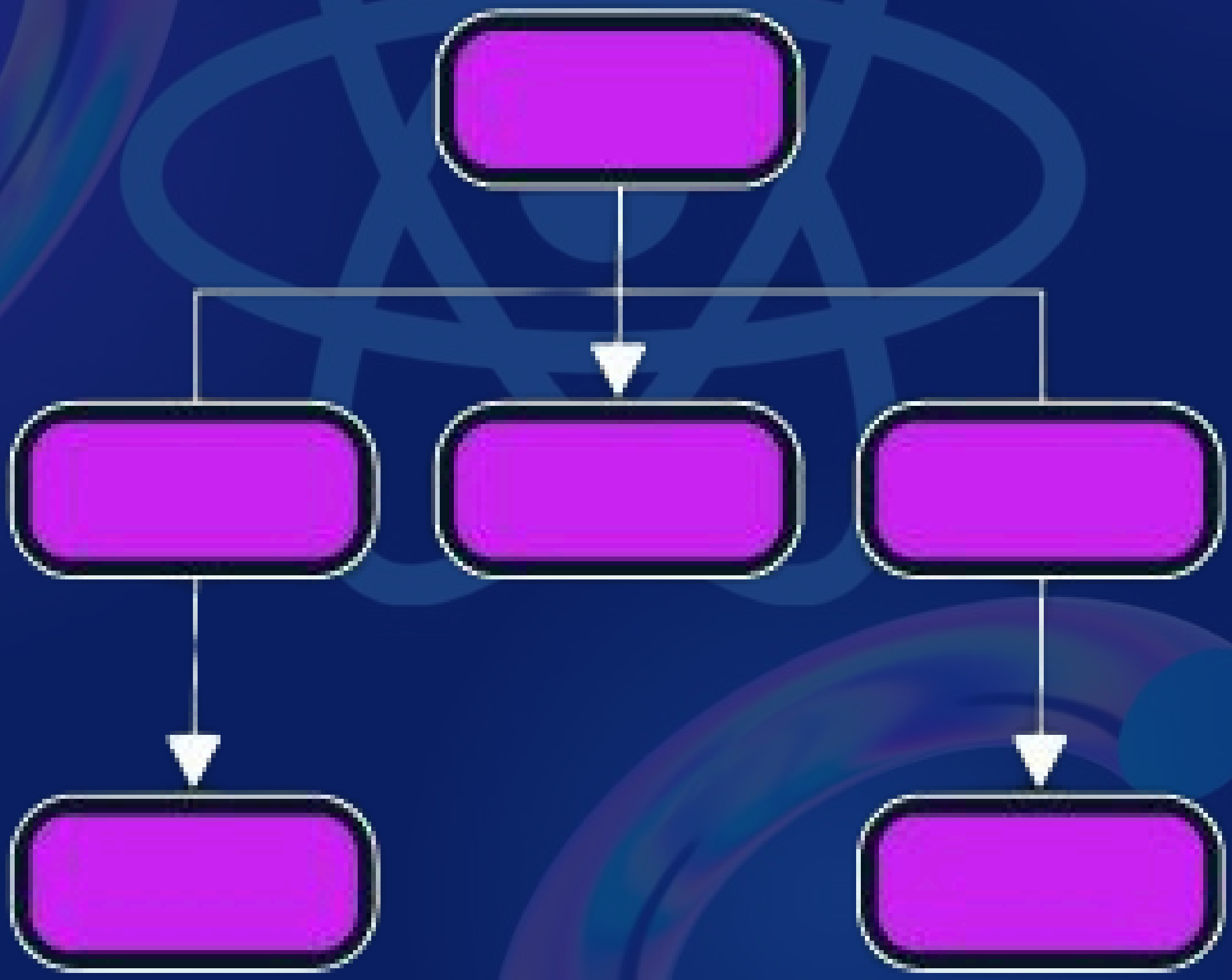
Day 19

Global State Management In React (Part 1)



@oluwakemi Oluwadahunsi

Introduction

Global state management in React refers to managing the state that is shared across multiple components in an application.

In simpler terms, it allows various parts of an app to access and modify the same data, no matter how deeply nested the components are within the component tree.

In React, each component can have its own local state—this is state that is contained within a single component. However, in larger applications, you'll often need to share data between multiple components that are not directly related or are spread out across the app.

This is where global state comes into play.

@oluwakemi Oluwadahunsi

Why is Global State Management Important?

- Avoids Prop Drilling:

Prop drilling refers to the process of passing data down through multiple layers of components just to get it to the one that needs it. This can make your code complex and hard to maintain.

Global state management solves this problem by allowing any component to access the global state directly, without needing to pass props through intermediate components.

- Centralized State:

By having a centralized place for your application's state, you can better manage and organize your data, making it easier to debug and maintain.

- Consistency Across the Application:

When multiple components share the same data, global state management ensures that these components stay in sync.

@oluwakemi Oluwadahunsi

Examples of When You Need Global State

- Authentication: Managing the logged-in user’s data and session details that are needed across different pages.
- Theme Management: A user can switch between light and dark themes, and this preference needs to be applied across the entire application.
- Cart Management in E-commerce: You need to track a user's shopping cart across various pages like product listings and checkout.
- Language/Localization Settings: Managing the selected language or region for translation across the app.

How Global State Works in React (Using Context API)

The **React Context API** is a feature that helps to manage state globally across your application without having to pass props down manually at every level.

It's a way to share data between components without having to explicitly pass **props** at each **parent-child** level.

This makes state management simpler, especially when dealing with complex component trees or deeply nested components.

@oluwakemi Oluwadahunsi

The Problem Context API Solves

In a typical React app, data flows from **parent** components to **child** components via **props**.

However, in applications where multiple components need access to the same data, passing **props** through every intermediate component (that doesn’t necessarily need that data) can be tedious and lead to "**prop drilling.**"

Prop drilling refers to passing data down from **parent components** to **deeply nested child components**, even when some of the intermediate components do not require this data.

This can make the code harder to maintain and scale, as components that don’t need the data still have to handle it.

@oluwakemi Oluwadahunsi

Example of Prop Drilling:

```jsx
const Grandparent = () => {
  const userName = "John Doe";
  return <Parent userName={userName} />;
};

const Parent = ({ userName }) => {
  return <Child userName={userName} />;
};

const Child = ({ userName }) => {
  return <p>Hello, {userName}!</p>;
};
```

In this example, the **Parent** component is only passing the **userName** prop down to the **Child** component, but doesn't need it itself. This can become cumbersome as the application grows.

@oluwakemi Oluwadahunsi

When to Use the Context API

- When you need to manage global state across multiple components.
- When several nested components need access to the same data.
- When you are avoiding prop drilling like passing the same props through many component levels just to reach a child component.

Examples of use cases:

- **Authentication** (sharing the logged-in user's data)
- Theme management (light/dark modes)
- Language or localization settings
- Cart management in e-commerce apps

@oluwakemi Oluwadahunsi

Setting Up Global State with the Context API

Step by step example on how to set up context API in a React application:

Step 1: Setting Up the App

First, create a new React app using Vite (or Create React App).

Run these commands in your terminal to create a project:

```
npm create vite@latest my-app --template react
cd my-app
npm install
npm run dev
```

This sets up your basic React app.

@oluwakemi Oluwadahunsi

Step 2: Creating the Theme Context

Next, we'll create a **ThemeContext** that will hold the **current theme** and a **function** to toggle between the two modes.

Create a new file called **ThemeContext.js** in the **src** folder and write these codes:

```
import { createContext, useState } from 'react';

// Create a context with default value of 'light'
export const ThemeContext = createContext();

export const ThemeProvider = ({ children }) => {
  const [theme, setTheme] = useState('light'); // Global state

  const toggleTheme = () => {
    setTheme((prevTheme) => (prevTheme === 'light' ? 'dark' : 'light'));
  };

  return (
    <ThemeContext.Provider value={{ theme, toggleTheme }}>
      {children}
    </ThemeContext.Provider>
  );
};
```

context created

default state

function that toggles the theme state

all components that the Provider wraps around, the theme is made available to them

- **ThemeContext:** This is our context created using **createContext()**. It will hold the current theme and the toggle function.

- **ThemeProvider:** This component wraps around the components that need access to the theme state. It provides the **current theme** and **toggleTheme function** to all its **children (every component that the Provider wraps around)**.

Step 3: Using the Theme Context in the App

Now, let's update our **App.js** file to use the **ThemeContext** and display the current theme. We'll also add a **button** to toggle the theme.

I used Vite method to create my React app, so I will wrap **<App />** in my **main.js** file with the Provider

```
createRoot(document.getElementById("root")).render(
  <StrictMode>
    <ThemeProvider>
      <App />
    </ThemeProvider>
  </StrictMode>
);
```

wrap your app with the exported ThemeProvider from Themecontext.js

You can also wrap your entire application in your App.js file like this:

```jsx
function App() {
  return (
    <>
      <ThemeProvider>
        <ReactStyle />
      </ThemeProvider>
    </>
  );
}

export default App;
```

It will also work the same way like the first method. I only prefer the first method because it makes my code looks neat, a lot usually go on in the app.js file already.

@oluwakemi Oluwadahunsi

Create a new file called **ThemeSwitcher.js** to use the **ThemeContext** and add these codes:

```
import { useContext } from 'react';
import { ThemeContext } from './ThemeContext';

const ThemeSwitcher = () => {
  const { theme, toggleTheme } = useContext(ThemeContext); // Access the global state

  return (
    <div
      style={{
        backgroundColor: theme === 'light' ? '#fff' : '#333',
        color: theme === 'light' ? '#000' : '#fff',
        padding: '20px',
        textAlign: 'center',
      }}
    >
      <p>Current Theme: {theme}</p>
      <button onClick={toggleTheme}>
        Switch to {theme === 'light' ? 'dark' : 'light'} mode
      </button>
    </div>
  );
};

export default ThemeSwitcher;
```

@oluwakemi Oluwadahunsi

In this component:

- **useContext:** The useContext hook is used to access the **theme** and **toggleTheme** function exported as '**values**' from the **ThemeContext**.

- **Dynamic Styling:** We change the background color and text color based on the current theme.

- **Button Logic:** The button toggles between **light** and **dark** modes by calling the **toggleTheme** function.

Now update your **App.js** file to include the **ThemeSwitcher** component:

```
function App() {
  return (
    <>
      <ReactStyle />
      <ThemeSwitcher />
    </>
  );
}

export default App;
```

Let’s add more components to demonstrate how the global state is shared across the app.

Create a **Header.jsx** component:

```jsx
import { useContext } from 'react';
import { ThemeContext } from './ThemeContext';

const Header = () => {
  const { theme } = useContext(ThemeContext); // Access the theme globally

  return (
    <header
      style={{
        backgroundColor: theme === 'light' ? '#f1f1f1' : '#222',
        padding: '10px',
        textAlign: 'center',
      }}
    >
      <h1>{theme === 'light' ? 'Light Mode' : 'Dark Mode'} Header</h1>
    </header>
  );
};

export default Header;
```

Now, include the Header in the **App.js** file:

```jsx
function App() {
  return (
    <>
      <Header />
      <ReactStyle />
      <ThemeSwitcher />
```

In this example, we have Multiple Components sharing the Same State. Both the **Header** and **ThemeSwitcher** components can access the theme state without any **prop drilling**.

This demonstrates how the **Context API** enables global state sharing across different parts of the app.

Step 5: Adding Some Basic Styling

We can add some simple CSS to make the app look better. In the **index.css** or **App.css** file:

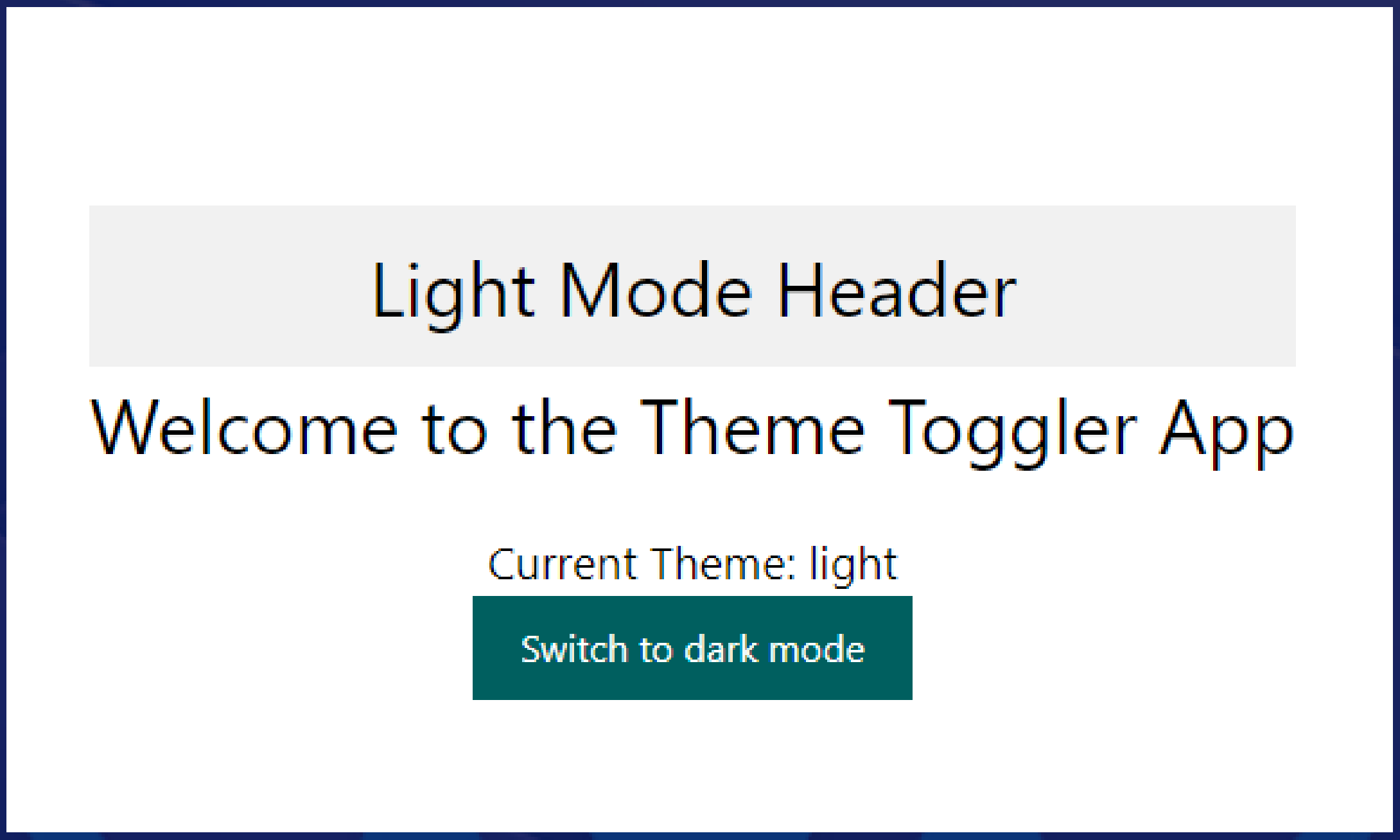
Check next page to see the CSS styles.

After this, run your application using "**npm run dev**" command in your terminal.

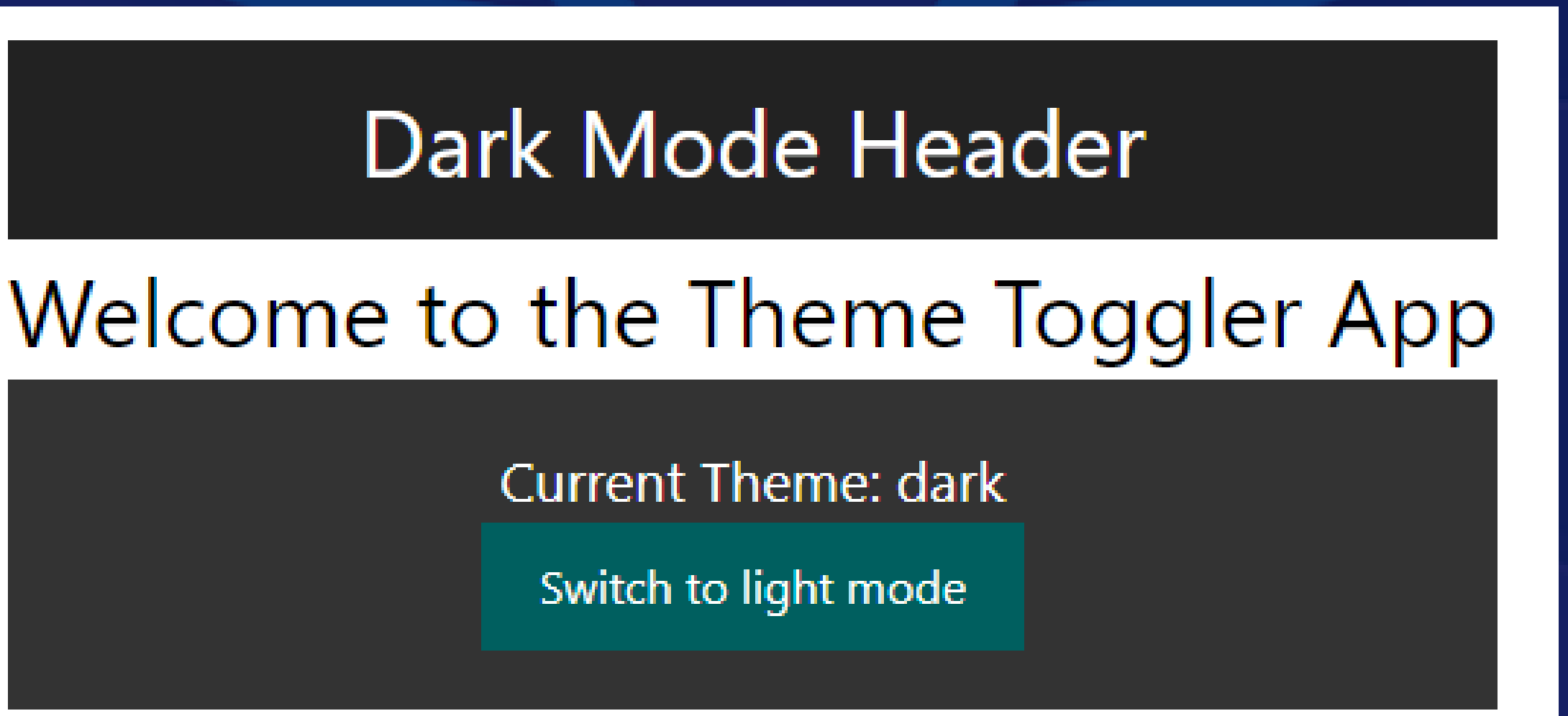
CSS styles:

```css
body {
  font-family: 'Arial', sans-serif;
}

button {
  padding: 10px 20px;
  border: none;
  background-color: #008cba;
  color: white;
  cursor: pointer;
}

button:hover {
  background-color: #005f5f;
}

h1 {
  font-size: 2em;
}

p {
  font-size: 1.2em;
}
```

You should have this in your browser: Light Mode



Dark Mode:



@oluwakemi Oluwadahunsi

When you click on the button, you toggle the different modes for both components (**Header** and **ThemeSwitcher**) at once because they both have access to the Theme Context provider wrapped around them.

Why Context API is Effective for Global State Management

- **Avoids Prop Drilling:** By using the Context API, you don't need to pass props through multiple levels of components. Any component within the Provider can access the global state directly.
- **Simplicity:** For small to medium applications, Context API is a great solution. It's simpler than using external state management libraries like **Redux**.
- **Performance:** When used correctly (memoizing values, avoiding unnecessary renders), Context API can be an efficient state management solution.

When you click on the button, you toggle the different modes for both components (**Header** and **ThemeSwitcher**) at once because they both have access to the Theme Context provider wrapped around them.

Why Context API is Effective for Global State Management

- **Avoids Prop Drilling:** By using the Context API, you don't need to pass props through multiple levels of components. Any component within the Provider can access the global state directly.
- **Simplicity:** For small to medium applications, Context API is a great solution. It's simpler than using external state management libraries like **Redux**.
- **Performance:** When used correctly (memoizing values, avoiding unnecessary renders), Context API can be an efficient state management solution.

Best Practices for Context API in React

1. Avoid Overuse of Context

While Context API is powerful, it's not always the best choice for all state management. Overusing it for every piece of state can lead to unnecessary complexity.

2. Split Contexts for Different Concerns

Group related state into separate contexts instead of putting everything into one context. This allows for better performance and clearer separation of concerns.

For example, create separate contexts for user data and theme settings.

3. Use Context Only for Shared State

Only use Context for state that is truly shared across multiple components. For state that only affects one or two components, use local component state (useState or useReducer).

@oluwakemi Oluwadahunsi

4. Memoize Context Values

If the context value is complex or changes frequently, memoize it to avoid unnecessary re-renders of components that consume the context.

5. Avoid Frequent Re-renders

Context providers re-render every time their value changes, causing all consuming components to re-render.

Keep the context provider near the top-level component and optimize its value to avoid frequent updates.

@oluwakemi Oluwadahunsi

Stay tuned for Part 2, where we'll cover another advanced topic like combining the Context API with Reducers to handle more complex state logic.

@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