MUI Core v5.3.0 introduces the ability to write a callback in style overrides (global theming), giving you full control of component customization at the theme level.

Why is using a callback better than the existing plain object? Let me explain from the beginning…

The problems

In v4, the style engine library was JSS which had some limitations. Style overrides were not able to support dynamic props via a callback so we relied on using classes. Take a look at the `Chip` classes for example – there are more than 20 classes that are incomplete if we count the permutation of elements (`root | avatar | icon | label | deleteIcon`), size (`small | medium | large`), and color (`primary | secondary | ...`). This leads to a poor theming experience because developers need to know which specific key to customize.

We believe it would be better for developers if they could create custom styles by reading the component props, without ever needing to know what key they should use.

Fortunately, it is now possible in v5 because of the new style engine powered by emotion. Theming is simpler and more flexible. You only need to know the component's slot name and then provide an **object** (static overrides) or a **callback** (dynamic overrides).

Using callback in `styleOverrides`

The callback gives you the `props` that the slot received. Most of the time you would use:

- `props.ownerState` : the combination of runtime props and internal states.
- `props.theme` : the theme object you provided to `ThemeProvider` , or the default one.

```
import { ThemeProvider, createTheme } from '@mui/material/styles';

<ThemeProvider
  theme={createTheme({
    components: {
      MuiChip: {
        styleOverrides: {
          // you can now use the theme without creating the initial theme!
          root: ({ ownerState, theme }) => ({
            padding: {
              small: '8px 4px',
              medium: '12px 6px',
              large: '16px 8px',
            }[ownerState.size],
            ...(ownerState.variant === 'outlined' && {
              borderWidth: '2px',
              ...(ownerState.variant === 'primary' && {
                borderColor: theme.palette.primary.light,
              }),
            }),
          }),
          label: {
            padding: 0,
          },
```

```
        },
      },
    },
  }))}
>
  ...your app
</ThemeProvider>;
```

> 💡 *The side benefit of using a callback is that you can use the runtime theme without creating the outer scoped variable.*

TypeScript

The callback is type-safe.

- `ownerState` : `ComponentProps` interface, eg. `ButtonProps` , `ChipProps` , etc.
- `theme` : `Theme` interface from `@mui/material/styles` .

```
{
  MuiChip: {
    styleOverrides: {
      // ownerState: ChipProps
      // theme: Theme
      root: ({ ownerState, theme }) => ({...}),
    },
  }
}
```

If you extend the interface via module augmentation like this:

```
declare module '@mui/material/Button' {
  interface ButtonPropsVariantOverrides {
    dashed: true;
  }
}
```

you will be able to see those props in `ownerState.variant` 🥕. `theme` can be augmented as well.

Experimental `sx` function

Initially, `sx` was designed to be a prop that enables you to inject styles with a shorthand notation to components created with the `styled` API:

```
import { styled } from '@mui/material/styles';
import Box from '@mui/material/Box';

const Label = styled('span')({
  fontWeight: 'bold',
  fontSize: '0.875rem',
})
```

```
<Box sx={{ display: 'flex' }}>
  <Label sx={{ color: 'text.secondary' }}>Label</Label>
</Box>;
```

> 💡 *All MUI components are created with the `styled` API, so they accept `sx` prop by default.*

`sx` helps developers write less code and be more productive once they are familiar with the API. With the callback support in `styleOverrides`, it is now possible to use an `sx`-like syntax in global theme overrides.

All you need is to use the `experimental_sx` function. In the following example, I use `sx` to theme the `Chip` component:

```
import {
  ThemeProvider,
  createTheme,
  experimental_sx as sx,
} from '@mui/material/styles';

<ThemeProvider
  theme={createTheme({
    components: {
      MuiChip: {
        styleOverrides: {
          root: sx({
            px: '12px', // shorthand for padding-left & right
            py: '6px', // shorthand for padding-top & bottom
            fontWeight: 500,
            borderRadius: '8px',
          }),
          label: {
            padding: 0,
          },
        },
      },
    },
  })}
>
  ...your app
</ThemeProvider>;
```

If I want to add more styles based on these conditions:

- border color `palette.text.secondary` is applied when `<Chip variant="outlined" />`.
- font size is `0.875rem` in mobile viewport, `0.75rem` in larger than mobile viewport when `<Chip size="small" />`.

An array can be used as a return type to make the code easier to add/remove conditions:

```
// The <ThemeProvider> is omitted for readability.
{
```

```
  root: ({ ownerState }) => [
    sx({
      px: '12px',
      py: '6px',
      fontWeight: 500,
      borderRadius: '8px',
    }),
    ownerState.variant === 'outlined' && ownerState.color === 'default' &&
      sx({
        borderColor: 'text.secondary',
      }),
    ownerState.size === 'small' &&
      sx({
        fontSize: { xs: '0.875rem', sm: '0.75rem' },
      })
  ],
}
```

That's it for today! Happy styling  .

I hope this small update makes your customization experience better than ever. Don't forget to share this update with your friends and colleagues.

To get more updates like this in the future, **subscribe to our newsletter** at the bottom of this page.

Read more

- Component theming
- All supported shorthands in `sx`
- `sx` performance tradeoff
- `sx` with `styled`