      background-color: darkblue;
    }
  }
}
```

The **button class** is nested in the **container class**, while the **hover pseudo class** is nested in the **button element**.

In a traditional css file, all these classes will be written separately, but preprocessors makes codes less bloat by nesting styles that works together.

@oluwakemi Oluwadahunsi

3. Mixins and Functions: Preprocessors like Sass allow you to create reusable pieces of code (mixins) or functions for common styling patterns.

Example (Sass):

```sass
@mixin button-style($color) {
  background-color: $color;
  padding: 10px;
  border: none;
}

.primary-button {
  @include button-style(blue);
}

.secondary-button {
  @include button-style(gray);
}
```

The **button-style** created using **@mixin**, the **background-color** property can be **reused** and styled differently.

@oluwakemi Oluwadahunsi

Preprocessors pros:

- Preprocessors like Sass allow for variables, nesting, mixins, and other advanced features, making styles more maintainable.
- Works well with both CSS Modules and traditional stylesheets.

Cons:

- Slightly more complex setup as it requires a preprocessor.
- Global styles by default, which can lead to class name conflicts unless combined with CSS Modules.

@oluwakemi Oluwadahunsi

7. CSS Frameworks

When styling React applications, CSS frameworks and libraries are often used to speed up development and maintain consistent design patterns.

These tools provide pre-built styles, components, and utilities, allowing developers to focus more on functionality and less on writing repetitive styles.

CSS frameworks like **Bootstrap**, **Tailwind CSS**, and **Bulma** provide predefined styles and layout structures to quickly build responsive, aesthetically pleasing applications.

There are several CSS frameworks compatible with React, such as Bootstrap, Tailwind CSS, Semantic UI React, Bulma, Foundation, Ant Design, Tachyon, etc...

@oluwakemi Oluwadahunsi

Tailwind CSS

Tailwind CSS is a **utility-first CSS framework** that allows us to build custom designs without leaving their HTML or JSX files.

In contrast to traditional CSS frameworks, which provide pre-designed components, Tailwind focuses on providing **low-level utility classes** to style elements directly within your markup.

I see that a lot of people usually have issues setting up Tailwind CSS for their React project, so we will go through each installation process step by step.

Step 1: Install Tailwind CSS using npm:

Open a terminal in your project and type “npm install -D tailwindcss postcss autoprefixer” like this:

```
kemil@DESKTOP-QKVDT62 MINGW64 /c/Desktop/My-Projects/my-vite-app
$ npm install -D tailwindcss postcss autoprefixer
```

@oluwakemi Oluwadahunsi

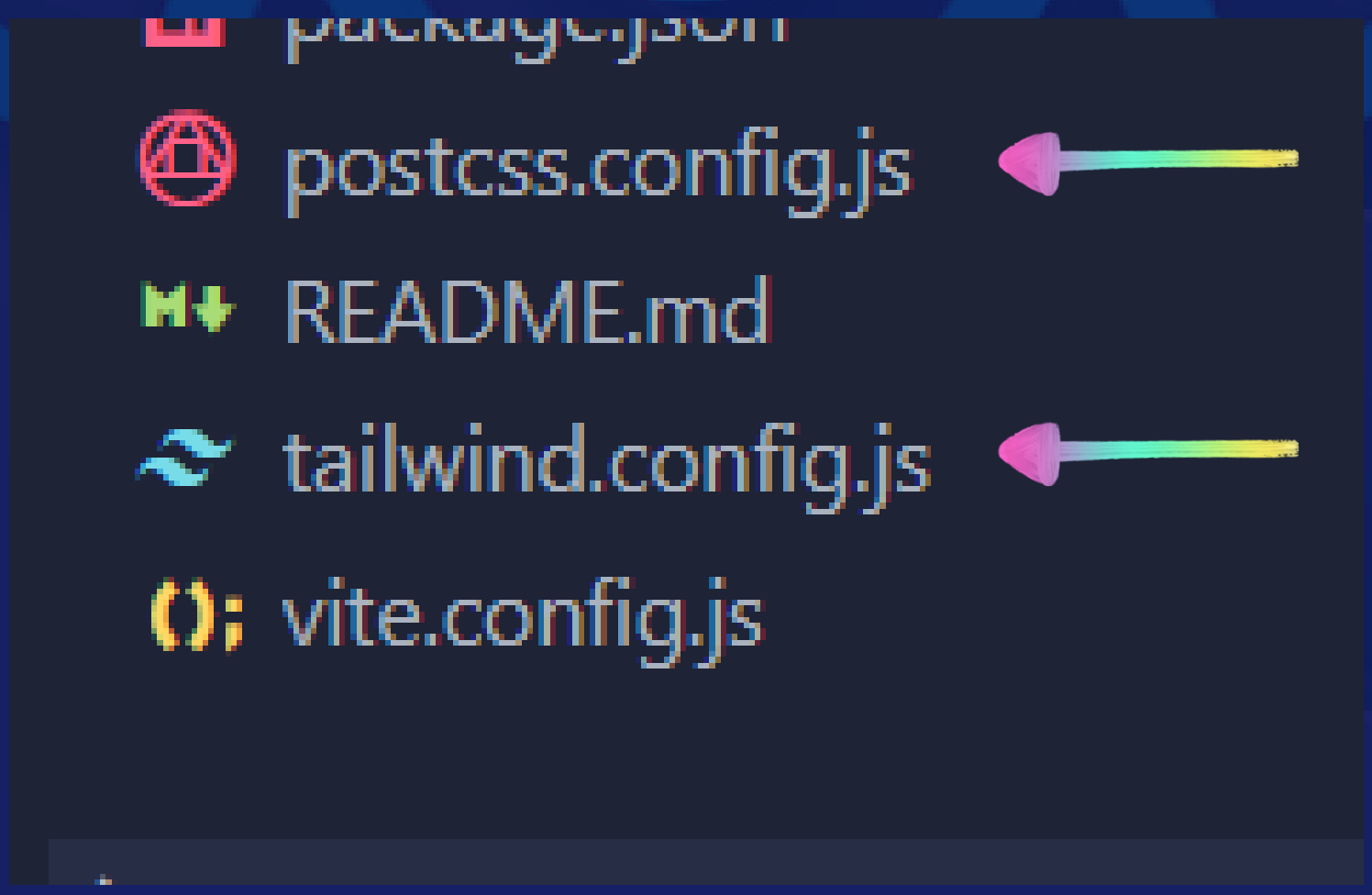
Step 2: Initialize Tailwind CSS:

This initializes tailwind in your project by creating the necessary files like the tailwind.config.js file.

After successful installation, type this in your terminal “npx tailwindcss init -p”

```
kemil@DESKTOP-QKVDT62 MINGW64 /c/Desktop/My-Projects/my-vite-app
$ npx tailwindcss init -p
```

These 2 files will be created in your project’s root folder:

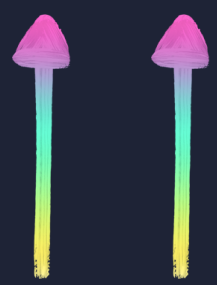


Step3: Declare your file types

Declare the types of files extensions you want tailwind to recognize, make sure to include all the file types that you will be using tailwind in.

I have one I use always, just like a base code for me. Add this line of code in the content variable in your **tailwind.config.js** file:
"./index.html", "./src//*.{js,jsx,ts,tsx}"**

```
export default {
  content: ["./index.html", "./src/**/*.{js,jsx,ts,tsx}"],
  theme: {
    extend: {},
  },
  plugins: [],
};
```



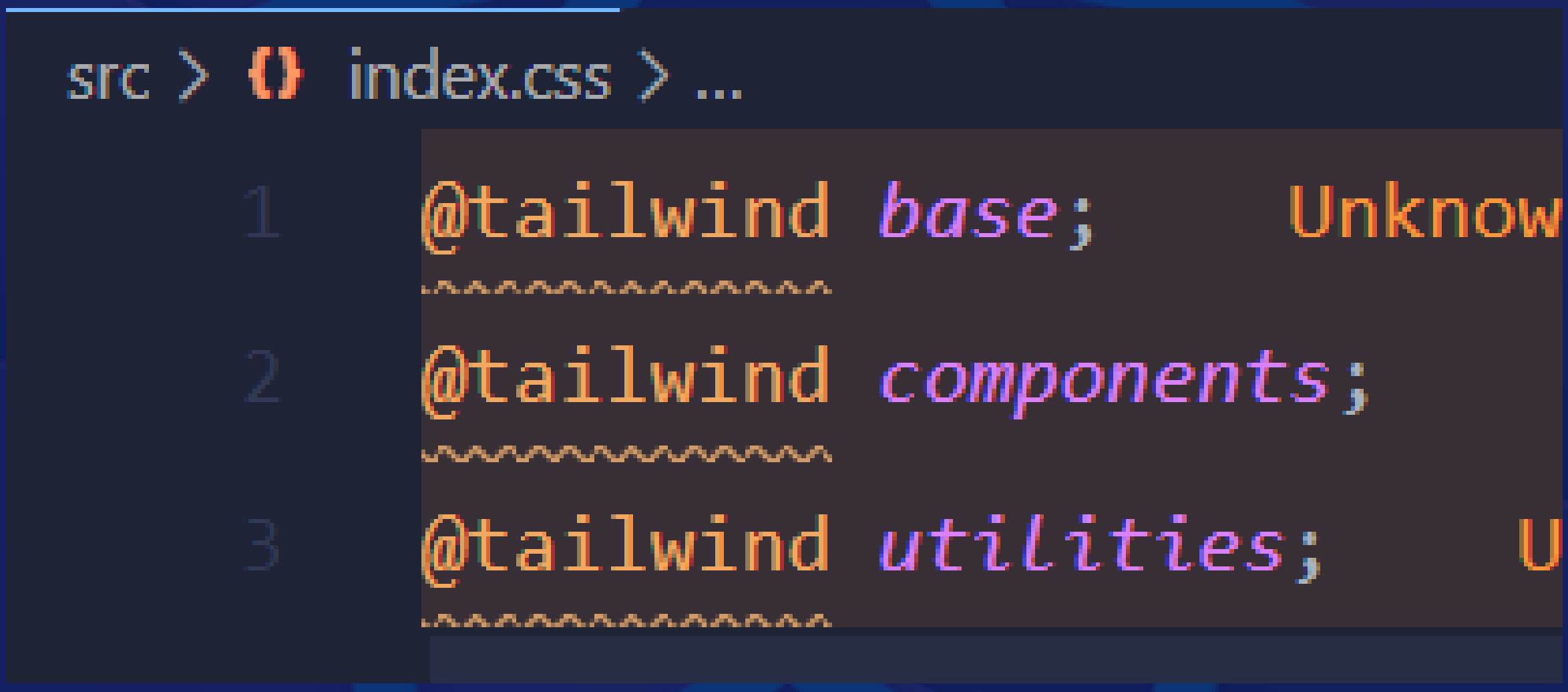
write it here, inside the angle brackets

@oluwakemi Oluwadahunsi

Step 4: Add tailwind styles:

Add Tailwind’s base, components, and utilities styles in the **src/index.css**:

```
@tailwind base;
@tailwind components;
@tailwind utilities;
```



You will get these underlines exactly like in he image, but don’t worry about it, it’s always been like that.

I don’t know why it’s always like that though, if you have an explanation for that, we would love to know in the comments!

@oluwakemi Oluwadahunsi

Step 5: Testing

We are now done with all basic setup, now let's test if it's working.

```jsx
function TailwindTesting() {
  return (
    <div className="p-4 flex flex-col items-center justify-center">
      <p className="text-[2rem] text-white mb-4">
        This is a paragraph styled with Tailwind CSS.
      </p>
      <button
        className="bg-pink-600 text-white px-4
      py-2 rounded hover:[#5a0834]"
      >
        Click Me
      </button>
    </div>
  );
}
export default TailwindTesting;
```

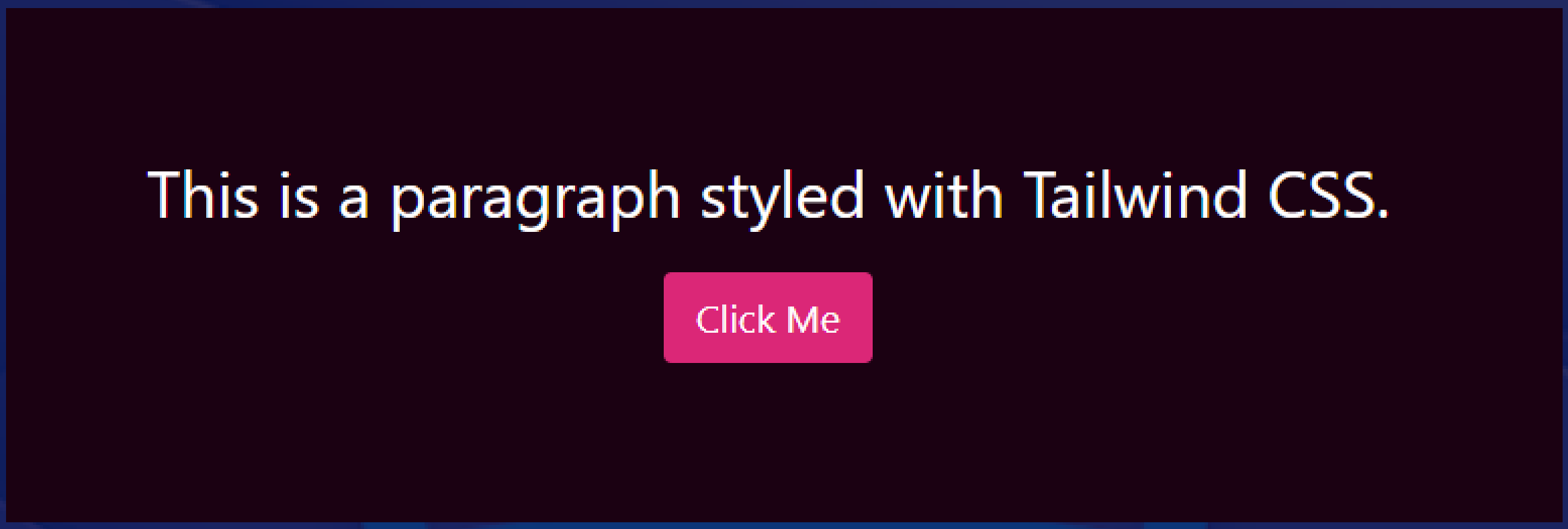
You can use basic css units in tailwind by wrapping the values in square brackets

tailwind color utility classes

using hex codes wrapped in angle brackets

@oluwakemi Oluwadahunsi

You should get this in your browser:



Tailwind CSS pros:

- **Utility-First Approach:** Tailwind promotes reusability and avoids repetition by using small utility classes that are easy to combine.
- **Design Consistency:** Provides a large set of utility classes, ensuring consistent styling across components.
- **Customization:** You can configure Tailwind to suit your needs through configuration files for themes, colors, and other design properties.

Cons:

- **JSX Clutter:** JSX can become cluttered due to the large number of classes applied to each element.

@oluwakemi Oluwadahunsi

Documentations

Documentations for further reading and research on some common CSS frameworks and libraries:

- Tailwind CSS - v2.tailwindcss.com/docs
- Bootstrap - getbootstrap.com/docs/4.1
- Foundation - get.foundation/frameworks-docs.html
- Material UI - mui.com/material-ui
- Ant Design - ant.design/docs/react/introduce
- Bulma - bulma.io/documentation/
- Styled components-styled-components.com/docs
- CSS modules - github.com/css-modules/css-modules

@oluwakemi Oluwadahunsi

I hope you found this material useful and helpful.

Remember to:

Like

Save for future reference

&

Share with your network, be helpful to someone 👌

@oluwakemi Oluwadahunsi

Hi There!

Thank you for reading through

Did you enjoy this knowledge?

💼 Follow my LinkedIn page for more work-life balancing and Coding tips.

🌐 LinkedIn: Oluwakemi Oluwadahunsi

kodemaven-portfolio.vercel.app