MUI System

CSS utilities for rapidly laying out custom designs.

MUI comes with dozens of **ready-to-use** components in the core. These components are an incredible starting point but when it comes to making your site stand out with a custom design, it can be simpler to start from an unstyled state. Introducing the system:

The **system** lets you quickly build custom UI components leveraging the values defined in your theme.

Demo

(Resize the window to see the responsive breakpoints)

{{"demo": "Demo.js", "bg": true, "defaultCodeOpen": true}}

Installation

```
// with npm
npm install @mui/system @emotion/react @emotion/styled

// with yarn
yarn add @mui/system @emotion/react @emotion/styled
```

Or if you want to use `styled-components` as a styling engine:

```
// with npm
npm install @mui/system @mui/styled-engine-sc styled-components

// with yarn
yarn add @mui/system @mui/styled-engine-sc styled-components
```

Take a look at the [Styled Engine guide]() for more information about how to configure `styled-components` as the style engine.

Why use the system?

Compare how the same stat component can be built with two different APIs.

{{"demo": "Why.js", "bg": true, "defaultCodeOpen": false}}

1. ❌ using the styled-components's API:

```
const StatWrapper = styled('div')(
  ({ theme }) => `
  background-color: ${theme.palette.background.paper};
  box-shadow: ${theme.shadows[1]};
  border-radius: ${theme.shape.borderRadius}px;
  padding: ${theme.spacing(2)};
  min-width: 300px;
`,
);

const StatHeader = styled('div')(
  ({ theme }) => `
  color: ${theme.palette.text.secondary};
`,
);

const StyledTrend = styled(TrendingUpIcon)(
  ({ theme }) => `
  color: ${theme.palette.success.dark};
  font-size: 16px;
  vertical-alignment: sub;
`,
);

const StatValue = styled('div')(
  ({ theme }) => `
  color: ${theme.palette.text.primary};
  font-size: 34px;
  font-weight: ${theme.typography.fontWeightMedium};
`,
);

const StatDiff = styled('div')(
  ({ theme }) => `
  color: ${theme.palette.success.dark};
  display: inline;
  font-weight: ${theme.typography.fontWeightMedium};
  margin-left: ${theme.spacing(0.5)};
  margin-right: ${theme.spacing(0.5)};
`,
);

const StatPrevious = styled('div')(
  ({ theme }) => `
  color: ${theme.palette.text.secondary};
  display: inline;
  font-size: 12px;
`,
);

return (
  <StatWrapper>
    <StatHeader>Sessions</StatHeader>
    <StatValue>98.3 K</StatValue>
    <StyledTrend />
    <StatDiff>18.77%</StatDiff>
    <StatPrevious>vs last week</StatPrevious>
  </StatWrapper>
);
```

2. ✅ using the system:

```
<Box
  sx={{
    bgcolor: 'background.paper',
    boxShadow: 1,
    borderRadius: 1,
    p: 2,
    minWidth: 300,
  }}
>
  <Box sx={{ color: 'text.secondary' }}>Sessions</Box>
  <Box sx={{ color: 'text.primary', fontSize: 34, fontWeight: 'medium' }}>
    98.3 K
  </Box>
  <Box
    component={TrendingUpIcon}
    sx={{ color: 'success.dark', fontSize: 16, verticalAlign: 'sub' }}
  />
  <Box
    sx={{
      color: 'success.dark',
      display: 'inline',
      fontWeight: 'medium',
      mx: 0.5,
    }}
  >
    18.77%
  </Box>
  <Box sx={{ color: 'text.secondary', display: 'inline', fontSize: 12 }}>
    vs. last week
  </Box>
</Box>
```

Problem solved

The system focus on solving 3 main problems:

1. Switching context wastes time.

There's no need to constantly jump between the usage of the styled components and where they are defined. With the system, those descriptions are right where you need them.

2. Naming things is hard.

Have you ever found yourself struggling to find a good name for a styled component? The system maps the styles directly to the element. All you have to do is worry about actual style properties.

3. Enforcing consistency in UIs is hard.

This is especially true when more than one person is building the application, as there has to be some coordination amongst members of the team regarding the choice of design tokens and how they are used, what parts of the theme structure should be used with what CSS properties, and so on.

The system provides direct access to the value in the theme. It makes it easier to design with constraints.

The `sx` prop

The `sx` prop, as the main part of the system, solves these problems by providing a fast & simple way of applying the correct design tokens for specific CSS properties directly to a React element. The demo above shows how it can be used to create a one-off design.

This prop provides a superset of CSS (contains all CSS properties/selectors in addition to custom ones) that maps values directly from the theme, depending on the CSS property used. Also, it allows a simple way of defining responsive values that correspond to the breakpoints defined in the theme. For more details, visit the `sx` prop page.

When to use it?

- **styled-components**: the API is great to build components that need to support a wide variety of contexts. These components are used in many different parts of the application and support different combinations of props.
- **`sx` prop**: the API is great to apply one-off styles. It's called "utility" for this reason.

Performance tradeoff

The system relies on CSS-in-JS. It works with both emotion and styled-components.

Pros:

- 📚 It allows a lot of flexibility in the API. The `sx` prop supports a superset of CSS. There is **no need to learn CSS twice**. You are set once you have learn the standardized CSS syntax, it's safe, it hasn't changed for a decade. Then, you can **optionally** learn the shorthands if you value the save of time they bring.
- 📦 Auto-purge. Only the used CSS on the page is sent to the client. The initial bundle size cost is **fixed**. It's not growing with the number of used CSS properties. You pay the cost of @emotion/react and @mui/system. It cost around ~15 kB gzipped. If you are already using the core components, it comes with no extra overhead.

Cons:

- The runtime performance takes a hit.

Benchmark case	Code snippet	Time normalized
a. Render 1,000 primitives	`<div className="…">`	100ms
b. Render 1,000 components	`<Div>`	120ms
c. Render 1,000 styled components	`<StyledDiv>`	160ms
d. Render 1,000 Box	`<Box sx={…}>`	370ms

Head to the benchmark folder for a reproduction of these metrics.

We believe that for most uses it's **fast enough**, but there are simple workarounds when performance becomes critical. For instance, when rendering a list with many items, you can use a CSS child selector to have a single "style injection" point (using d. for the wrapper and a. for each item).

API tradeoff

Having the system under one prop (`sx`) helps to differentiate props defined for the sole purpose of CSS utilities, vs. those for component business logic. It's important for the **separation of concerns**. For instance, a `color` prop on a button impacts multiple states (hover, focus, etc.), not to be confused with the color CSS property.

Only the `Box` , `Stack` , `Typography` , and `Grid` components accept the system properties as *props* for the above reason. These components are designed to solve CSS problems, they are CSS component utilities.

Usage

Design tokens in the theme

You can explore the System properties page to discover how the different CSS (and custom) properties are mapped to the theme keys.

Shorthands

There are lots of shorthands available for the CSS properties. These are documented in the next pages, for instance, the spacing. Here is an example leveraging them:

```
<Box
  sx={{
    boxShadow: 1, // theme.shadows[1]
    color: 'primary.main', // theme.palette.primary.main
    m: 1, // margin: theme.spacing(1)
    p: {
      xs: 1, // [theme.breakpoints.up('xs')]: { padding: theme.spacing(1) }
    },
    zIndex: 'tooltip', // theme.zIndex.tooltip
  }}
>
```

These shorthands are **optional**, they are great to save time when writing styles but it can be overwhelming to learn new custom APIs. You might want to skip this part and bet on CSS, it has been standardized for decades, head to the next section.

Superset of CSS

As part of the prop, you can use any regular CSS too: child or pseudo-selectors, media queries, raw CSS values, etc. Here are a few examples:

- Using pseudo selectors:

```
<Box
  sx={{
    // some styles
    ":hover": {
      boxShadow: 6,
    },
  }}
>
```

- Using media queries:

```
<Box
  sx={{
    // some styles
    '@media print': {
      width: 300,
    },
  }}
>
```

- Using nested selector:

```
<Box
  sx={{
    // some styles
    '& .ChildSelector': {
      bgcolor: 'primary.main',
    },
  }}
>
```

Responsive values

If you would like to have responsive values for a CSS property, you can use the breakpoints shorthand syntax. There are two ways of defining the breakpoints:

1. Breakpoints as an object

The first option for defining breakpoints is to define them as an object, using the breakpoints as keys. Note that each breakpoint property matches the breakpoint and every larger breakpoint. For example, `width: { lg: 100 }` is equivalent to `theme.breakpoints.up('lg')`. Here is the previous example again, using the object syntax.

{{"demo": "BreakpointsAsObject.js"}}

2. Breakpoints as an array

The second option is to define your breakpoints as an array, from the smallest to the largest breakpoint.

{{"demo": "BreakpointsAsArray.js"}}

> ⚠️ *This option is only recommended when the theme has a limited number of breakpoints, e.g. 3.*
> *Prefer the object API if you have more breakpoints. For instance, the default theme of MUI has 5.*

You can skip breakpoints with the `null` value:

```
<Box sx={{ width: [null, null, 300] }}>This box has a responsive width.</Box>
```

Custom breakpoints

You can also specify your own custom breakpoints, and use them as keys when defining the breakpoints object. Here is an example of how to do that.

```
import * as React from 'react';
import Box from '@mui/material/Box';
import { createTheme, ThemeProvider } from '@mui/material/styles';

const theme = createTheme({
  breakpoints: {
    values: {
      mobile: 0,
      tablet: 640,
      laptop: 1024,
      desktop: 1280,
    },
  },
});

export default function CustomBreakpoints() {
  return (
    <ThemeProvider theme={theme}>
      <Box
        sx={{
          width: {
            mobile: 100,
            laptop: 300,
          },
        }}
      >
        This box has a responsive width
      </Box>
    </ThemeProvider>
  );
}
```

If you are using TypeScript, you will also need to use module augmentation for the theme to accept the above values.

```
declare module '@mui/material/styles' {
  interface BreakpointOverrides {
    xs: false; // removes the `xs` breakpoint
    sm: false;
    md: false;
    lg: false;
    xl: false;
    tablet: true; // adds the `tablet` breakpoint
    laptop: true;
    desktop: true;
  }
}
```

Theme getter

If you wish to use the theme for a CSS property that is not supported natively by the system, you can use a function as the value, in which you can access the theme object.

{{"demo": "ValueAsFunction.js"}}

Implementations

The `sx` prop can be used in four different locations:

1. Core components

All core MUI components will support the `sx` prop.

2. Box

`Box` is a lightweight component that gives access to the `sx` prop, and can be used as a utility component, and as a wrapper for other components. It renders a `<div>` element by default.

3. Custom components

In addition to MUI components, you can add the `sx` prop to your custom components too, by using the `styled` utility from `@mui/material/styles` .

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

const Div = styled('div')``;
```

4. Any element with the babel plugin

TODO #23220.