How to customize

You can easily customize the appearance of a MUI component.

As components can be used in different contexts, there are several approaches to customizing them. Going from the narrowest use-case to the broadest, these are:

1. One-off customization
2. Reusable style overrides
3. Dynamic variation
4. Global theme variation
5. Global CSS override

1. One-off customization

You might need to change the style of a component for a specific implementation, for which you have the following solutions available:

Use the `sx` prop

The easiest way to add style overrides for a one-off situation is to use the `sx` prop available on all MUI components. Here is an example:

{{"demo": "SxProp.js"}}

Next you'll see how you can use global class selectors for accessing slots inside the component. You'll also learn how to easily identify the classes which are available to you for each of the states and slots in the component.

Overriding nested component styles

You can use the browser dev tools to identify the slot for the component you want to override. It can save you a lot of time. The styles injected into the DOM by MUI rely on class names that follow a simple pattern: `[hash]-Mui[Component name]-[name of the slot]`.

⚠ These class names can't be used as CSS selectors because they are unstable, however, MUI applies global class names using a consistent convention: `Mui[Component name]-[name of the slot]`.

Let's go back to the above demo. How can you override the slider's thumb?

 dev-tools

In this example, the styles are applied with `.css-ae2u5c-MuiSlider-thumb` so the name of the component is `Slider` and the name of the slot is `thumb`.

You now know that you need to target the `.MuiSlider-thumb` class name for overriding the look of the thumb:

{{"demo": "DevTools.js"}}

Overriding styles with class names

If you would like to override the styles of the components using classes, you can use the `className` prop available on each component. For overriding the styles of the different parts inside the component, you can use the global classes available for each slot, as described in the previous section.

You can find examples of this using different styles libraries in the Styles library interoperability guide.

State classes

The components special states, like *hover*, *focus*, *disabled* and *selected*, are styled with a higher CSS specificity. Specificity is a weight that is applied to a given CSS declaration.

In order to override the components' special states, **you need to increase specificity**. Here is an example with the *disable* state and the Button component using a pseudo-class (`:disabled`):

```
.Button {
  color: black;
}

/* Increase the specificity */
.Button:disabled {
  color: white;
}
```

```
<Button disabled className="Button">
```

Sometimes, you can't use a CSS pseudo-class, as the state doesn't exist in the web specification. Let's take the MenuItem component and its *selected* state as an example. In such cases you can use a MUI equivalent of CSS pseudo-classes - **state classes**. Target the `.Mui-selected` global class name to customize the special state of the `MenuItem` component:

```
.MenuItem {
  color: black;
}

/* Increase the specificity */
.MenuItem.Mui-selected {
  color: blue;
}
```

```
<MenuItem selected className="MenuItem">
```

Why do I need to increase specificity to override one component state?

By design, the CSS specification makes the pseudo-classes increase the specificity. For consistency with native elements, MUI increases the specificity of its custom state classes. This has one important advantage, it allows you to cherry-pick the state you want to customize.

What custom state classes are available in MUI?

You can rely on the following global class names generated by MUI:

State	Global class name
active	`.Mui-active`

checked	`.Mui-checked`
completed	`.Mui-completed`
disabled	`.Mui-disabled`
error	`.Mui-error`
expanded	`.Mui-expanded`
focus visible	`.Mui-focusVisible`
focused	`.Mui-focused`
required	`.Mui-required`
selected	`.Mui-selected`

> ⚠️ *Never style these state classes' names directly:*

```
/* ❌ NOT OK, impact all the components with unclear side-effects */
.Mui-error {
  color: red;
}

/* ✅ OK */
.MuiOutlinedInput-root.Mui-error {
  color: red;
}
```

2. Reusable style overrides

If you find that you need the same overrides in multiple places across your application, you can use the `styled()` utility to create a reusable component:

{{"demo": "StyledCustomization.js", "defaultCodeOpen": true}}

With it, you have access to all of a component's props to dynamically style the component.

3. Dynamic variation

In the previous section, we learned how to override the style of a MUI component. Now, let's see how we can make these overrides dynamic. Here are four alternatives; each has its pros and cons.

Dynamic CSS

Using the `styled()` utility offers a simple way for adding dynamic styles based on props.

{{"demo": "DynamicCSS.js", "defaultCodeOpen": false}}

> ⚠️ *Note that if you are using TypeScript you will need to update the prop's types of the new component.*

```
import * as React from 'react';
import { styled } from '@mui/material/styles';
```

```tsx
import Slider, { SliderProps } from '@mui/material/Slider';

interface StyledSliderProps extends SliderProps {
  success?: boolean;
}

const StyledSlider = styled(Slider, {
  shouldForwardProp: (prop) => prop !== 'success',
})<StyledSliderProps>(({ success, theme }) => ({
  ...(success &&
    {
      // the overrides added when the new prop is used
    }),
}));
```

CSS variables

{{"demo": "DynamicCSSVariables.js"}}

4. Global theme variation

In order to promote consistency between components, and manage the user interface appearance as a whole, MUI provides a mechanism to apply global changes.

Please take a look at the theme's global overrides page for more details.

5. Global CSS override

Components expose global class names to enable customization with CSS.

```css
.MuiButton-root {
  font-size: 1rem;
}
```

You can reference the Styles library interoperability guide to find examples of this using different styles libraries or plain CSS.

If you just want to add some global baseline styles for some of the HTML elements, you can use the `GlobalStyles` component. Here is an example of how you can override styles for the `h1` elements.

{{"demo": "GlobalCssOverride.js", "iframe": true, "height": 100}}

If you are already using the CssBaseline component for setting baseline styles, you can also add these global styles as overrides for this component. Here is how you can achieve the same by using this approach.

{{"demo": "OverrideCssBaseline.js", "iframe": true, "height": 100}}

> *Note: It is a good practice to hoist the `<GlobalStyles />` to a static constant, to avoid rerendering. This will ensure that the `<style>` tag generated would not recalculate on each render.*

```tsx
import * as React from 'react';
import GlobalStyles from '@mui/material/GlobalStyles';
```

```
+const inputGlobalStyles = <GlobalStyles styles={...} />;

 const Input = (props) => {
   return (
     <React.Fragment>
-      <GlobalStyles styles={...} />
+      {inputGlobalStyles}
       <input {...props} />
     </React.Fragment>
   )
 }
```