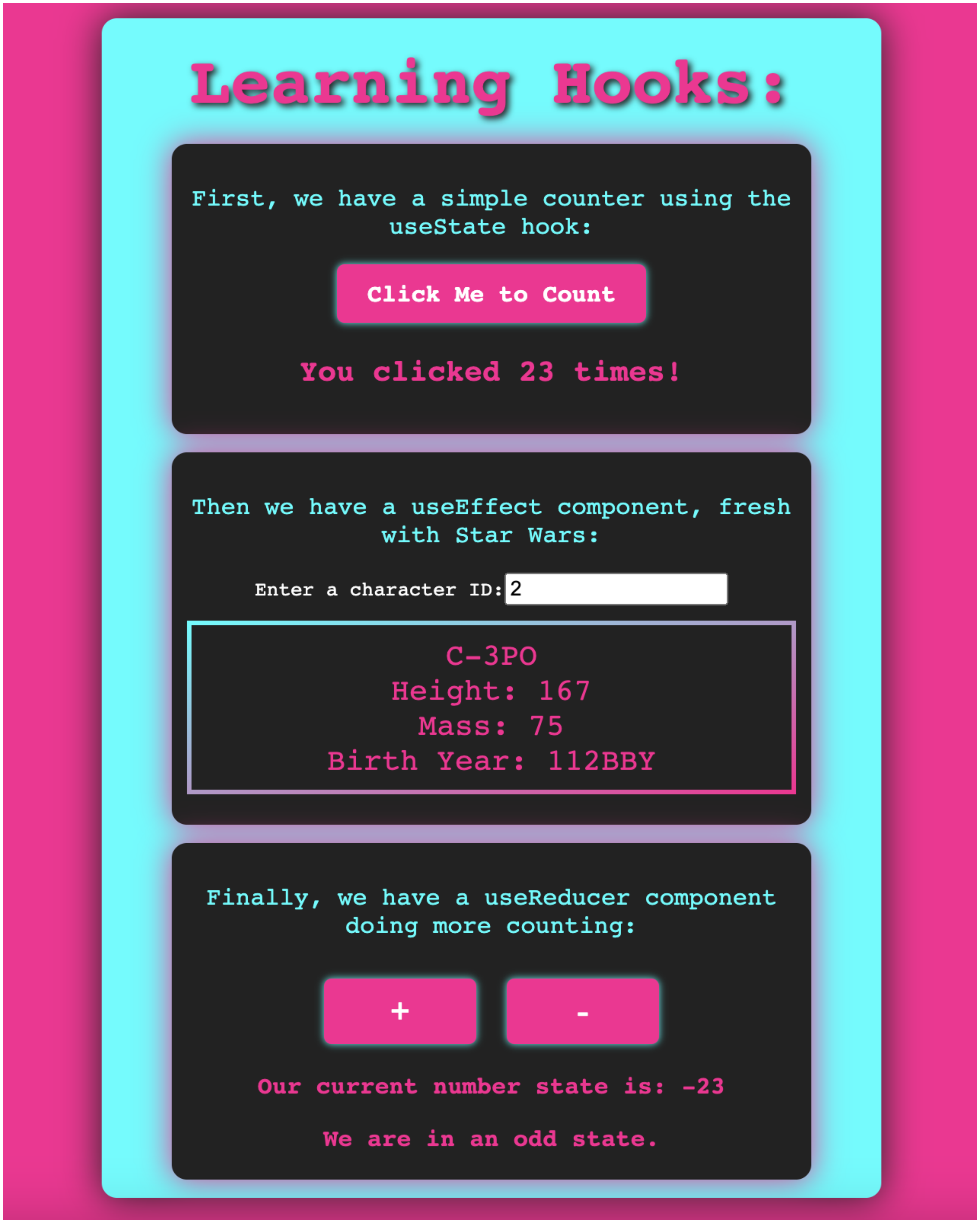
23 April 2023 - Learning Hooks in React.js with Vite and Typescript



Learning Hooks:

First, we have a simple counter using the useState hook:

You clicked 23 times!

Then we have a useEffect component, fresh with Star Wars:

Enter a character ID: 2

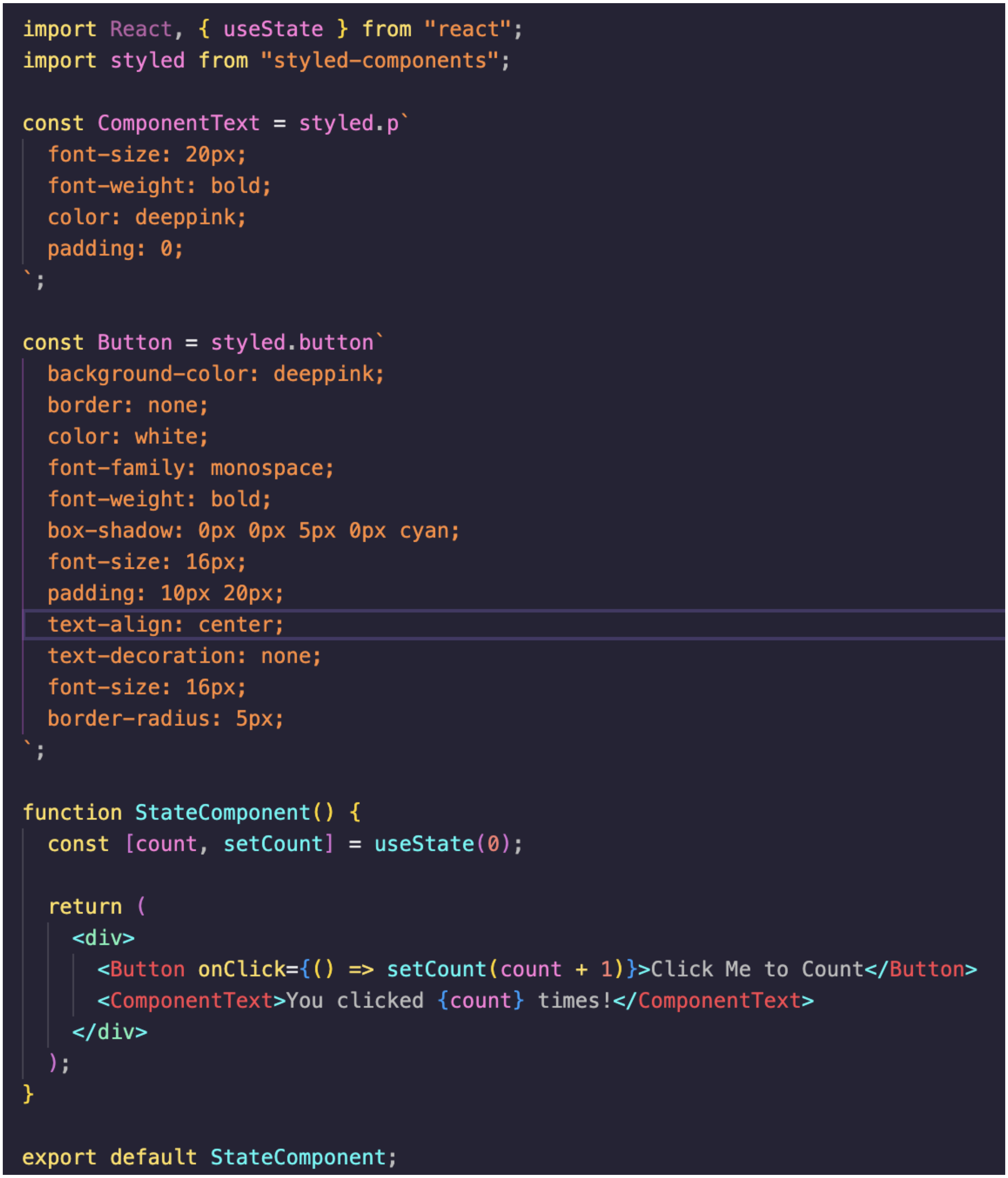
C-3PO
Height: 167
Mass: 75
Birth Year: 112BBY

Finally, we have a useReducer component doing more counting:

Our current number state is: -23

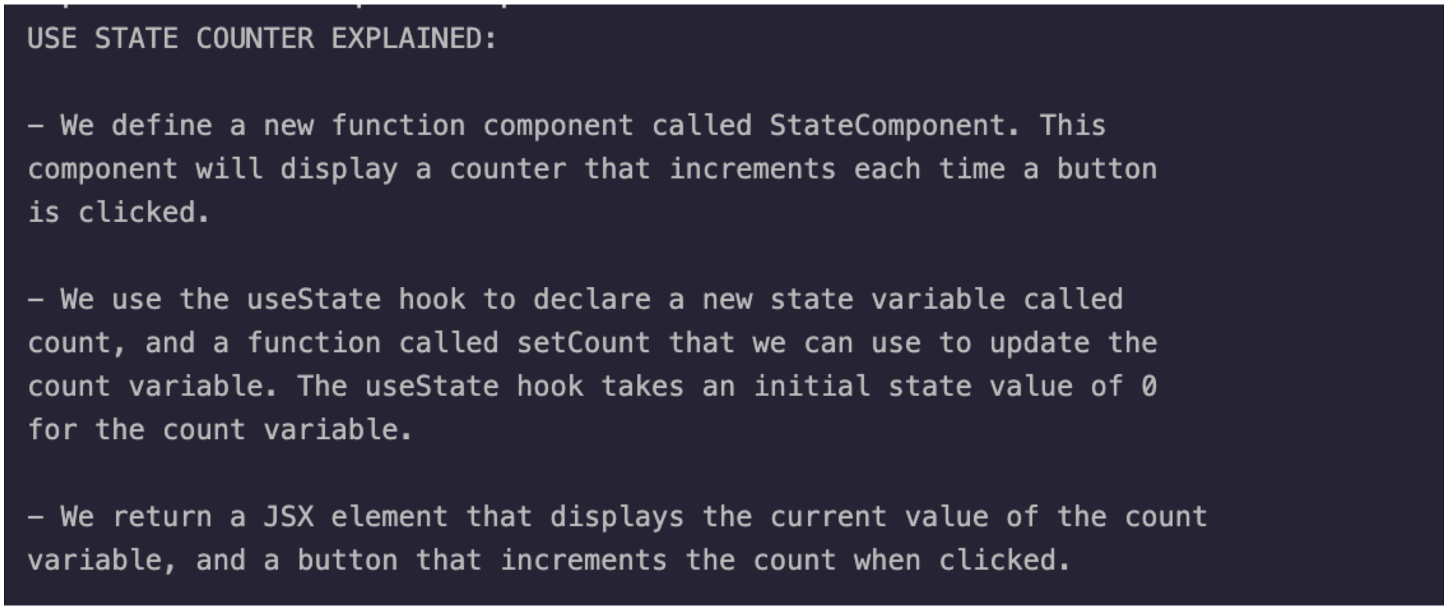
We are in an odd state.

Using useState: CSS Code and Code for StateComponent



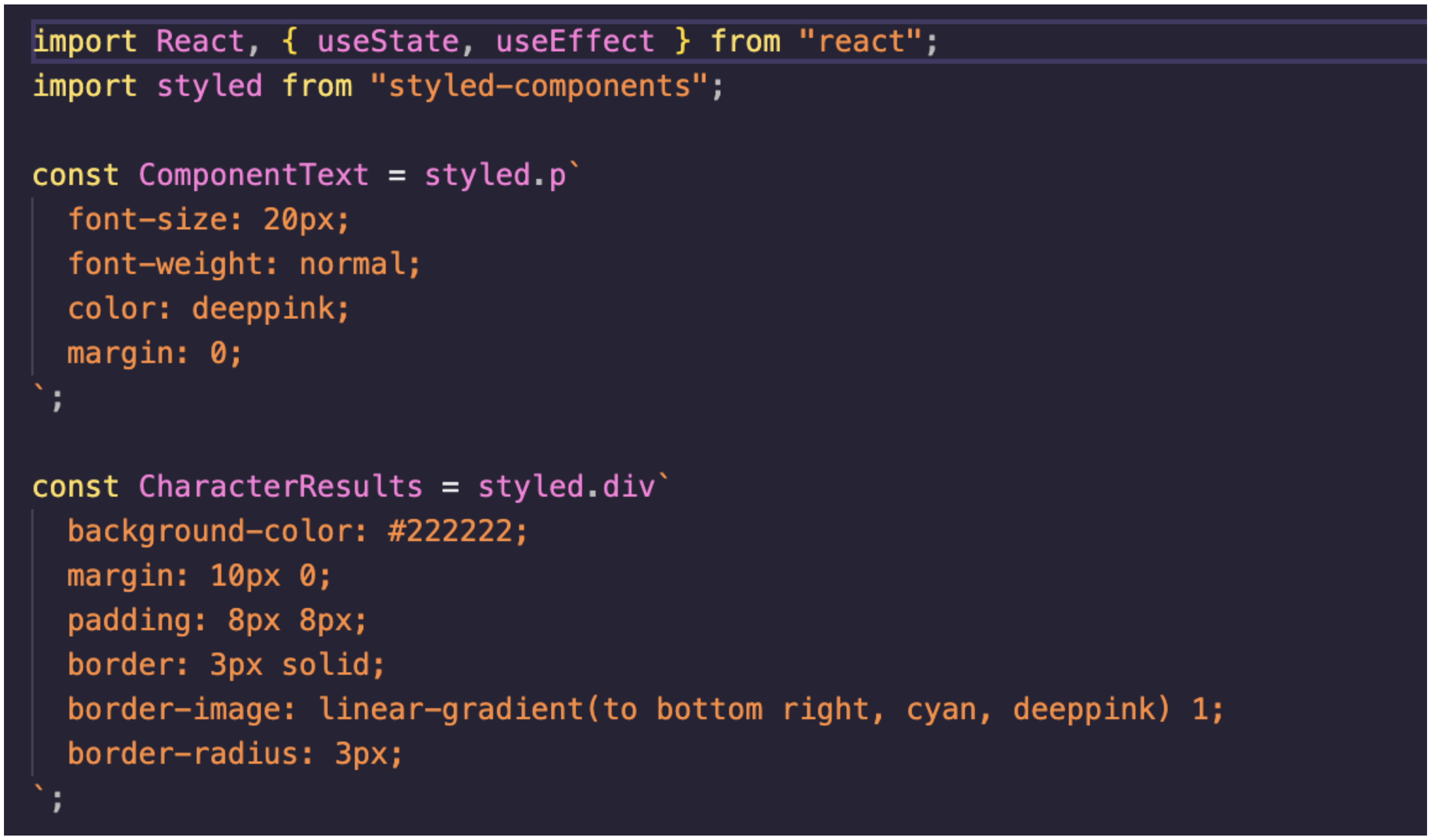
```
import React, { useState } from "react";
import styled from "styled-components";

const ComponentText = styled.p`
  font-size: 20px;
  font-weight: bold;
  color: deeppink;
  padding: 0;
`;

const Button = styled.button`
  background-color: deeppink;
  border: none;
  color: white;
  font-family: monospace;
  font-weight: bold;
  box-shadow: 0px 0px 5px 0px cyan;
  font-size: 16px;
  padding: 10px 20px;
  text-align: center;
  text-decoration: none;
  font-size: 16px;
  border-radius: 5px;
`;

function StateComponent() {
  const [count, setCount] = useState(0);

  return (
    <div>
      <Button onClick={() => setCount(count + 1)}>Click Me to Count</Button>
      <ComponentText>You clicked {count} times!</ComponentText>
    </div>
  );
}

export default StateComponent;
```

Using useState: ChatGPT Explanation of the code



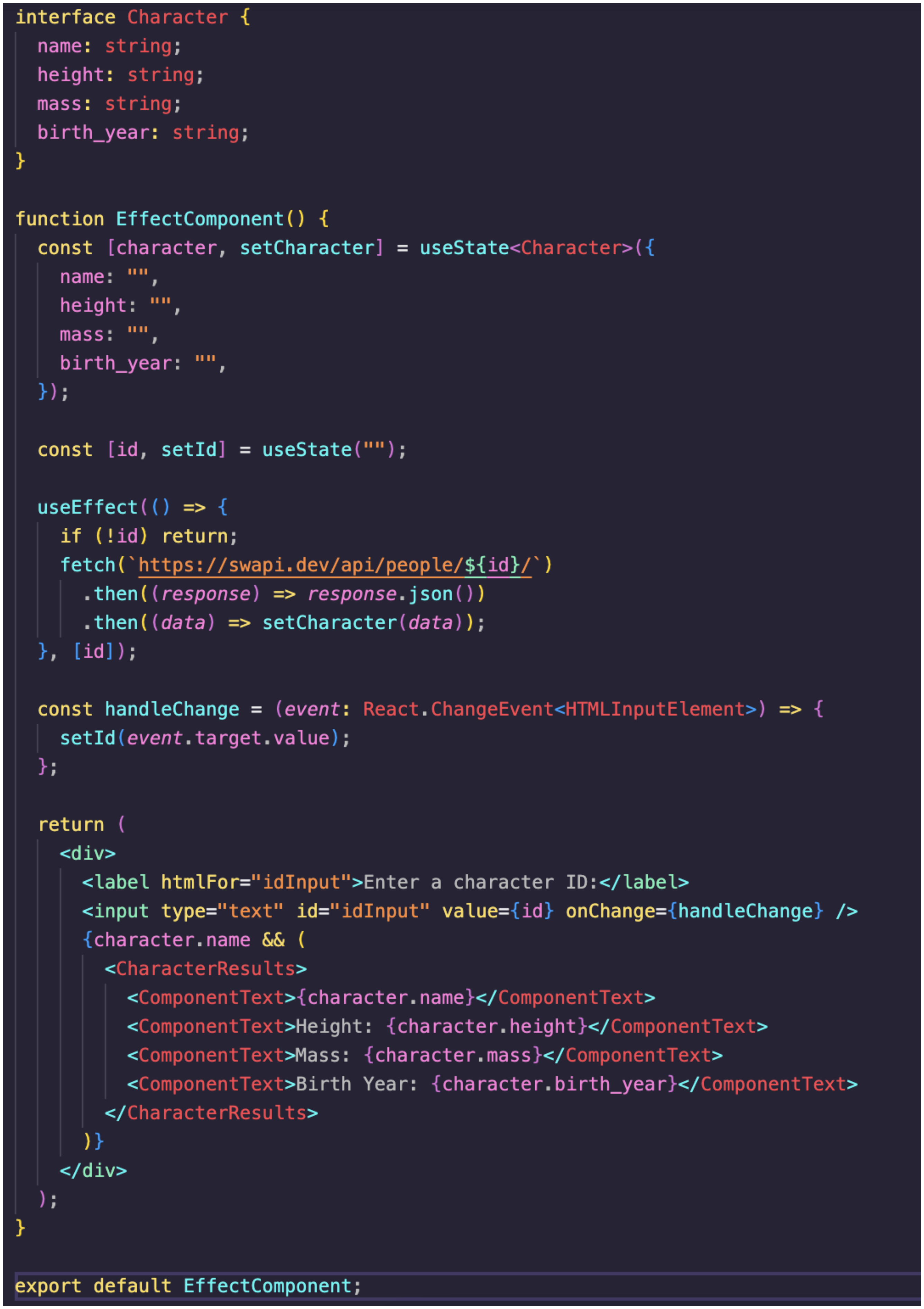
```
USE STATE COUNTER EXPLAINED:

- We define a new function component called StateComponent. This
component will display a counter that increments each time a button
is clicked.

- We use the useState hook to declare a new state variable called
count, and a function called setCount that we can use to update the
count variable. The useState hook takes an initial state value of 0
for the count variable.

- We return a JSX element that displays the current value of the count
variable, and a button that increments the count when clicked.
```

Using useEffect: CSS styling for component



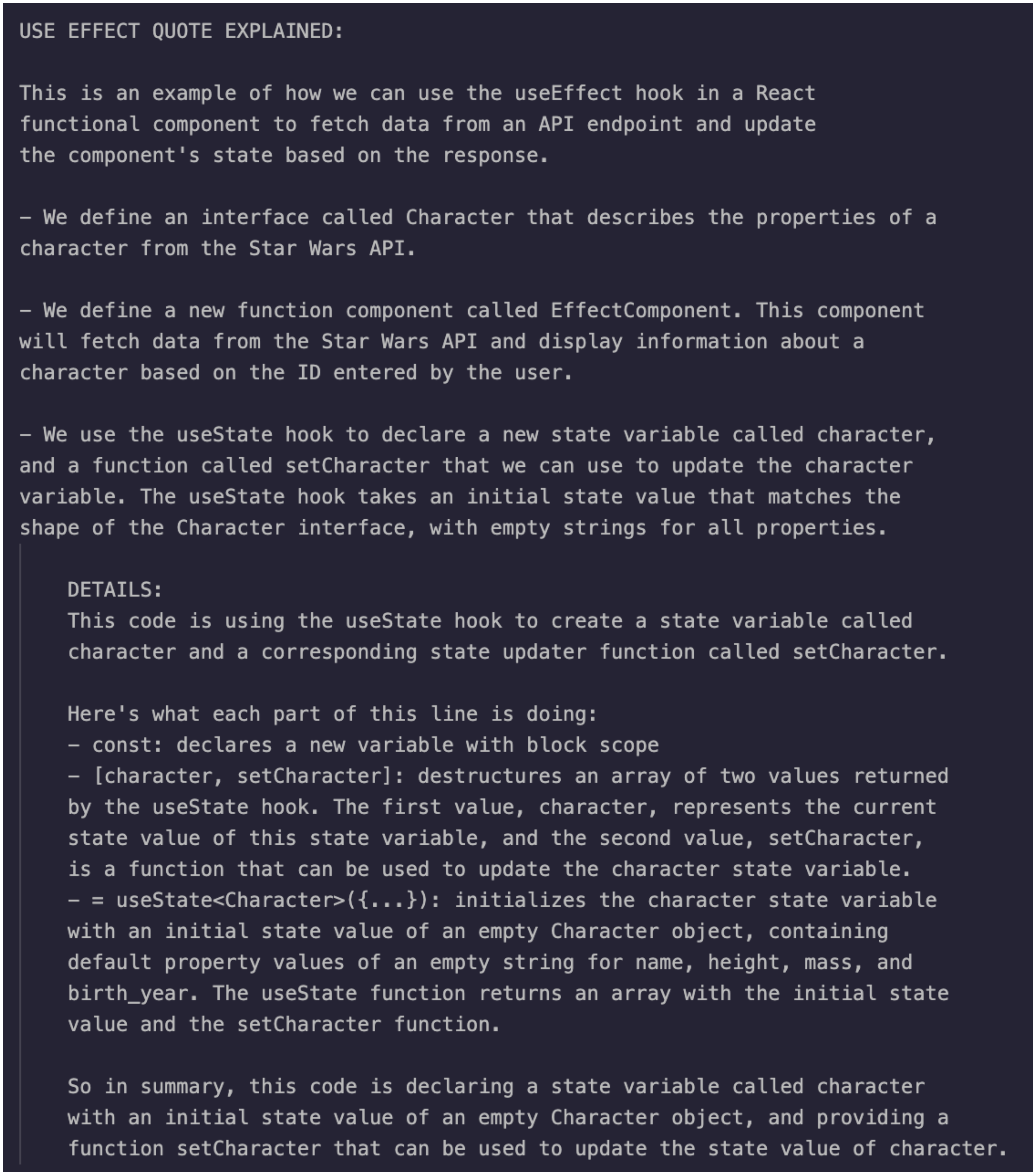
```
import React, { useState, useEffect } from "react";
import styled from "styled-components";

const ComponentText = styled.p`
  font-size: 20px;
  font-weight: normal;
  color: deeppink;
  margin: 0;
`;

const CharacterResults = styled.div`
  background-color: #222222;
  margin: 10px 0;
  padding: 8px 8px;
  border: 3px solid;
  border-image: linear-gradient(to bottom right, cyan, deeppink) 1;
  border-radius: 3px;
`;
```

Using useEffect: Effect component code



```
interface Character {
  name: string;
  height: string;
  mass: string;
  birth_year: string;
}

function EffectComponent() {
  const [character, setCharacter] = useState<Character>({
    name: "",
    height: "",
    mass: "",
    birth_year: "",
  });

  const [id, setId] = useState("");

  useEffect(() => {
    if (!id) return;
    fetch(`https://swapi.dev/api/people/${id}/`)
      .then((response) => response.json())
      .then((data) => setCharacter(data));
  }, [id]);

  const handleChange = (event: React.ChangeEvent<HTMLInputElement>) => {
    setId(event.target.value);
  };

  return (
    <div>
      <label htmlFor="idInput">Enter a character ID:</label>
      <input type="text" id="idInput" value={id} onChange={handleChange} />
      {character.name && (
        <CharacterResults>
          <ComponentText>{character.name}</ComponentText>
          <ComponentText>Height: {character.height}</ComponentText>
          <ComponentText>Mass: {character.mass}</ComponentText>
          <ComponentText>Birth Year: {character.birth_year}</ComponentText>
        </CharacterResults>
      )}
    </div>
  );
}

export default EffectComponent;
```

Using useEffect: ChatGPT explanation of code

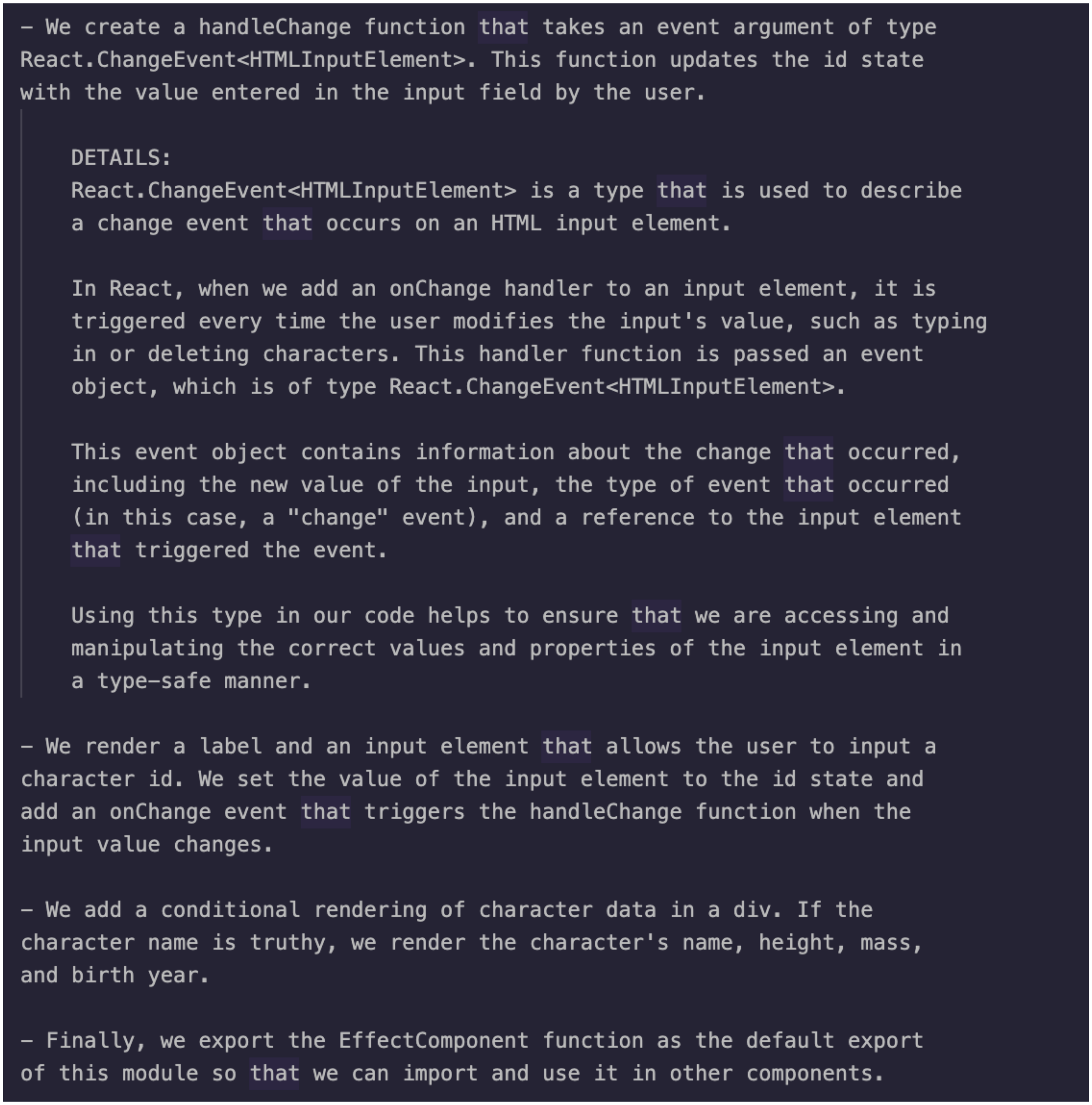


```
USE EFFECT QUOTE EXPLAINED:

This is an example of how we can use the useEffect hook in a React
functional component to fetch data from an API endpoint and update
the component's state based on the response.

- We define an interface called Character that describes the properties of a
character from the Star Wars API.

- We define a new function component called EffectComponent. This component
will fetch data from the Star Wars API and display information about a
character based on the ID entered by the user.

- We use the useState hook to declare a new state variable called character,
and a function called setCharacter that we can use to update the character
variable. The useState hook takes an initial state value that matches the
shape of the Character interface, with empty strings for all properties.

    DETAILS:
    This code is using the useState hook to create a state variable called
    character and a corresponding state updater function called setCharacter.

    Here's what each part of this line is doing:
    - const: declares a new variable with block scope
    - [character, setCharacter]: destructures an array of two values returned
    by the useState hook. The first value, character, represents the current
    state value of this state variable, and the second value, setCharacter,
    is a function that can be used to update the character state variable.
    - = useState<Character>({...}): initializes the character state variable
    with an initial state value of an empty Character object, containing
    default property values of an empty string for name, height, mass, and
    birth_year. The useState function returns an array with the initial state
    value and the setCharacter function.

    So in summary, this code is declaring a state variable called character
    with an initial state value of an empty Character object, and providing a
    function setCharacter that can be used to update the state value of character.
```

- We use the useState hook to declare a new state variable called id, and a function called setId that we can use to update the id variable.

- We use the useEffect hook to fetch data from the Star Wars API when the id state variable changes. The useEffect hook takes a callback function that fetches data from the API using the fetch function, and then updates the character state variable using the setCharacter function. We also pass an array containing id as a dependency to the useEffect hook, which means that the effect will only re-run if the id value changes.

> DETAILS:
> - The useEffect hook is used to perform a side effect in a functional component. In this case, we want to fetch data from an API when the id state changes.
>
> - We use a conditional statement to check if the id state has a value. If it doesn't, we return out of the useEffect hook without doing anything. This is because we only want to fetch data when the id state has a value.
>
> - The fetch function is used to make an HTTP request to the specified URL. In this case, we are requesting data from the Star Wars API using the id state to construct the URL. The backticks () are used to create a template literal, which allows us to embed the id` value into the URL string.
>
> - The fetch function returns a Promise that resolves with a Response object. We can use the .then() method to access the response data. The first .then() method is used to convert the response data to JSON format using the .json() method.
>
> - The second .then() method is used to update the character state with the retrieved data. The data parameter is the parsed JSON data from the API response. We pass this data to the setCharacter function, which updates the character state with the new data.
>
> - The second argument of the useEffect hook is an array of dependencies. This tells React when the useEffect hook should be re-run. In this case, we want to re-run the hook whenever the id state changes, so we include [id] as a dependency.
>
> So in summary, the useEffect hook is used to fetch data from the Star Wars API when the id state changes. We construct the API URL using the id state, make the request using the fetch function, and update the character state with the retrieved data.



- We create a handleChange function that takes an event argument of type React.ChangeEvent<HTMLInputElement>. This function updates the id state with the value entered in the input field by the user.

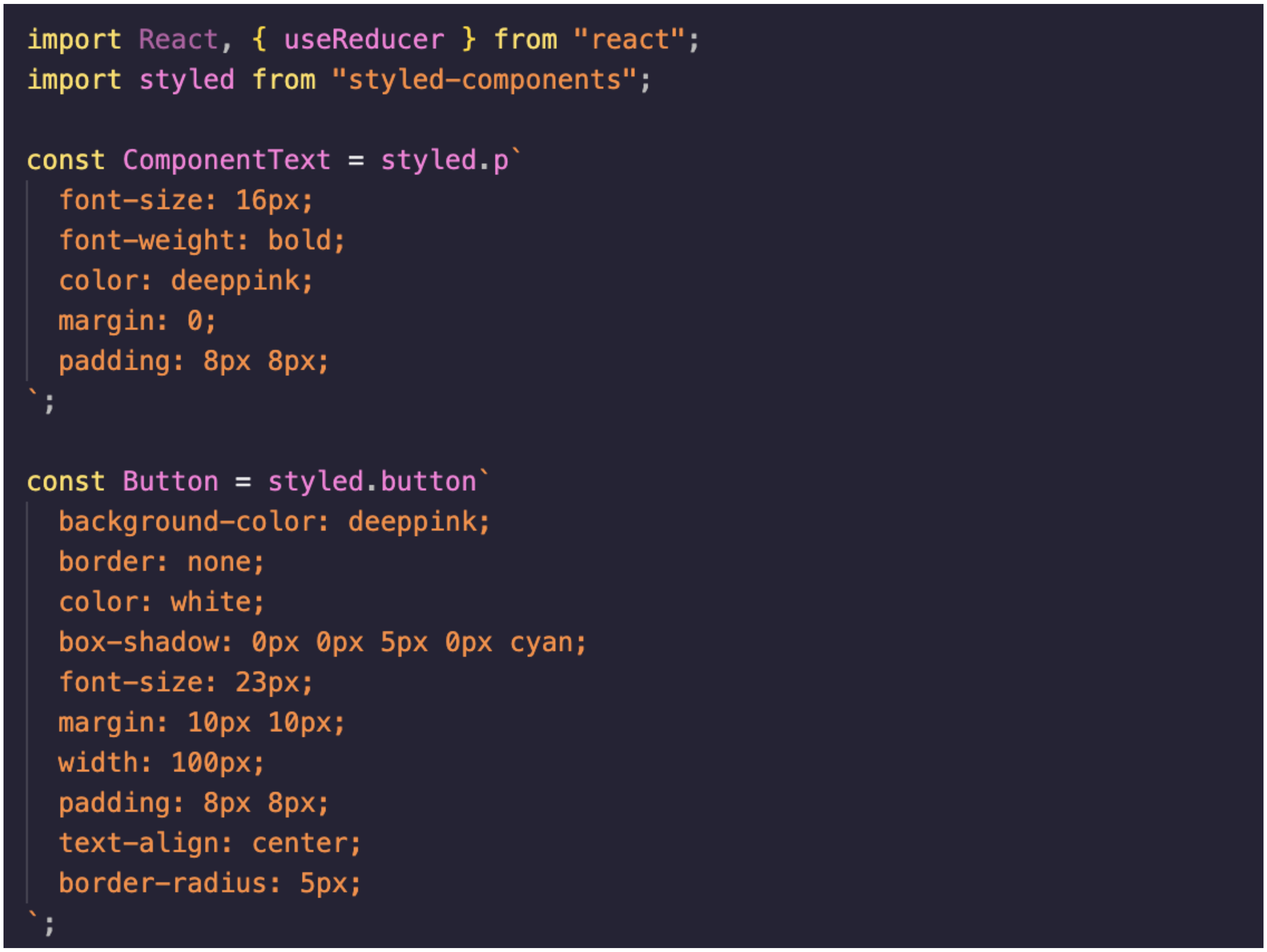
> DETAILS:
> React.ChangeEvent<HTMLInputElement> is a type that is used to describe a change event that occurs on an HTML input element.
>
> In React, when we add an onChange handler to an input element, it is triggered every time the user modifies the input's value, such as typing in or deleting characters. This handler function is passed an event object, which is of type React.ChangeEvent<HTMLInputElement>.
>
> This event object contains information about the change that occurred, including the new value of the input, the type of event that occurred (in this case, a "change" event), and a reference to the input element that triggered the event.
>
> Using this type in our code helps to ensure that we are accessing and manipulating the correct values and properties of the input element in a type-safe manner.

- We render a label and an input element that allows the user to input a character id. We set the value of the input element to the id state and add an onChange event that triggers the handleChange function when the input value changes.

- We add a conditional rendering of character data in a div. If the character name is truthy, we render the character's name, height, mass, and birth year.

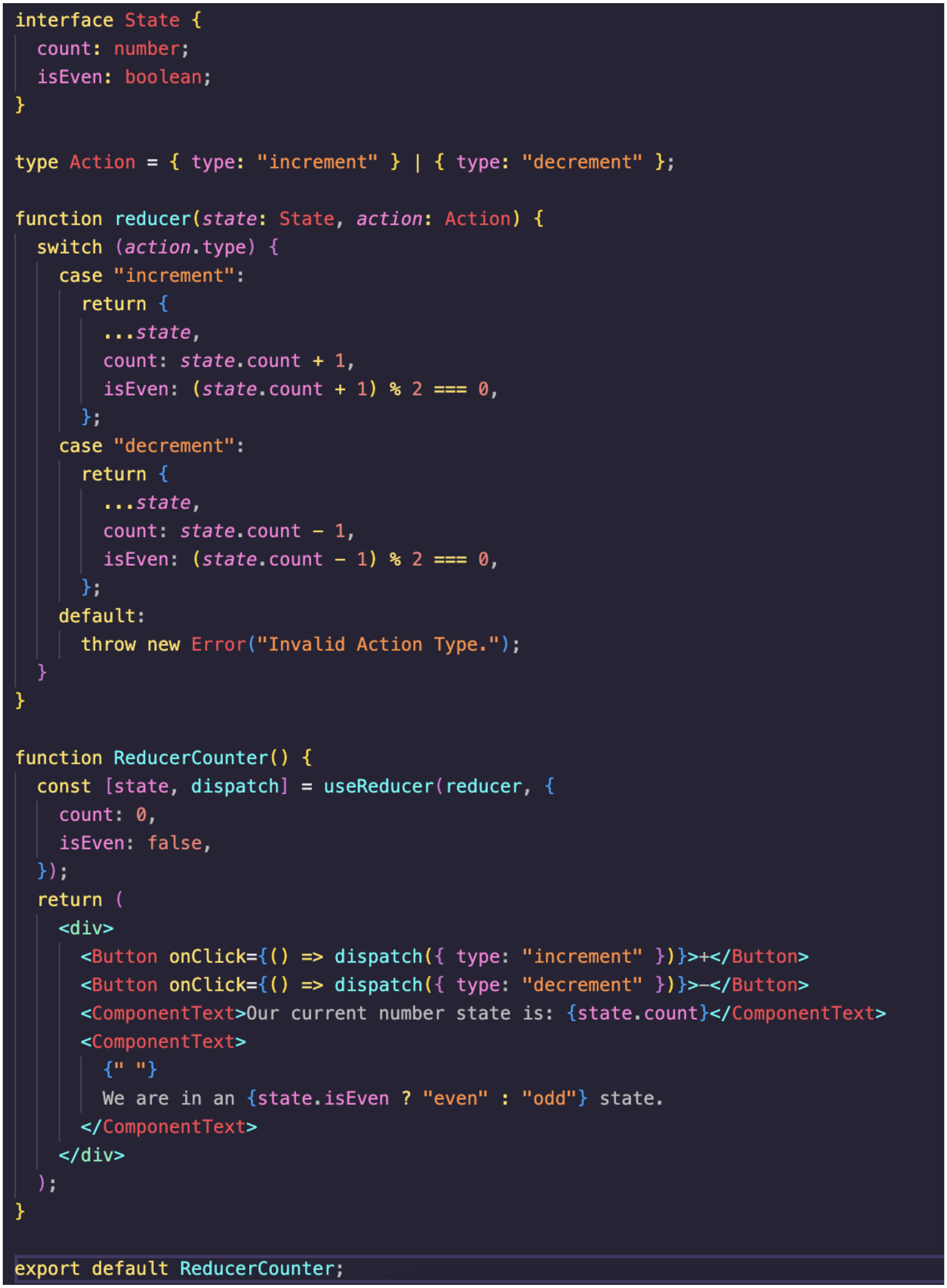
- Finally, we export the EffectComponent function as the default export of this module so that we can import and use it in other components.

Using useReducer: CSS styling for component



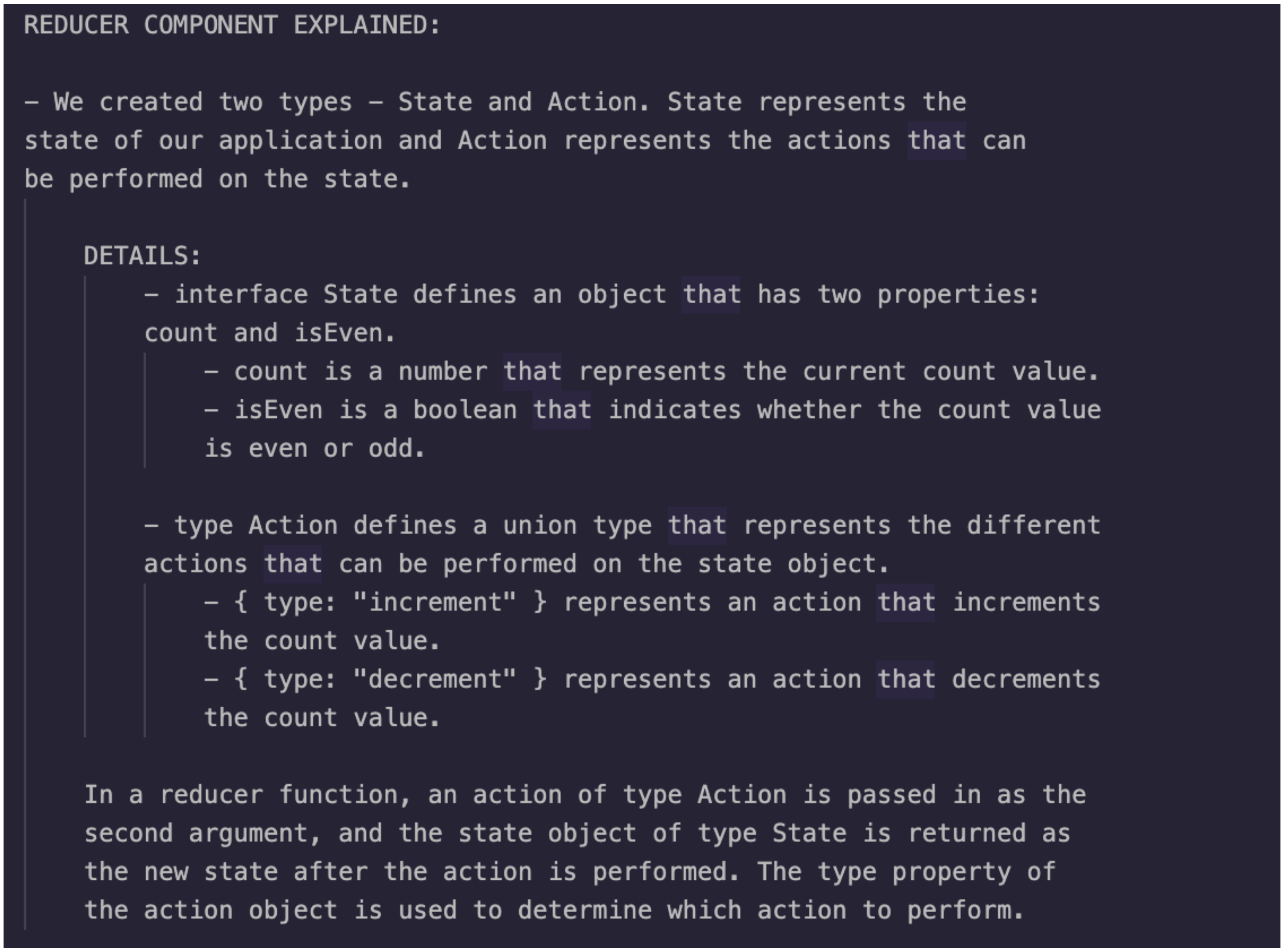
```
import React, { useReducer } from "react";
import styled from "styled-components";

const ComponentText = styled.p`
  font-size: 16px;
  font-weight: bold;
  color: deeppink;
  margin: 0;
  padding: 8px 8px;
`;

const Button = styled.button`
  background-color: deeppink;
  border: none;
  color: white;
  box-shadow: 0px 0px 5px 0px cyan;
  font-size: 23px;
  margin: 10px 10px;
  width: 100px;
  padding: 8px 8px;
  text-align: center;
  border-radius: 5px;
`;
```

Using useReducer: Reducer component code

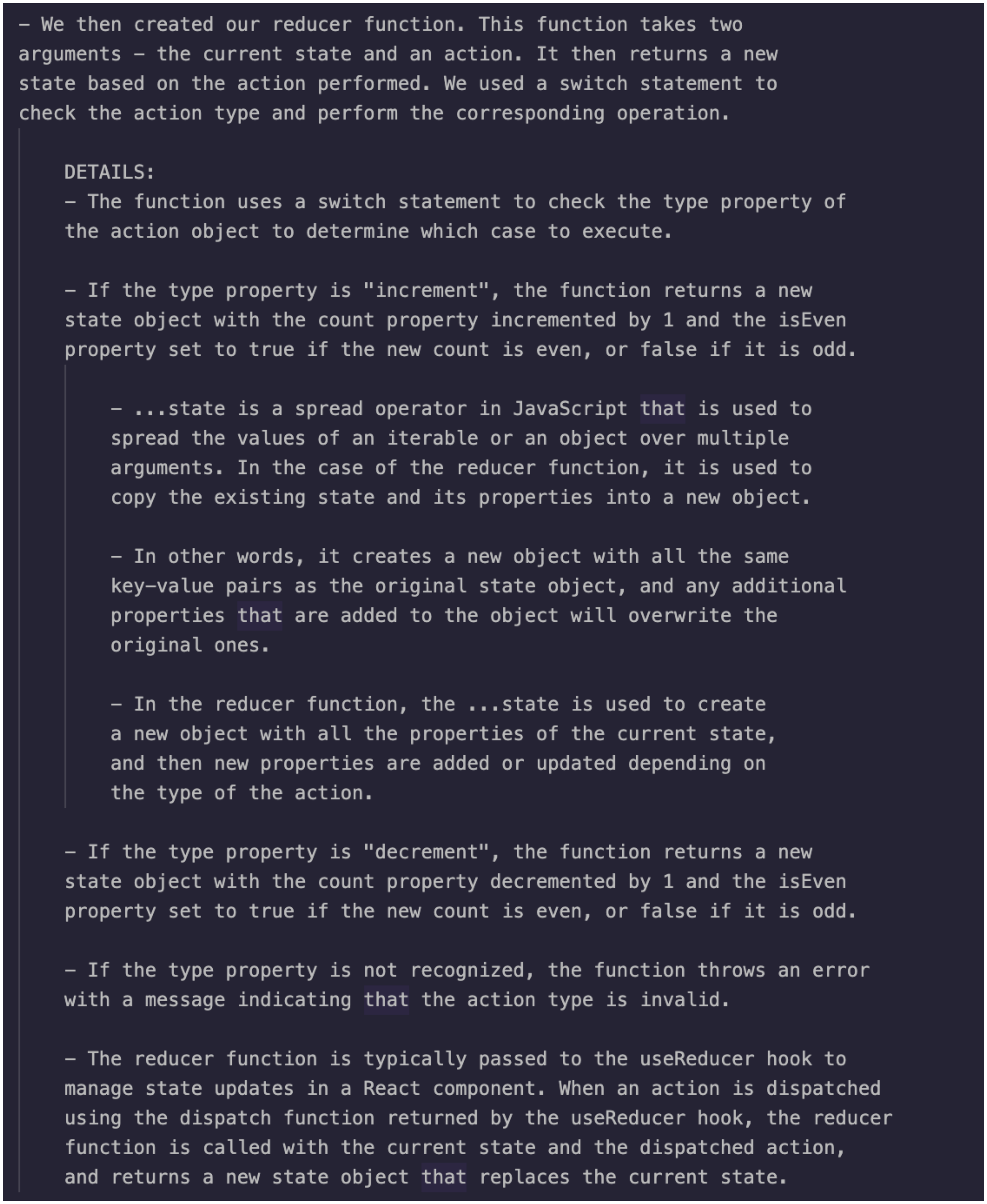


```tsx
interface State {
  count: number;
  isEven: boolean;
}

type Action = { type: "increment" } | { type: "decrement" };

function reducer(state: State, action: Action) {
  switch (action.type) {
    case "increment":
      return {
        ...state,
        count: state.count + 1,
        isEven: (state.count + 1) % 2 === 0,
      };
    case "decrement":
      return {
        ...state,
        count: state.count - 1,
        isEven: (state.count - 1) % 2 === 0,
      };
    default:
      throw new Error("Invalid Action Type.");
  }
}

function ReducerCounter() {
  const [state, dispatch] = useReducer(reducer, {
    count: 0,
    isEven: false,
  });
  return (
    <div>
      <Button onClick={() => dispatch({ type: "increment" })}>+</Button>
      <Button onClick={() => dispatch({ type: "decrement" })}>-</Button>
      <ComponentText>Our current number state is: {state.count}</ComponentText>
      <ComponentText>
        {" "}
        We are in an {state.isEven ? "even" : "odd"} state.
      </ComponentText>
    </div>
  );
}

export default ReducerCounter;
```

Using useReducer: ChatGPT explanation of code



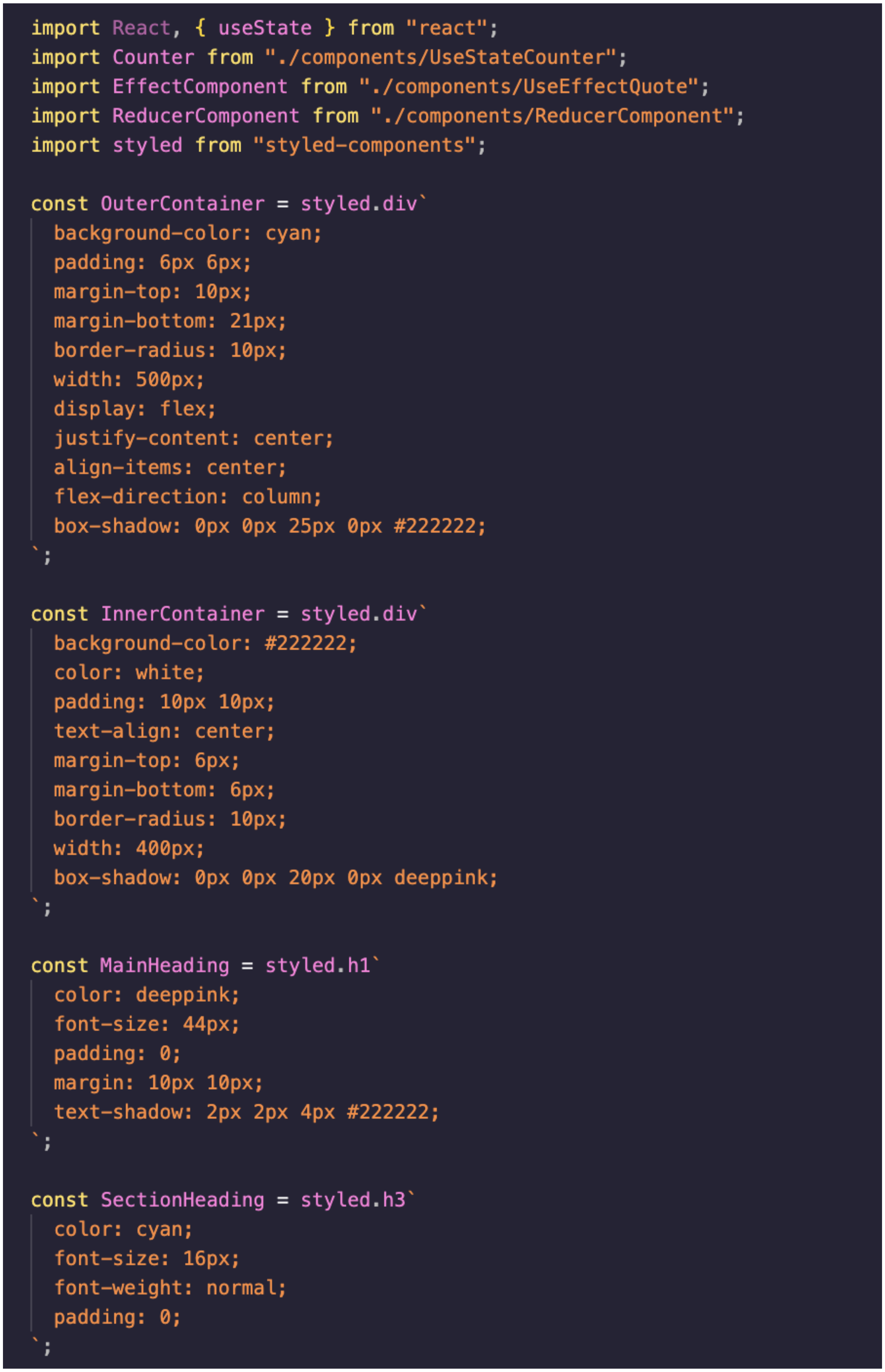
```
REDUCER COMPONENT EXPLAINED:

- We created two types - State and Action. State represents the
state of our application and Action represents the actions that can
be performed on the state.

    DETAILS:
        - interface State defines an object that has two properties:
        count and isEven.
            - count is a number that represents the current count value.
            - isEven is a boolean that indicates whether the count value
            is even or odd.

        - type Action defines a union type that represents the different
        actions that can be performed on the state object.
            - { type: "increment" } represents an action that increments
            the count value.
            - { type: "decrement" } represents an action that decrements
            the count value.

    In a reducer function, an action of type Action is passed in as the
    second argument, and the state object of type State is returned as
    the new state after the action is performed. The type property of
    the action object is used to determine which action to perform.
```



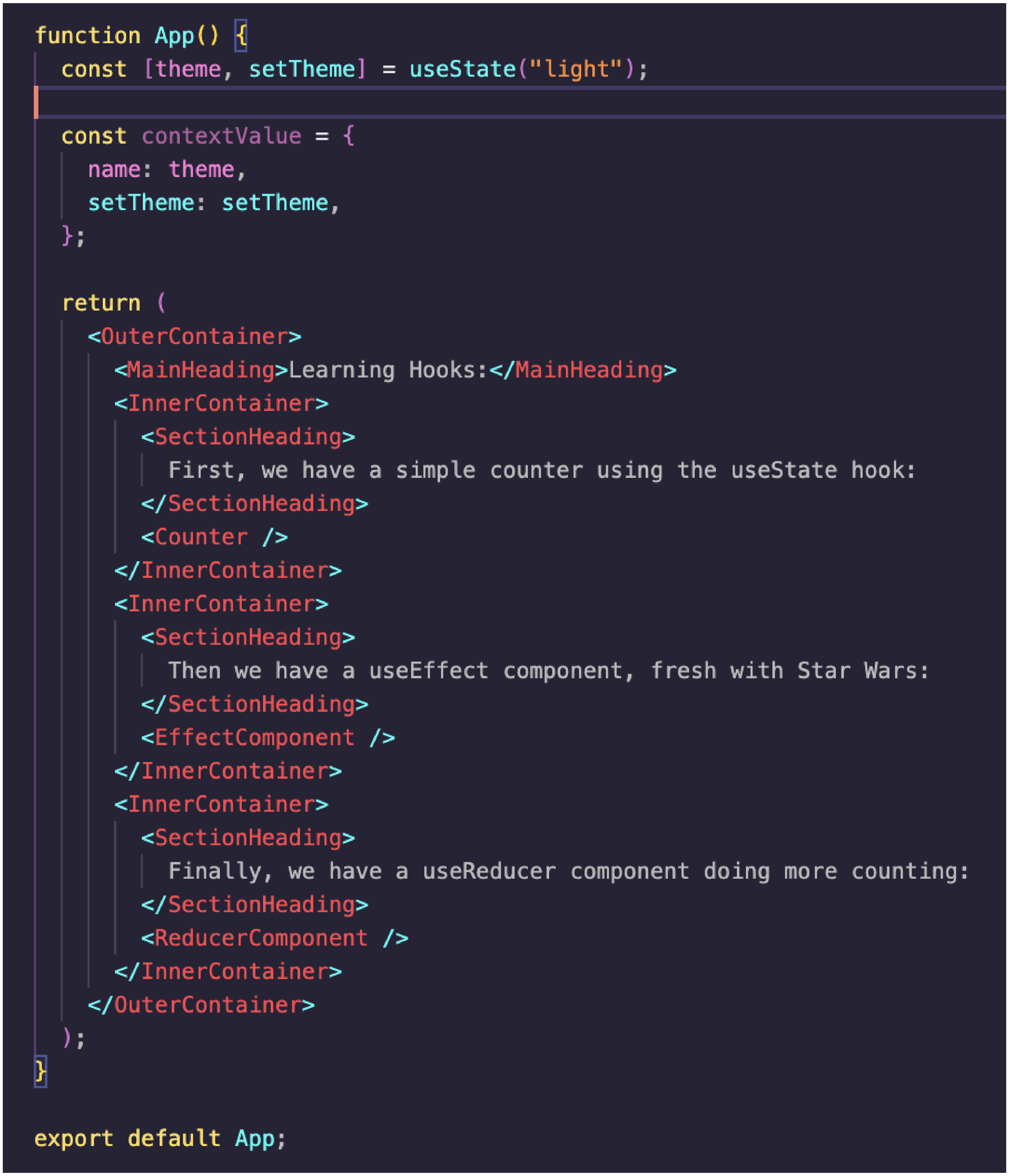
```
- We then created our reducer function. This function takes two
arguments - the current state and an action. It then returns a new
state based on the action performed. We used a switch statement to
check the action type and perform the corresponding operation.

    DETAILS:
    - The function uses a switch statement to check the type property of
    the action object to determine which case to execute.

    - If the type property is "increment", the function returns a new
    state object with the count property incremented by 1 and the isEven
    property set to true if the new count is even, or false if it is odd.

        - ...state is a spread operator in JavaScript that is used to
        spread the values of an iterable or an object over multiple
        arguments. In the case of the reducer function, it is used to
        copy the existing state and its properties into a new object.

        - In other words, it creates a new object with all the same
        key-value pairs as the original state object, and any additional
        properties that are added to the object will overwrite the
        original ones.

        - In the reducer function, the ...state is used to create
        a new object with all the properties of the current state,
        and then new properties are added or updated depending on
        the type of the action.

    - If the type property is "decrement", the function returns a new
    state object with the count property decremented by 1 and the isEven
    property set to true if the new count is even, or false if it is odd.

    - If the type property is not recognized, the function throws an error
    with a message indicating that the action type is invalid.

    - The reducer function is typically passed to the useReducer hook to
    manage state updates in a React component. When an action is dispatched
    using the dispatch function returned by the useReducer hook, the reducer
    function is called with the current state and the dispatched action,
    and returns a new state object that replaces the current state.
```

```
- We then created our ReducerCounter component. This component uses
the useReducer hook to manage the state of our application. It takes
two arguments - our reducer function and an initial state object. It
returns the current state and a dispatch function that can be used to
perform actions on the state.

- Finally, we created the buttons that can be used to increment and
decrement the count. We used the dispatch function to perform the
corresponding actions when the buttons are clicked. We also displayed
the current state in two separate paragraphs.

Here are some examples of tasks that could call for using reducer functions:

    - Counters: Reducers are often used to maintain state for simple counters
    that are incremented or decremented based on user actions.

    - Forms: A form with multiple inputs may require a complex state object
    that can be modified by different actions such as "set name", "set email",
    "set phone", etc. Reducers can be used to manage this state and handle
    these actions.

    - Shopping carts: A shopping cart may need to keep track of items added
    or removed, and the total cost of the items in the cart. A reducer can
    be used to manage the cart state and handle the add/remove item actions.

    - Authentication: Authentication is a common use case where reducers
    can be used to maintain the state of the user's login status and handle
    login and logout actions.

    - Pagination: Reducers can also be used to manage pagination state, such
    as the current page number, page size, and total number of pages. The
    reducer can handle actions such as "set current page", "set page size",
    and "set total number of pages".
```

App.tsx: CSS styling and code



```
import React, { useState } from "react";
import Counter from "./components/UseStateCounter";
import EffectComponent from "./components/UseEffectQuote";
import ReducerComponent from "./components/ReducerComponent";
import styled from "styled-components";

const OuterContainer = styled.div`
  background-color: cyan;
  padding: 6px 6px;
  margin-top: 10px;
  margin-bottom: 21px;
  border-radius: 10px;
  width: 500px;
  display: flex;
  justify-content: center;
  align-items: center;
  flex-direction: column;
  box-shadow: 0px 0px 25px 0px #222222;
`;

const InnerContainer = styled.div`
  background-color: #222222;
  color: white;
  padding: 10px 10px;
  text-align: center;
  margin-top: 6px;
  margin-bottom: 6px;
  border-radius: 10px;
  width: 400px;
  box-shadow: 0px 0px 20px 0px deeppink;
`;

const MainHeading = styled.h1`
  color: deeppink;
  font-size: 44px;
  padding: 0;
  margin: 10px 10px;
  text-shadow: 2px 2px 4px #222222;
`;

const SectionHeading = styled.h3`
  color: cyan;
  font-size: 16px;
  font-weight: normal;
  padding: 0;
`;
```



```
function App() {
  const [theme, setTheme] = useState("light");

  const contextValue = {
    name: theme,
    setTheme: setTheme,
  };

  return (
    <OuterContainer>
      <MainHeading>Learning Hooks:</MainHeading>
      <InnerContainer>
        <SectionHeading>
          First, we have a simple counter using the useState hook:
        </SectionHeading>
        <Counter />
      </InnerContainer>
      <InnerContainer>
        <SectionHeading>
          Then we have a useEffect component, fresh with Star Wars:
        </SectionHeading>
        <EffectComponent />
      </InnerContainer>
      <InnerContainer>
        <SectionHeading>
          Finally, we have a useReducer component doing more counting:
        </SectionHeading>
        <ReducerComponent />
      </InnerContainer>
    </OuterContainer>
  );
}

export default App;
```