Just for fun! Explore styled-components.

Step 1: Install dependencies

If you haven't already installed Styled-Components, install the library in your project:

```
npm install styled-components
```

Step 2: Create a React component with styled-components

The code:

- `styled.div` creates a styled `div` element.
- `styled.button` creates a styled `button` element.
- `styled.h1` creates a styled `h1` element.
- We use CSS-in-JS syntax directly inside the `styled-components` to define the styles for each component.
- In the `Button`, we use the `&:hover` selector to define the hover state.

```
// App.js
import React from 'react';
import styled from 'styled-components';

// Define styled components
const Container = styled.div`
  display: flex;
  justify-content: center;
  align-items: center;
  height: 100vh;
  background-color: #f0f0f0;
`;

const Button = styled.button`
  background-color: #3498db;
  color: white;
  padding: 10px 20px;
  border: none;
  border-radius: 5px;
  font-size: 16px;
  cursor: pointer;
```

```
  &:hover {
    background-color: #2980b9;
  }
`;

const Heading = styled.h1`
  font-size: 36px;
  color: #2c3e50;
`;

// App component
function App() {
  return (
    <Container>
      <div>
        <Heading>Welcome to React with Styled Components</
Heading>
        <Button onClick={() => alert('Button clicked!')}
>Click Me</Button>
      </div>
    </Container>
  );
}

export default App;
```

Step 3: Run the app

Run your app with `npm run dev`

Look at your React app styled with Styled-Components!

Styled-Components enable **CSS-in-JS**—writing CSS directly inside your JavaScript files. In React, components and styles can be tightly coupled.

Benefits of using Styled-Components:

1. Scoped Styles

- **Styled-Components** allows you to scope your styles to individual components, meaning the styles you define are automatically applied only to that specific component, preventing conflicts between styles across the application.

2. Dynamic Styling

- You can pass props to styled components and use those props to dynamically change the styles. This is particularly useful for building reusable components that adapt to different contexts (e.g., color changes based on props).

Example with `props`:

```
const Button = styled.button`
  background-color: ${props => props.primary ? 'blue' :
'gray'};
  color: white;
  padding: 10px 20px;
  border: none;
  border-radius: 5px;
`;

<Button primary={true}>Primary Button</Button>
<Button primary={false}>Secondary Button</Button>
```

3. No CSS Class Name Collisions

- Since styles are defined within JavaScript, the class names for elements are automatically generated with unique identifiers to avoid any clashes between global CSS rules. You don't have to worry about naming collisions or dealing with specific class selectors.

4. Easier Maintenance

- By keeping styles with the components they belong to, Styled-Components makes it easier to maintain and scale your application. The styling for each component is self-contained, and you don't have to jump between different CSS files.

5. Theming Support

- Styled-Components support *theming*, which allows you to define a set of consistent colors, fonts, and spacing for your entire app. You can easily switch between themes (e.g., light/dark mode) by changing theme values in one place.

Theming:

```
// Define theme
const theme = {
  primaryColor: 'blue',
  secondaryColor: 'gray',
};

// Apply theme using ThemeProvider
import { ThemeProvider } from 'styled-components';

function App() {
  return (
    <ThemeProvider theme={theme}>
      <Button primary={true}>Primary Button</Button>
    </ThemeProvider>
  );
}
```

6. Performance Optimization

- Styled-Components optimizes performance by only injecting the styles that are actually used in the page. It also supports server-side rendering, which helps with faster page loads and better SEO.

7. No Need for External Stylesheets

- Since styles are part of the JavaScript file, there's no need to manage external CSS files. This reduces dependencies and keeps everything in one place, simplifying the overall project structure.

8. Powerful Features

- **Styled-Components** comes with a variety of advanced features, such as:
  - **Nesting**: You can nest your styles like you would in SCSS.
  - **Mixins**: Reusable sets of styles.
  - **Media Queries**: Dynamically add responsive design within your components.

```
}
```

Add a theme:

Step 1: Define the Theme

First, create a theme object that contains the values you want to apply globally, like colors, fonts, etc.

```
// src/theme.js

export const theme = {
  colors: {
    primary: '#3498db',
    secondary: '#2c3e50',
    background: '#f0f0f0',
    button: '#2980b9',
    buttonHover: '#1d6fa5',
  },
  fonts: {
    main: 'Arial, sans-serif',
    heading: 'Georgia, serif',
  },
};
```

Step 2: Use `ThemeProvider`

Next, import `ThemeProvider` from **styled-components** and wrap your entire application with it. This makes the theme available to all styled components within your app.

```
// src/App.jsx
import React from 'react';
import styled, { ThemeProvider } from 'styled-components';
import { theme } from './theme'; // Import the theme

// Define styled components
const Container = styled.div`
  display: flex;
  justify-content: center;
  align-items: center;
  height: 100vh;
```

```
  background-color: ${(props) =>
props.theme.colors.background};
`;

const Button = styled.button`
  background-color: ${(props) =>
props.theme.colors.primary};
  color: white;
  padding: 10px 20px;
  border: none;
  border-radius: 5px;
  font-size: 16px;
  cursor: pointer;

  &:hover {
    background-color: ${(props) =>
props.theme.colors.buttonHover};
  }
`;

const Heading = styled.h1`
  font-size: 36px;
  color: ${(props) => props.theme.colors.secondary};
  font-family: ${(props) => props.theme.fonts.heading};
`;

function App() {
  return (
    <ThemeProvider theme={theme}>
      <Container>
        <div>
          <Heading>Welcome to Vite + React + Styled
Components</Heading>
          <Button onClick={() => alert('Button clicked!')}
>Click Me</Button>
        </div>
      </Container>
    </ThemeProvider>
  );
}
```

```
export default App;
```

Step 3: Run Your App

Start the Vite development server again to see the theme applied:

```
npm run dev
```

Now, your application is using the theme values globally. For example, you can change the primary button color or the background color of the entire app by modifying the `theme.js` file, and the changes will automatically be reflected in all components that use the theme.

Benefits of Using `ThemeProvider`:

1. **Centralized Management**: All styles dependent on the theme (like colors, fonts, spacings, etc.) are managed in one place, making it easy to update.
2. **Global Access**: With `ThemeProvider`, all the styled components can access the theme without needing to pass props down manually, which keeps your code cleaner.
3. **Flexibility**: You can easily switch themes or modify styles dynamically by changing the theme object.

Next: Switch Themes Dynamically

If you want to switch between themes dynamically (e.g., toggle between light and dark mode), you can modify the theme using state and update it at runtime. Here's a basic way to switch between two themes:

```
// src/App.jsx
import React, { useState } from 'react';
import styled, { ThemeProvider } from 'styled-components';
import { theme, darkTheme } from './theme'; // Assuming
darkTheme is defined

// Define styled components (same as before)
const Container = styled.div`
  display: flex;
  justify-content: center;
  align-items: center;
  height: 100vh;
  background-color: ${(props) =>
props.theme.colors.background};
```

```
`;

const Button = styled.button`
  background-color: ${(props) =>
props.theme.colors.primary};
  color: white;
  padding: 10px 20px;
  border: none;
  border-radius: 5px;
  font-size: 16px;
  cursor: pointer;

  &:hover {
    background-color: ${(props) =>
props.theme.colors.buttonHover};
  }
`;

const Heading = styled.h1`
  font-size: 36px;
  color: ${(props) => props.theme.colors.secondary};
  font-family: ${(props) => props.theme.fonts.heading};
`;

function App() {
  const [isDarkMode, setIsDarkMode] = useState(false);

  return (
    <ThemeProvider theme={isDarkMode ? darkTheme : theme}>
      <Container>
        <div>
          <Heading>Welcome to Vite + React + Styled
Components</Heading>
          <Button onClick={() => setIsDarkMode(!
isDarkMode)}>
            Switch Theme
          </Button>
        </div>
      </Container>
    </ThemeProvider>
```

```
  );
}

export default App;
```

Add `darkTheme` to `theme.js`:

```
//add to src/theme.js
export const darkTheme = {
  colors: {
    primary: '#2c3e50',
    secondary: '#ecf0f1',
    background: '#34495e',
    button: '#2980b9',
    buttonHover: '#1d6fa5',
  },
  fonts: {
    main: 'Arial, sans-serif',
    heading: 'Georgia, serif',
  },
};
```

The button toggles between the light and dark theme based on the state.