Frequently Asked Questions

Stuck on a particular problem? Check some of these common gotchas first in the FAQ.

If you still can't find what you're looking for, you can refer to our support page.

MUI is awesome. How can I support the project?

There are many ways to support MUI:

- **Spread the word**. Evangelize MUI by linking to mui.com on your website, every backlink matters. Follow us on Twitter, like and retweet the important news. Or just talk about us with your friends.
- **Give us feedback**. Tell us what we're doing well or where we can improve. Please upvote (👍) the issues that you are the most interested in seeing solved.
- **Help new users**. You can answer questions on StackOverflow.
- **Make changes happen**.
  - Edit the documentation. Every page has an "EDIT THIS PAGE" link in the top right.
  - Report bugs or missing features by creating an issue.
  - Review and comment on existing pull requests and issues.
  - Help translate the documentation.
  - Improve our documentation, fix bugs, or add features by submitting a pull request.
- **Support us financially on OpenCollective**. If you use MUI in a commercial project and would like to support its continued development by becoming a Sponsor, or in a side or hobby project and would like to become a Backer, you can do so through OpenCollective. All funds donated are managed transparently, and Sponsors receive recognition in the README and on the MUI home page.

Why do the fixed positioned elements move when a modal is opened?

Scrolling is blocked as soon as a modal is opened. This prevents interacting with the background when the modal should be the only interactive content. However, removing the scrollbar can make your **fixed positioned elements** move. In this situation, you can apply a global `.mui-fixed` class name to tell MUI to handle those elements.

How can I disable the ripple effect globally?

The ripple effect is exclusively coming from the `BaseButton` component. You can disable the ripple effect globally by providing the following in your theme:

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

const theme = createTheme({
  components: {
    // Name of the component ⚛️
    MuiButtonBase: {
      defaultProps: {
        // The props to apply
        disableRipple: true, // No more ripple, on the whole application 💣!
      },
    },
  },
});
```

How can I disable transitions globally?

MUI uses the same theme helper for creating all its transitions. Therefore you can disable all transitions by overriding the helper in your theme:

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

const theme = createTheme({
  transitions: {
    // So we have `transition: none;` everywhere
    create: () => 'none',
  },
});
```

It can be useful to disable transitions during visual testing or to improve performance on low-end devices.

You can go one step further by disabling all transitions and animations effects:

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

const theme = createTheme({
  components: {
    // Name of the component ⚛️
    MuiCssBaseline: {
      styleOverrides: {
        '*, *::before, *::after': {
          transition: 'none !important',
          animation: 'none !important',
        },
      },
    },
  },
});
```

Notice that the usage of `CssBaseline` is required for the above approach to work. If you choose not to use it, you can still disable transitions and animations by including these CSS rules:

```
*,
*::before,
*::after {
  transition: 'none !important';
  animation: 'none !important';
}
```

Do I have to use emotion to style my app?

No, it's not required. But if you are using the default styled engine (`@mui/styled-engine`) the emotion dependency comes built in, so carries no additional bundle size overhead.

Perhaps, however, you're adding some MUI components to an app that already uses another styling solution, or are already familiar with a different API, and don't want to learn a new one? In that case, head over to the Style Library Interoperability section, where we show how simple it is to restyle MUI components with alternative style libraries.

When should I use inline-style vs. CSS?

As a rule of thumb, only use inline-styles for dynamic style properties. The CSS alternative provides more advantages, such as:

- auto-prefixing
- better debugging
- media queries
- keyframes

How do I use react-router?

We detail the integration with third-party routing libraries like react-router, Gatsby or Next.js in our guide.

How can I access the DOM element?

All MUI components that should render something in the DOM forward their ref to the underlying DOM component. This means that you can get DOM elements by reading the ref attached to MUI components:

```
// or a ref setter function
const ref = React.createRef();
// render
<Button ref={ref} />;
// usage
const element = ref.current;
```

If you're not sure if the MUI component in question forwards its ref you can check the API documentation under "Props" e.g. the Button API includes

> *The ref is forwarded to the root element.*

indicating that you can access the DOM element with a ref.

I have several instances of styles on the page

If you are seeing a warning message in the console like the one below, you probably have several instances of `@mui/styles` initialized on the page.

> *It looks like there are several instances of `@mui/styles` initialized in this application. This may cause theme propagation issues, broken class names, specificity issues, and make your application bigger without a good reason.*

Possible reasons

There are several common reasons for this to happen:

- You have another `@mui/styles` library somewhere in your dependencies.
- You have a monorepo structure for your project (e.g, lerna, yarn workspaces) and `@mui/styles` module is a dependency in more than one package (this one is more or less the same as the previous one).

- You have several applications that are using `@mui/styles` running on the same page (e.g., several entry points in webpack are loaded on the same page).

Duplicated module in node_modules

If you think that the issue may be in the duplication of the @mui/styles module somewhere in your dependencies, there are several ways to check this. You can use `npm ls @mui/styles`, `yarn list @mui/styles` or `find -L ./node_modules | grep /@mui/styles/package.json` commands in your application folder.

If none of these commands identified the duplication, try analyzing your bundle for multiple instances of @mui/styles. You can just check your bundle source, or use a tool like source-map-explorer or webpack-bundle-analyzer.

If you identified that duplication is the issue that you are encountering there are several things you can try to solve it:

If you are using npm you can try running `npm dedupe`. This command searches the local dependencies and tries to simplify the structure by moving common dependencies further up the tree.

If you are using webpack, you can change the way it will resolve the @mui/styles module. You can overwrite the default order in which webpack will look for your dependencies and make your application node_modules more prioritized than default node module resolution order:

```diff
  resolve: {
+   alias: {
+     "@mui/styles": path.resolve(appFolder, "node_modules", "@mui/styles"),
+   }
  }
```

Usage with Lerna

One possible fix to get @mui/styles to run in a Lerna monorepo across packages is to hoist shared dependencies to the root of your monorepo file. Try running the bootstrap option with the --hoist flag.

```
lerna bootstrap --hoist
```

Alternatively, you can remove @mui/styles from your package.json file and hoist it manually to your top-level package.json file.

Example of a package.json file in a Lerna root folder

```
{
  "name": "my-monorepo",
  "devDependencies": {
    "lerna": "latest"
  },
  "dependencies": {
    "@mui/styles": "^4.0.0"
  },
  "scripts": {
    "bootstrap": "lerna bootstrap",
    "clean": "lerna clean",
    "start": "lerna run start",
    "build": "lerna run build"
  }
}
```

Running multiple applications on one page

If you have several applications running on one page, consider using one @mui/styles module for all of them. If you are using webpack, you can use [CommonsChunkPlugin](https://webpack.js.org/plugins/commons-chunk-plugin/) to create an explicit [vendor chunk](https://webpack.js.org/plugins/commons-chunk-plugin/#explicit-vendor-chunk), that will contain the @mui/styles module:

```diff
 module.exports = {
   entry: {
+    vendor: ["@mui/styles"],
     app1: "./src/app.1.js",
     app2: "./src/app.2.js",
   },
   plugins: [
+    new webpack.optimize.CommonsChunkPlugin({
+      name: "vendor",
+      minChunks: Infinity,
+    }),
   ]
 }
```

My App doesn't render correctly on the server

If it doesn't work, in 99% of cases it's a configuration issue. A missing property, a wrong call order, or a missing component – server-side rendering is strict about configuration.

The best way to find out what's wrong is to compare your project to an **already working setup**. Check out the [reference implementations](/material-ui/guides/server-rendering/#reference-implementations), bit by bit.

Why are the colors I am seeing different from what I see here?

The documentation site is using a custom theme. Hence, the color palette is different from the default theme that MUI ships. Please refer to [this page](/material-ui/customization/theming/) to learn about theme customization.

Why does component X require a DOM node in a prop instead of a ref object?

Components like the [Portal](/material-ui/api/portal/#props) or [Popper](/material-ui/api/popper/#props) require a DOM node in the `container` or `anchorEl` prop respectively. It seems convenient to simply pass a ref object in those props and let MUI access the current value. This works in a simple scenario:

```jsx
function App() {
  const container = React.useRef(null);

  return (
    <div className="App">
      <Portal container={container}>
```

```
        <span>portaled children</span>
      </Portal>
      <div ref={container} />
    </div>
  );
}
```

where `Portal` would only mount the children into the container when `container.current` is available. Here is a naive implementation of Portal:

```
function Portal({ children, container }) {
  const [node, setNode] = React.useState(null);

  React.useEffect(() => {
    setNode(container.current);
  }, [container]);

  if (node === null) {
    return null;
  }
  return ReactDOM.createPortal(children, node);
}
```

With this simple heuristic `Portal` might re-render after it mounts because refs are up-to-date before any effects run. However, just because a ref is up-to-date doesn't mean it points to a defined instance. If the ref is attached to a ref forwarding component it is not clear when the DOM node will be available. In the example above, the `Portal` would run an effect once, but might not re-render because `ref.current` is still `null`. This is especially apparent for React.lazy components in Suspense. The above implementation could also not account for a change in the DOM node.

This is why we require a prop with the actual DOM node so that React can take care of determining when the `Portal` should re-render:

```
function App() {
  const [container, setContainer] = React.useState(null);
  const handleRef = React.useCallback(
    (instance) => setContainer(instance),
    [setContainer],
  );

  return (
    <div className="App">
      <Portal container={container}>
        <span>Portaled</span>
      </Portal>
      <div ref={handleRef} />
    </div>
  );
}
```

What's the clsx dependency for?

clsx is a tiny utility for constructing `className` strings conditionally, out of an object with keys being the class strings, and values being booleans.

Instead of writing:

```
// let disabled = false, selected = true;

return (
  <div
    className={`MuiButton-root ${disabled ? 'Mui-disabled' : ''} ${
      selected ? 'Mui-selected' : ''
    }`}
  />
);
```

you can do:

```
import clsx from 'clsx';

return (
  <div
    className={clsx('MuiButton-root', {
      'Mui-disabled': disabled,
      'Mui-selected': selected,
    })}
  />
);
```

I cannot use components as selectors in the styled() utility. What should I do?

If you are getting the error: `TypeError: Cannot convert a Symbol value to a string`, take a look at the styled() docs page for instructions on how you can fix this.

[v4] Why aren't my components rendering correctly in production builds?

The #1 reason this happens is likely due to class name conflicts once your code is in a production bundle. For MUI to work, the `className` values of all components on a page must be generated by a single instance of the class name generator.

To correct this issue, all components on the page need to be initialized such that there is only ever **one class name generator** among them.

You could end up accidentally using two class name generators in a variety of scenarios:

- You accidentally **bundle** two versions of MUI. You might have a dependency not correctly setting MUI as a peer dependency.

- You are using `StylesProvider` for a **subset** of your React tree.
- You are using a bundler and it is splitting code in a way that causes multiple class name generator instances to be created.

> *If you are using webpack with the [SplitChunksPlugin](), try configuring the [`runtimeChunk` setting under `optimizations`]().*

Overall, it's simple to recover from this problem by wrapping each MUI application with [`StylesProvider`]() components at the top of their component trees **and using a single class name generator shared among them**.

[v4] CSS works only on first load and goes missing

The CSS is only generated on the first load of the page. Then, the CSS is missing on the server for consecutive requests.

Action to Take

The styling solution relies on a cache, the *sheets manager*, to only inject the CSS once per component type (if you use two buttons, you only need the CSS of the button one time). You need to create **a new `sheets` instance for each request**.

Example of fix:

```diff
-// Create a sheets instance.
-const sheets = new ServerStyleSheets();

function handleRender(req, res) {
+ // Create a sheets instance.
+ const sheets = new ServerStyleSheets();

  //…

  // Render the component to a string.
  const html = ReactDOMServer.renderToString(
```

[v4] React class name hydration mismatch

> *Warning: Prop className did not match.*

There is a class name mismatch between the client and the server. It might work for the first request. Another symptom is that the styling changes between initial page load and the downloading of the client scripts.

Action to Take

The class names value relies on the concept of [class name generator](). The whole page needs to be rendered with **a single generator**. This generator needs to behave identically on the server and on the client. For instance:

- You need to provide a new class name generator for each request. But you shouldn't share a `createGenerateClassName()` between different requests:

  Example of fix:

  ```diff
  -// Create a new class name generator.
  -const generateClassName = createGenerateClassName();
  ```

```
function handleRender(req, res) {
+ // Create a new class name generator.
+ const generateClassName = createGenerateClassName();

  //…

  // Render the component to a string.
  const html = ReactDOMServer.renderToString(
```

- You need to verify that your client and server are running the **exactly the same version** of MUI. It is possible that a mismatch of even minor versions can cause styling problems. To check version numbers, run `npm list @mui/material` in the environment where you build your application and also in your deployment environment.

  You can also ensure the same version in different environments by specifying a specific MUI version in the dependencies of your package.json.

  example of fix (package.json):

```
  "dependencies": {
    ...
-   "@mui/material": "^4.0.0",
+   "@mui/material": "4.0.0",
    ...
  },
```

- You need to make sure that the server and the client share the same `process.env.NODE_ENV` value.