**Material UI Customization (TypeScript)**

[#typescript HYPERLINK "https://dev.to/t/frontend"#frontend HYPERLINK "https://dev.to/t/react"#react HYPERLINK "https://dev.to/t/mui"#mui](https://dev.to/t/typescript)

This is a summary of different customizations I did while building one of my projects. you can read more about it [here](https://dev.to/rashidshamloo/clipboard-landing-page-the-process-311i).

**- Component Customization**

I've used multiple methods of customizing Material UI components in this project:

* Using inline properties and style:

import { Typography } from "@mui/material";

<Typography

fontSize={18}

fontWeight={600}

style={{ textDecoration: "line-through" }}

>

TEXT

</Typography>

* Using the [*sx*](https://mui.com/system/getting-started/the-sx-prop/) property which provides access to the [theme](https://mui.com/material-ui/customization/theming/) and [breakpoints](https://mui.com/material-ui/customization/breakpoints/) and some shorthand properties like p and m instead of padding and margin:

import { Typography, SxProps, Theme } from "@mui/material";

const MyStyles: SxProps<Theme> = (theme: Theme) => ({

mt: 7,

fontSize: {

xs: theme.typography.h4.fontSize,

md: theme.typography.h3.fontSize,

},

fontWeight: 600,

});

<Typography sx={MyStyles}>TEXT</Typography>

* Setting the style on the parent by directly targeting the child's Mui class:

(in this example "&>p" would work too and this method is more suited for other components like [Switch](https://mui.com/material-ui/react-switch/) and classes like ".MuiSwitch-thumb")

import { Box, Typography } from "@mui/material";

<Box

sx={{ "&>.MuiTypography-root": { fontSize: 18, fontWeight: 600 } }}

>

<Typography>Text</Typography>

</Box>

* Setting the style on the parent and using inherit in the child

You can set component properties to have the value of "inherit", in which case they inherit the style of their parent element.

import { Box, Typography } from "@mui/material";

<Box

sx={{ fontSize: 18, fontWeight: 600 }}

>

<Typography fontSize="inherit" fontWeight="inherit">

Text

</Typography>

</Box>

* Using the [styled()](https://mui.com/system/styled/) utility:

import { Typography, TypographyProps, styled() } from "@mui/material";

const CustomTypography = styled(Typography)<TypographyProps>(({ theme }) => ({

fontSize: 18,

fontWeight: 600,

[theme.breakpoints.up("xs")]: {

textAlign: "center",

},

[theme.breakpoints.up("md")]: {

textAlign: "left",

},

}));

* Using a wrapper component:

import { Typography, TypographyProps } from "@mui/material";

const CustomTypography = (props: TypographyProps) => (

<Typography

fontSize={18}

fontWeight="600"

sx={{ textAlign: { xs: "center", md: "left" } }}

{...props}

>

{props.children}

</Typography>

);

* Using the combination of both the [styled()](https://mui.com/system/styled/) utility and a wrapper component:

import { Link, LinkProps, styled() } from "@mui/material";

const CustomLink = (props: LinkProps) => {

const MyLink = styled(Link)<LinkProps>(({ theme }) => ({

color: "inherit",

transition: theme.transitions.create(["color"], {

duration: theme.transitions.duration.standard,

}),

"&:hover": {

color: theme.palette.strongCyan.main,

},

}));

return (

<MyLink {...props} underline="none" rel="noopener">

{props.children}

</MyLink>

);

};

**- Theming**

You can customize the Material UI theme by changing/adding custom colors to the palette or setting a custom font to be used by default. then by wrapping your component in a <ThemeProvider>, the theme will be available to the child components:

import {

ThemeProvider,

createTheme,

PaletteColor,

SimplePaletteColorOptions,

} from "@mui/material/styles";

declare module "@mui/material/styles" {

interface Palette {

strongCyan: PaletteColor;

}

interface PaletteOptions {

strongCyan: SimplePaletteColorOptions;

}

}

const theme = createTheme({

palette: {

strongCyan: { main: "hsl(171, 66%, 44%)" },

},

typography: {

fontFamily: "'Bai Jamjuree', 'sans-serif';",

},

});

...

<ThemeProvider theme={theme}>

<ChildComponent />

</ThemeProvider>

You can also customize your components globally [using the theme](https://mui.com/material-ui/customization/theme-components/):

const theme = createTheme({

components: {

// component

MuiLink: {

// change property defaults

defaultProps: {

underline: "hover"

},

// override CSS

styleOverrides: {

// Mui class

root: {

fontWeight: 600,

}

}

}

}

});

You can define [new variants](https://mui.com/material-ui/customization/theme-components/#creating-new-component-variants) for your components in the theme and use [nested themes](https://mui.com/system/styles/advanced/#theme-nesting) as well.

**- CSS Reset / Normalize**

Some elements have margin and padding values applied by default which can mess up the layout. Material UI provides a handy component called [< HYPERLINK "https://mui.com/material-ui/react-css-baseline/"CssBaseline HYPERLINK "https://mui.com/material-ui/react-css-baseline/">](https://mui.com/material-ui/react-css-baseline/) that acts as a CSS Reset and removes those nasty default stylings:

import { CssBaseline } from "@mui/material";

...

<CssBaseline />

<YourOtherComponents />

In order to apply <CssBaseline> only to some of your components, you can use the [< HYPERLINK "https://mui.com/material-ui/react-css-baseline/#scoping-on-children"ScopedCssBaseline HYPERLINK "https://mui.com/material-ui/react-css-baseline/#scoping-on-children">](https://mui.com/material-ui/react-css-baseline/#scoping-on-children) component instead:

import { ScopedCssBaseline } from "@mui/material";

...

<Component />

<ScopedCssBaseline>

<AffectedComponent />

</ScopedCssBaseline>

**- Transitions**

To add transitions to Material UI components, you can use the theme.transitions.create() function which takes the properties you want to apply transition to as the first argument and a settings object as the second. you can set the duration to the value defined in the theme so it's easy to adjust/change at a later stage:

sx={(theme) => ({

transition: theme.transitions.create(

["color", "background-color"],

{

duration: theme.transitions.duration.standard,

}

),

})}

**- Media Queries**

Material UI provides a handy [useMediaQuery()](https://mui.com/material-ui/react-use-media-query/) hook we can use to detect the screen size and do things like showing/hiding a component on certain screen sizes or in my case, disabling animation delays on smaller screens.  
You can use it like this:

import { useMediaQuery, Theme } from "@mui/material";

...

const matches = useMediaQuery((theme: Theme) => theme.breakpoints.up("md"));

In this case, matches will be true if the screen is bigger than md (medium) and false if it's not. then you can use it like any other boolean variable to add conditions to your logic/render.  
You can also use exact pixel values: useMediaQuery('(min-width: 900px)')

**- Customizing the child component of another component**

Some components in Material UI have other nested components inside of them. for example a [Dialog](https://mui.com/material-ui/react-dialog/) component has a [Paper](https://mui.com/material-ui/react-paper/) component inside. and in order to customize the properties of the nested component, it exposes a property called PaperProps which you can use to do that. You will have to check the [Material UI API](https://mui.com/material-ui/api/dialog/) to know all the properties available for each component.

<Dialog

PaperProps={{

sx: {

borderRadius: '1rem'

},

}}

>

...

</Dialog>

**- Forwarding ref to component children**

Some components like [Tooltip](https://mui.com/material-ui/react-tooltip/) need to assign ref to their children to work properly, which means if you place a custom component inside a Tooltip component, you then have to use [React.forwardRef()](https://react.dev/reference/react/forwardRef) with your custom component so it accepts a ref. this is how I implemented a custom [Link](https://mui.com/material-ui/react-link/) inside a custom [Tooltip](https://mui.com/material-ui/react-tooltip/) component:

import React from "react";

import { Link, Tooltip, LinkProps, TooltipProps } from "@mui/material";

// custom Tooltip wrapper component

const MyTooltip = (props: TooltipProps) => (

<Tooltip

{...props}

arrow

placement="top"

TransitionComponent={Fade}

TransitionProps={{ timeout: 500 }}

>

{props.children}

</Tooltip>

);

// custom Link wrapper component

const MyLink = React.forwardRef<HTMLAnchorElement, LinkProps>(

(props, ref) => {

const linkStyles = (theme: Theme) => ({

transition: theme.transitions.create(["filter", "transform", "border"], {

duration: theme.transitions.duration.standard,

}),

"&:hover": {

filter: "brightness(150%)",

transform: "scale(1.2)",

},

});

return (

<Link {...props} target="\_blank" rel="noopener" sx={linkStyles} ref={ref}>

{props.children}

</Link>

);

}

);

...

<MyTooltip title="React.js">

<MyLink href="https://react.dev" target="\_blank">

React.js

</MyLink>

</MyTooltip>

Instead of  
React.forwardRef<HTMLAnchorElement, LinkProps>(props, ref)  
you can use  
React.forwardRef((props: LinkProps, ref: React.Ref<HTMLAnchorElement>)

**- Modifying / Merging sx properties**

Sometimes you need to use an [sx](https://mui.com/system/getting-started/the-sx-prop/) property you've already defined but change or remove some properties from it. There are multiple ways to do this:

**1. Removing properties from the sx prop**

Imagine we want to remove backgroundColor from the sx prop below:

import { SxProps } from "@mui/material";

const myProp: SxProps = {

color: "red",

backgroundColor: "blue",

}

* Using the spread operator

const {backgroundColor, ...myNewProp} = myProp;

* Deleting the key

import { SystemCssProperties } from "@mui/system";

const myNewProp: SystemCssProperties = myProp;

delete myNewProp.backgroundColor;

* Resetting the key by merging

You can reset the backgroundColor property by merging your sx prop with another sx prop like this: {backgroundColor: "transparent"}. merging is explained in the next sections.

**2. Adding / Modifying properties of sx prop**

* Adding/Modifying the key

myProp.backgroundColor = "green";

myProp.mt = 2;

* You can also accomplish this by merging your style with another which will add/replace the required keys. (explained next)

**3. Merging multiple sx properties**

* Using sx Array

sx prop accepts an array as input which can contain multiple sx properties that will be merged together:

<ComponentName sx={[{color: "red"},{backgroundColor: "blue"}]} />

* Using Object.assign()

sx properties are objects, so you can use [Object.assign()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/assign) to merge multiple sx properties together:

const myNewProp: SxProps = Object.assign(myProp, {

backgroundColor: "green",

});

* Using the [merge-sx](https://github.com/RobinTail/merge-sx) package You can use the mergeSx() function provided by this package to merge multiple sx properties.

npm install merge-sx

import { mergeSx } from "merge-sx";

const mySxProp1: SxProps = { ... }

const mySxProp2: SxProps = { ... }

const mySxProp3: SxProps = mergeSx(mySxProp1, mySxProp2);

The good thing about this package is that it works with functional SxProps as well, which I'll explain next.

**4. Dealing with functional sx properties**

sx properties can also be functions that take the theme as input and return an SxProps object:

import { SxProps, Theme } from "@mui/material";

const myStyles: SxProps<Theme> = (theme: Theme) => ({

fontSize: {

xs: theme.typography.h4.fontSize,

md: theme.typography.h3.fontSize,

},

});

This way you can use the variables in your theme inside your sx prop.  
But what this means is that you can't use Object.assign() or modify the keys directly because you're not dealing with objects anymore.  
In this case, the best way is to use the sx array method. just be sure to pass the theme to sx functional properties too. and also you will need to use a more specific type for your sx prop:

import { SxProps, Theme } from "@mui/material";

// wrong type ("This expression is not callable." error)

const myStyle1 : SxProps<Theme> = (theme: Theme) => ({ ...

// correct type

import { SystemStyleObject } from "@mui/system";

type SxPropsFunc<T extends object> = (\_: T) => SystemStyleObject<T>;

const myStyle1 : SxPropsFunc<Theme> = (theme: Theme) => ({...

// wrong ("No overload matches this call." error)

<Component sx={[myStyle1, myStyle2]} />

// correct

<Component

sx={[

(theme:Theme) => myStyle1(theme),

(theme:Theme) => myStyle2(theme)

]}

/>

Figuring out the correct type above took a good chunk of time...

Overall when merging, you can use the [merge-sx](https://github.com/RobinTail/merge-sx) package mentioned before to save yourself some trouble. you can also pass the theme:

import { SxProps, Theme } from "@mui/material";

const myStyles: SxProps<Theme> = (theme: Theme) => {...

<MyComponent

sx={

mergeSx<Theme>(

(theme:Theme) => {

fontSize: theme.typography.h4.fontSize

},

myStyles

)

}

/>