Advanced (LEGACY)

This section covers more advanced usage of @mui/styles.

> ⚠️ *`@mui/styles` is the **legacy** styling solution for MUI. It depends on JSS as a styling solution, which is not used in the `@mui/material` anymore, deprecated in v5. If you don't want to have both emotion & JSS in your bundle, please refer to the `@mui/system` documentation which is the recommended alternative.*

> ⚠️ *`@mui/styles` is not compatible with React.StrictMode or React 18.*

Theming

Add a `ThemeProvider` to the top level of your app to pass a theme down the React component tree. Then, you can access the theme object in style functions.

> *This example creates a theme object for custom-built components. If you intend to use some of the MUI's components you need to provide a richer theme structure using the `createTheme()` method. Head to the theming section to learn how to build your custom MUI theme.*

```
import { ThemeProvider } from '@mui/styles';
import DeepChild from './my_components/DeepChild';

const theme = {
  background: 'linear-gradient(45deg, #FE6B8B 30%, #FF8E53 90%)',
};

function Theming() {
  return (
    <ThemeProvider theme={theme}>
      <DeepChild />
    </ThemeProvider>
  );
}
```

{{"demo": "Theming.js"}}

Accessing the theme in a component

You might need to access the theme variables inside your React components.

`useTheme` hook

For use in function components:

```
import { useTheme } from '@mui/styles';

function DeepChild() {
  const theme = useTheme();
  return <span>{`spacing ${theme.spacing}`}</span>;
}
```

{{"demo": "UseTheme.js"}}

`withTheme` HOC

For use in class or function components:

```
import { withTheme } from '@mui/styles';

function DeepChildRaw(props) {
  return <span>{`spacing ${props.theme.spacing}`}</span>;
}

const DeepChild = withTheme(DeepChildRaw);
```

{{"demo": "WithTheme.js"}}

Theme nesting

You can nest multiple theme providers. This can be really useful when dealing with different areas of your application that have distinct appearance from each other.

```
<ThemeProvider theme={outerTheme}>
  <Child1 />
  <ThemeProvider theme={innerTheme}>
    <Child2 />
  </ThemeProvider>
</ThemeProvider>
```

{{"demo": "ThemeNesting.js"}}

The inner theme will **override** the outer theme. You can extend the outer theme by providing a function:

```
<ThemeProvider theme={…} >
  <Child1 />
  <ThemeProvider theme={outerTheme => ({ darkMode: true, ...outerTheme })}>
    <Child2 />
  </ThemeProvider>
</ThemeProvider>
```

Overriding styles - `classes` prop

The `makeStyles` (hook generator) and `withStyles` (HOC) APIs allow the creation of multiple style rules per style sheet. Each style rule has its own class name. The class names are provided to the component with the `classes` variable. This is particularly useful when styling nested elements in a component.

```
// A style sheet
const useStyles = makeStyles({
  root: {}, // a style rule
  label: {}, // a nested style rule
});
```

```
function Nested(props) {
  const classes = useStyles();
  return (
    <button className={classes.root}>
      {/* 'jss1' */}
      <span className={classes.label}>{/* 'jss2' nested */}</span>
    </button>
  );
}

function Parent() {
  return <Nested />;
}
```

However, the class names are often non-deterministic. How can a parent component override the style of a nested element?

`withStyles`

This is the simplest case. The wrapped component accepts a `classes` prop, it simply merges the class names provided with the style sheet.

```
const Nested = withStyles({
  root: {}, // a style rule
  label: {}, // a nested style rule
})(({ classes }) => (
  <button className={classes.root}>
    <span className={classes.label}>{/* 'jss2 my-label' Nested*/}</span>
  </button>
));

function Parent() {
  return <Nested classes={{ label: 'my-label' }} />;
}
```

`makeStyles`

The hook API requires a bit more work. You have to forward the parent props to the hook as a first argument.

```
const useStyles = makeStyles({
  root: {}, // a style rule
  label: {}, // a nested style rule
});

function Nested(props) {
  const classes = useStyles(props);
  return (
    <button className={classes.root}>
      <span className={classes.label}>{/* 'jss2 my-label' nested */}</span>
    </button>
```

```
  );
}

function Parent() {
  return <Nested classes={{ label: 'my-label' }} />;
}
```

JSS plugins

JSS uses plugins to extend its core, allowing you to cherry-pick the features you need, and only pay the performance overhead for what you are using.

Not all the plugins are available in MUI by default. The following (which is a subset of jss-preset-default) are included:

- jss-plugin-rule-value-function
- jss-plugin-global
- jss-plugin-nested
- jss-plugin-camel-case
- jss-plugin-default-unit
- jss-plugin-vendor-prefixer
- jss-plugin-props-sort

Of course, you are free to use additional plugins. Here is an example with the jss-rtl plugin.

```
import { create } from 'jss';
import { StylesProvider, jssPreset } from '@mui/styles';
import rtl from 'jss-rtl';

const jss = create({
  plugins: [...jssPreset().plugins, rtl()],
});

export default function App() {
  return <StylesProvider jss={jss}>...</StylesProvider>;
}
```

String templates

If you prefer CSS syntax over JSS, you can use the jss-plugin-template plugin.

```
const useStyles = makeStyles({
  root: `
    background: linear-gradient(45deg, #fe6b8b 30%, #ff8e53 90%);
    border-radius: 3px;
    font-size: 16px;
    border: 0;
    color: white;
    height: 48px;
    padding: 0 30px;
    box-shadow: 0 3px 5px 2px rgba(255, 105, 135, 0.3);
```

```
  `,
});
```

Note that this doesn't support selectors, or nested rules.

{{"demo": "StringTemplates.js"}}

CSS injection order

> *It's **really important** to understand how the CSS specificity is calculated by the browser, as it's one of the key elements to know when overriding styles. You are encouraged to read this MDN paragraph: [How is specificity calculated?]*

By default, the style tags are injected **last** in the `<head>` element of the page. They gain more specificity than any other style tags on your page e.g. CSS modules, styled components.

injectFirst

The `StylesProvider` component has an `injectFirst` prop to inject the style tags **first** in the head (less priority):

```
import { StylesProvider } from '@mui/styles';

<StylesProvider injectFirst>
  {/* Your component tree.
      Styled components can override MUI's styles. */}
</StylesProvider>;
```

`makeStyles` / `withStyles` / `styled`

The injection of style tags happens in the **same order** as the `makeStyles` / `withStyles` / `styled` invocations. For instance the color red wins in this case:

```
import clsx from 'clsx';
import { makeStyles } from '@mui/styles';

const useStylesBase = makeStyles({
  root: {
    color: 'blue', // 🔵
  },
});

const useStyles = makeStyles({
  root: {
    color: 'red', // 🔴
  },
});

export default function MyComponent() {
  // Order doesn't matter
  const classes = useStyles();
```

```
  const classesBase = useStylesBase();

  // Order doesn't matter
  const className = clsx(classes.root, classesBase.root);

  // color: red 🔴 wins.
  return <div className={className} />;
}
```

The hook call order and the class name concatenation order **don't matter**.

insertionPoint

JSS provides a mechanism to control this situation. By adding an `insertionPoint` within the HTML you can control the order that the CSS rules are applied to your components.

HTML comment

The simplest approach is to add an HTML comment to the `<head>` that determines where JSS will inject the styles:

```
<head>
  <!-- jss-insertion-point -->
  <link href="..." />
</head>
```

```
import { create } from 'jss';
import { StylesProvider, jssPreset } from '@mui/styles';

const jss = create({
  ...jssPreset(),
  // Define a custom insertion point that JSS will look for when injecting the
styles into the DOM.
  insertionPoint: 'jss-insertion-point',
});

export default function App() {
  return <StylesProvider jss={jss}>...</StylesProvider>;
}
```

Other HTML elements

Create React App strips HTML comments when creating the production build. To get around this issue, you can provide a DOM element (other than a comment) as the JSS insertion point, for example, a `<noscript>` element:

```
<head>
  <noscript id="jss-insertion-point" />
  <link href="..." />
</head>
```

```
import { create } from 'jss';
import { StylesProvider, jssPreset } from '@mui/styles';

const jss = create({
  ...jssPreset(),
  // Define a custom insertion point that JSS will look for when injecting the
styles into the DOM.
  insertionPoint: document.getElementById('jss-insertion-point'),
});

export default function App() {
  return <StylesProvider jss={jss}>...</StylesProvider>;
}
```

JS createComment

codesandbox.io prevents access to the `<head>` element. To get around this issue, you can use the JavaScript `document.createComment()` API:

```
import { create } from 'jss';
import { StylesProvider, jssPreset } from '@mui/styles';

const styleNode = document.createComment('jss-insertion-point');
document.head.insertBefore(styleNode, document.head.firstChild);

const jss = create({
  ...jssPreset(),
  // Define a custom insertion point that JSS will look for when injecting the
styles into the DOM.
  insertionPoint: 'jss-insertion-point',
});

export default function App() {
  return <StylesProvider jss={jss}>...</StylesProvider>;
}
```

Server-side rendering

This example returns a string of HTML and inlines the critical CSS required, right before it's used:

```
import ReactDOMServer from 'react-dom/server';
import { ServerStyleSheets } from '@mui/styles';

function render() {
  const sheets = new ServerStyleSheets();

  const html = ReactDOMServer.renderToString(sheets.collect(<App />));
  const css = sheets.toString();

  return `
```

```
<!DOCTYPE html>
<html>
  <head>
    <style id="jss-server-side">${css}</style>
  </head>
  <body>
    <div id="root">${html}</div>
  </body>
</html>
  `;
}
```

You can follow the server side guide for a more detailed example, or read the `ServerStyleSheets` API documentation.

Gatsby

There is an official Gatsby plugin that enables server-side rendering for `@mui/styles`. Refer to the plugin's page for setup and usage instructions.

Refer to this example Gatsby project for an up-to-date usage example.

Next.js

You need to have a custom `pages/_document.js`, then copy this logic to inject the server-side rendered styles into the `<head>` element.

Refer to this example project for an up-to-date usage example.

Class names

The class names are generated by the class name generator.

Default

By default, the class names generated by `@mui/styles` are **non-deterministic**; you can't rely on them to stay the same. Let's take the following style as an example:

```
const useStyles = makeStyles({
  root: {
    opacity: 1,
  },
});
```

This will generate a class name such as `makeStyles-root-123`.

You have to use the `classes` prop of a component to override the styles. The non-deterministic nature of the class names enables style isolation.

- In **development**, the class name is: `.makeStyles-root-123`, following this logic:

```
const sheetName = 'makeStyles';
const ruleName = 'root';
const identifier = 123;

const className = `${sheetName}-${ruleName}-${identifier}`;
```

- In **production**, the class name is: `.jss123`, following this logic:

```
const productionPrefix = 'jss';
const identifier = 123;

const className = `${productionPrefix}-${identifier}`;
```

However, when the following conditions are met, the class names are **deterministic**:

- Only one theme provider is used (**No theme nesting**)
- The style sheet has a name that starts with `Mui` (all MUI components).
- The `disableGlobal` option of the class name generator is `false` (the default).

Global CSS

`jss-plugin-global`

The `jss-plugin-global` plugin is installed in the default preset. You can use it to define global class names.

{{"demo": "GlobalCss.js"}}

Hybrid

You can also combine JSS generated class names with global ones.

{{"demo": "HybridGlobalCss.js"}}

CSS prefixes

JSS uses feature detection to apply the correct prefixes. Don't be surprised if you can't see a specific prefix in the latest version of Chrome. Your browser probably doesn't need it.

TypeScript usage

Using `withStyles` in TypeScript can be a little tricky, but there are some utilities to make the experience as painless as possible.

Using `createStyles` to defeat type widening

A frequent source of confusion is TypeScript's type widening, which causes this example not to work as expected:

```
const styles = {
  root: {
    display: 'flex',
    flexDirection: 'column',
```

```
  },
};

withStyles(styles);
//         ^^^^^^
//         Types of property 'flexDirection' are incompatible.
//           Type 'string' is not assignable to type '"-moz-initial" | "inherit" |
"initial" | "revert" | "unset" | "column" | "column-reverse" | "row"...'.
```

The problem is that the type of the `flexDirection` prop is inferred as `string`, which is too wide. To fix this, you can pass the styles object directly to `withStyles`:

```
withStyles({
  root: {
    display: 'flex',
    flexDirection: 'column',
  },
});
```

However type widening rears its ugly head once more if you try to make the styles depend on the theme:

```
withStyles(({ palette, spacing }) => ({
  root: {
    display: 'flex',
    flexDirection: 'column',
    padding: spacing.unit,
    backgroundColor: palette.background.default,
    color: palette.primary.main,
  },
}));
```

This is because TypeScript widens the return types of function expressions.

Because of this, using the `createStyles` helper function to construct your style rules object is recommended:

```
// Non-dependent styles
const styles = createStyles({
  root: {
    display: 'flex',
    flexDirection: 'column',
  },
});

// Theme-dependent styles
const styles = ({ palette, spacing }: Theme) =>
  createStyles({
    root: {
      display: 'flex',
      flexDirection: 'column',
      padding: spacing.unit,
```

```
      backgroundColor: palette.background.default,
      color: palette.primary.main,
    },
  });
```

`createStyles` is just the identity function; it doesn't "do anything" at runtime, just helps guide type inference at compile time.

Media queries

`withStyles` allows a styles object with top level media-queries like so:

```
const styles = createStyles({
  root: {
    minHeight: '100vh',
  },
  '@media (min-width: 960px)': {
    root: {
      display: 'flex',
    },
  },
});
```

To allow these styles to pass TypeScript however, the definitions have to be unambiguous concerning the names for CSS classes and actual CSS property names. Due to this, class names that are equal to CSS properties should be avoided.

```
// error because TypeScript thinks `@media (min-width: 960px)` is a class name
// and `content` is the CSS property
const ambiguousStyles = createStyles({
  content: {
    minHeight: '100vh',
  },
  '@media (min-width: 960px)': {
    content: {
      display: 'flex',
    },
  },
});

// works just fine
const ambiguousStyles = createStyles({
  contentClass: {
    minHeight: '100vh',
  },
  '@media (min-width: 960px)': {
    contentClass: {
      display: 'flex',
    },
  },
});
```

Augmenting your props using `WithStyles`

Since a component decorated with `withStyles(styles)` gets a special `classes` prop injected, you will want to define its props accordingly:

```
const styles = (theme: Theme) =>
  createStyles({
    root: {
      /* ... */
    },
    paper: {
      /* ... */
    },
    button: {
      /* ... */
    },
  });

interface Props {
  // non-style props
  foo: number;
  bar: boolean;
  // injected style props
  classes: {
    root: string;
    paper: string;
    button: string;
  };
}
```

However this isn't very DRY because it requires you to maintain the class names (`'root'` , `'paper'` , `'button'` , ...) in two different places. We provide a type operator `WithStyles` to help with this, so that you can just write:

```
import { createStyles, WithStyles } from '@mui/styles';

const styles = (theme: Theme) =>
  createStyles({
    root: {
      /* ... */
    },
    paper: {
      /* ... */
    },
    button: {
      /* ... */
    },
  });

interface Props extends WithStyles<typeof styles> {
```

```
  foo: number;
  bar: boolean;
}
```

Decorating components

Applying `withStyles(styles)` as a function works as expected:

```
const DecoratedSFC = withStyles(styles)(({ text, type, color, classes }: Props) => (
  <Typography variant={type} color={color} classes={classes}>
    {text}
  </Typography>
));

const DecoratedClass = withStyles(styles)(
  class extends React.Component<Props> {
    render() {
      const { text, type, color, classes } = this.props;
      return (
        <Typography variant={type} color={color} classes={classes}>
          {text}
        </Typography>
      );
    }
  },
);
```

Unfortunately due to a [current limitation of TypeScript decorators](), `withStyles(styles)` can't be used as a decorator in TypeScript.