**Using the Context System**

We can use an alternative props, which is context. The goal is to reduce tediuous, we can share information through Context.

Context is not a replacement for props.

Context is not a replacement for redux.

A context object has two components: Provider and Consumer.

Provider is a component used to specify what data we want to share, and Consumer is used to get access to data.

import { createContext } from "react";

const BookContext = createContext();

<BookContext.Provider value={5}>

    <MyComponent />

</BookContext.Provider>

To share data, use the value props in context.Provider.

<MyComponent /> and its children can now access the value shared in context (5 in this example).

import { useContext } from "react";

import BookContext from "./book";

function SomeFunc() {

    const num = useContext(BookContext);

    return (

        <div>

            {num}

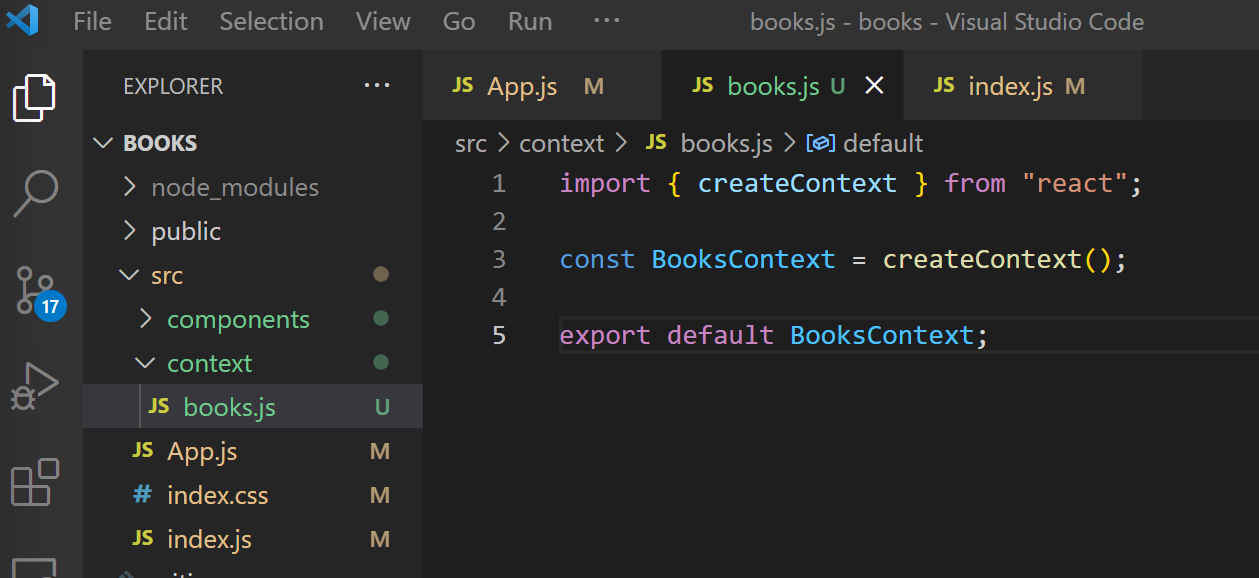
        </div>

    );

}

To get the values from context, call useContext(wantedContext).

We usually create a context file in a context folder in src.



We would then update our index.js file to make app() a component that receives context data.

Text

Description automatically generated

Currently, the value in shared context is static. We want dynamic.

(we need a second provider to wrap the first provider)

Text

Description automatically generated

Below is the code needed for this process.

For the BooksContext file, use createContext to create the context.

import { createContext, useState } from "react";

const BooksContext = createContext();

function Provider( {children} ) {

    const [count, setCount] = useState(5);

    const incrementCount = () => {

        setCount(count+1);

    };

    const valueToShare = {

        count,

        incrementCount,

    };

    return (

        <BooksContext.Provider value={valueToShare}>

            {children}

        </BooksContext.Provider>

    );

}

export { Provider };

export default BooksContext;

For index.js, apply the Provider function.

import { Provider } from "./context/books";

const el = document.getElementById("root");

const root = ReactDOM.createRoot(el);

root.render(

    <Provider>

        <App />

    </Provider>

);

For the other components where we want to use the BooksContext, we import and useContext.

import BookShow from "./BookShow";

import { useContext } from "react";

import BooksContext from "../context/books";

function BookList( {books, onDelete, onEdit} ) {

    const { count, incrementCount } = useContext(BooksContext);

    const renderedBooks = books.map(

        (book) => {

            return <BookShow key={book.id} book={book} onDelete={onDelete} onEdit={onEdit}/>

        }

    )

    return (

        <div className="book-list">

        {count}

        <button onClick={incrementCount}>click</button>

        {renderedBooks}

        </div>

    );

}

export default BookList;

Let us now look at other data we want to share across the app.

Two definitions:

1. Application state: data that is used by many different components
2. Component state: data that is used by very few components (local state)

**Hooks**

Functions given to us by react that add additional features to a component.

useState allows a component to use the state system.

useEffect allows a component to run code at specific points in time.

useContext allows a component to access values stored in context.

**Custom Hooks**

Functions we write to make reusable bits of logic, like similar to useState, useEffect, etc.

import BookShow from "./BookShow";

import { useContext } from "react";

import BooksContext from "../context/books";

function BookList() {

    const { books } = useContext(BooksContext);

change to below

import BookShow from "./BookShow";

import { useContext } from "react";

import BooksContext from "../context/books";

function useBooksContext() {

    return useContext(BooksContext);

}

function BookList() {

    const { books } = useBooksContext();

This is equivalent.

Next we put this custom hook into a new file in a new folder called “hooks”.

import { useContext } from "react";

import BooksContext from "../context/books";

function useBooksContext() {

    return useContext(BooksContext);

}

export default useBooksContext;

import BookShow from "./BookShow";

import useBooksContext from "../hooks/useBooksContext";

function BookList() {

    const { books } = useBooksContext();

**useEffect a closer look**

How does useEffect actually work, and the common bugs?

    useEffect(() => {

        fetchBooks();

    }, []);

When second argument is [], only called once after first render, never called again.

When second argument is nothing, this will be called after every render.

When second argument is [counter], this will be called after first render, and then will be called after every render where the counter changes value.

Note: document.body.onclick = () => {} can be used to do stuff when clicked the whole screen.

What is a stale variable reference?

Every time you render a component, a completely brand new set of variables are declared.

If counter pointed to some address with value 0, when incremented and rendered, we will get a different variable pointing to a different address with value 1, also called “counter”.

When you use useEffect that contains a function that refers to a variable, this bug may happen, called stale variable reference. (always refering to the original address when we should be looking at the new address, since useEffect with [] does not render the second time.)

Create-React-App now has an ESLint rule to help find this bug.

One way to fix bug is to not use [] but use [counter], this will let useEffect allow render everytime counter has changed (address changed) and app is rendering.

We may run into endless cycle of rerenders if we put a function as couter in [counter] where the function itself has ability to call rerender. We can use useCallback hook. It tells react the your function is actually not changing over time. The goal of useCallback is to fix these kinds of bugs in useEffect.

useCallback never runs your function, it just tells react that the function will not change.

    const stableFetchBooks = useCallback(

        fetchBooks,

        []

    );

If second argument is [], useCallback gives you back the original fetchBooks from the first render.

If second argument has elements that changed since last render, it will give you back the new version of fetchBooks.

useCallback is a function. Can also be used like this.

    const stableFetchBooks = useCallback(

        async () => {

            const response = await axios.get("http://localhost:3001/books");

            setBooks(response.data);

        },

        []

    );

Then,

    const { stableFetchBooks } = useContext(BooksContext);

    useEffect(() => {

        stableFetchBooks();

    }, [stableFetchBooks]);

What can we return from a useEffect function?

Cannot return numbers, strings, or async wait functions (returns a promise automatically).

Can only return functions. (a useful function is the cleanUp function.)

An example of a better way for document to listen for clicks.

useEffect(() => {

    const listener = () => {

        console.log(counter);

    };

    document.body.addEventListener("click", listener);

}, []

But everytime we rerender, we are creating a whole new listener function. Thus we need to clean up the old listener functions.

We do this by creating a cleanUp function, and returning that function from uesEffect.

The procedure is:

Diagram

Description automatically generated

useEffect(() => {

    const listener = () => {

        console.log(counter);

    };

    document.body.addEventListener("click", listener);

    const cleanUp = () => {

        document.body.removeEventListener("click", listener);

    };

    return cleanUp;

}, [counter]);

This works since the returned cleanUp doesn’t get called immediately, it gets called at the beginning of the next render.

**Navigation and Routing system**

Sometimes we have inconsistency (buttons style for example).

One solution is to use the button component for all buttons, rather than the <button /> element.

We can reate 1 single Button Component with props that we customize.

We also would want the customized Button to be very similar the the usual button.

            <div>

                <Button text={Click me}/>

            </div>

We want is this

            <div>

                <Button>Click me</Button>

            </div>

We can use the “children” prop. Children prop is default of whatever we place in between.

function Button( {children} ) {

    return <button>{children}</button>;

}

Javascript library: prop-types. It validates the props passed into your component. Now Typescript does similar thing.

We can use the customProp function to assign custom rules for the passed in props.

npm install prop-types.

import PropTypes from "prop-types";

function Button({

    children,

    primary,

    secondary,

    success,

    warning,

    danger,

    rounded,

    outline

}) {

    return <button>{children}</button>;

}

Button.propTypes = {

    checkVariationValue: ({primary, secondary, success, warning, danger}) => {

        const count = Number(!!primary)

         + Number(!!secondary)

         + Number(!!success)

         + Number(!!warning)

         + Number(!!danger);

        if (count > 1) {

            return new Error("only one of primary, secondary, success, warning, danger can be called");

        }

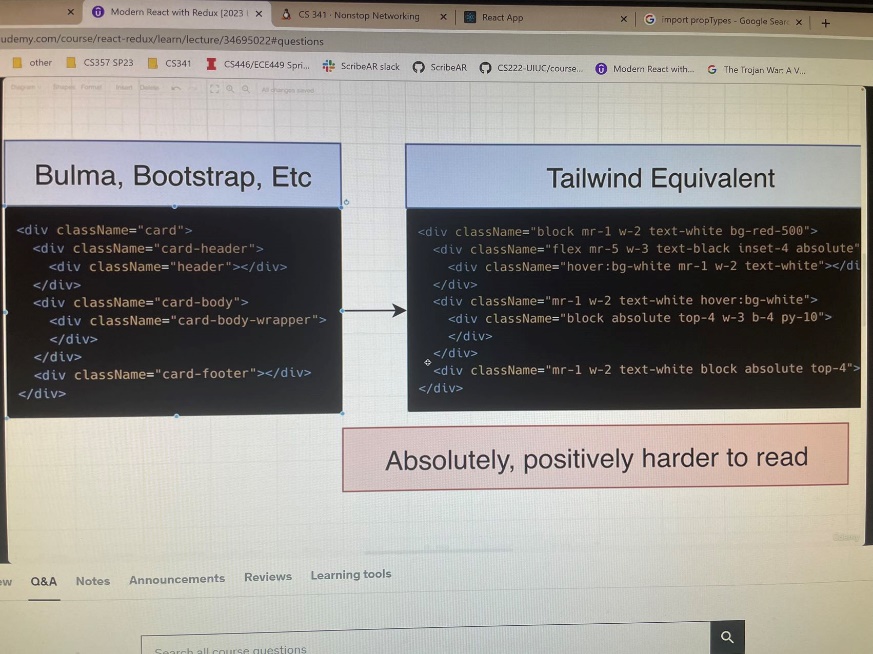
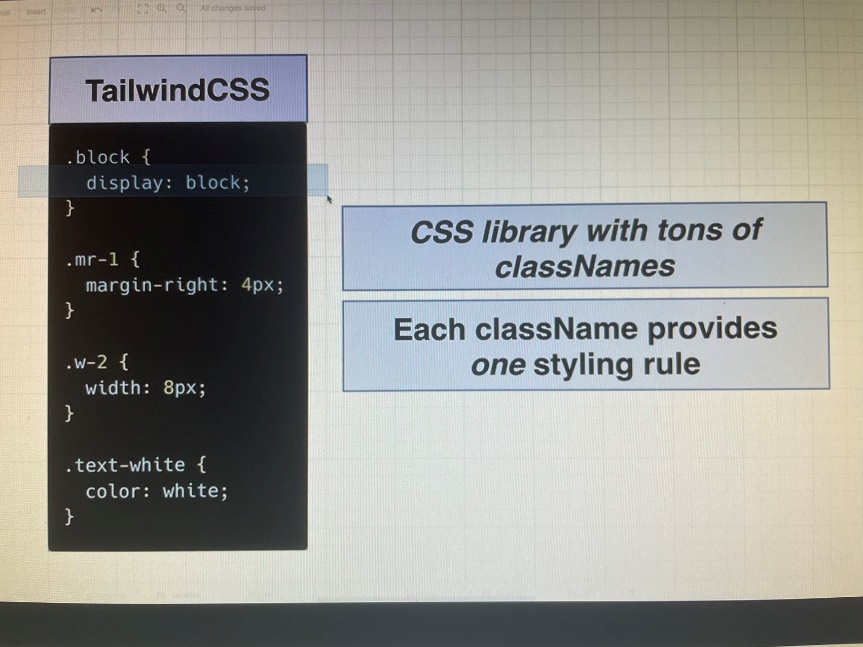
    }

}

export default Button;

TailwindCSS.

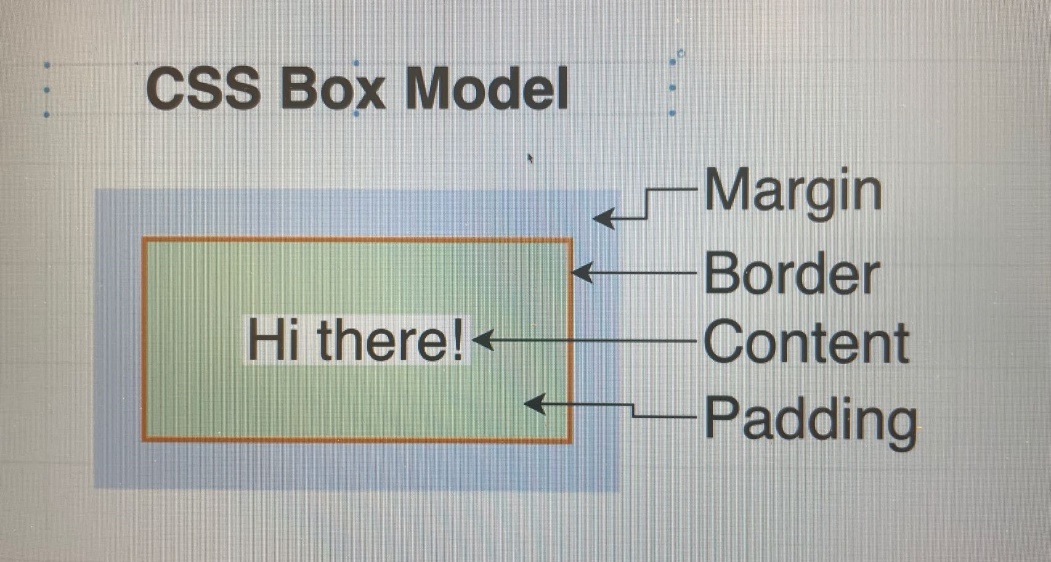
It is a CSS library with tons of classnames, and each classname provides 1 styling rule.



Follow the downloading procedure here: <https://tailwindcss.com/docs/guides/create-react-app>

How to use tailwind?

1. Determine a styling rule you want to add
2. Go to tailwindcss.com/docs
3. Go to quick search (ctrl k)
4. Find the perfect styling rule
5. Apply the correct classname to the element



We need to look at Padding, Border width, Border color, Background color, Text Color in tailwind.

    return <button className="px-3 py-1.5 border border-blue-700 bg-blue-500 text-white">{children}</button>;

}

Javascript classnames library. It builds up a classNmae string based on different props.

It is classnames library but the prop is classNames().

To download we do npm install classnames.

import className from "classnames";

const finalClassName = className('px-1.5', {

    'bg-blue-500': true,

    'text-yellow-500': true,

});

Or

    const classes = className('px-3', 'py-1.5', 'border', {

        'border-blue-500 bg-blue-500 text-white': primary,

        'border-gray-900 bg-gray-900 text-white': secondary,

        'border-green-500 bg-green-500 text-white': success,

        'border-yellow-400 bg-yellow-400 text-white': warning,

        'border-red-500 bg-red-500 text-white': danger,

        'rounded-full': rounded,

        'bg-white': outline,

        'text-blue-500': outline && primary,

        'text-gray-900': outline && secondary,

        'text-green-500': outline && success,

        'text-yellow-400': outline && warning,

        'text-red-500': outline && danger

    });

    return <button className={classes}>{children}</button>;

note that if there is a conflict, then the latter will overwrite the previous.

Icons

We can use the react-icons library. It provides tons of icons.

To install, we run npm install react-icons.

import { GoBell } from 'react-icons/go';

                <Button success danger>

                <GoBell />

                    hi

                </Button>

Wherever we have a button that has a svg, we do margin-right 5px.

button > svg {

    margin-right: 5px;

}

Now, how do we handle events?

Simple solution: pass the prop and pass to the actual plain button.

Problem is we need to add other props like onMouseOver, onClick, onMouseLeave,….

Some props are intended for our own Button. Some other props are just intended for the usual plain button.

Better solution: use …rest.

function Button({

    children,

    primary,

    secondary,

    success,

    warning,

    danger,

    rounded,

    outline,

    ...rest

}) {

This collects the rest of the props that are intended to the usual plain button.

    return <button {...rest} className={classes}>

        {children}

    </button>;

One potential bug would be overwriting. (if the user passes their own className prop). To solve this, we do

const classes = className(rest.className, 'flex items-center', 'px-3', 'py-1.5', 'border', {