Avançado

Esta seção aborda o uso mais avançado de @material-ui/core/styles.

> ***Note**: `@mui/styles` is the **legacy** styling solution for MUI. It is deprecated in v5. It depends on JSS as a styling solution, which is not used in the `@mui/material` anymore. 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.*

Temas

Adicione um `ThemeProvider` para o nível superior de sua aplicação para passar um tema pela árvore de componentes do React. Dessa maneira, você poderá acessar o objeto de tema em funções de estilo.

> *Este exemplo cria um objeto de tema para componentes customizados. If you intend to use some of the Material-UI's components you need to provide a richer theme structure using the `createTheme()` method. Vá até a seção de temas para aprender como construir seu tema customizado do Material-UI.*

```
import { ThemeProvider } from '@material-ui/core/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"}}

Acessando o tema em um componente

Você pode precisar acessar as variáveis de tema dentro de seus componentes React.

`useTheme` hook

Para uso em componentes de função:

```
import { useTheme } from '@material-ui/core/styles';

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

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

`withTheme` HOC

Para uso em componentes de classe ou função:

```
import { withTheme } from '@material-ui/core/styles';

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

const DeepChild = withTheme(DeepChildRaw);
```

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

Aninhamento de tema

Você pode aninhar vários provedores de tema. Isso pode ser muito útil ao lidar com diferentes áreas da sua aplicação que têm aparência distinta entre si.

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

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

O tema interno **sobrescreverá** o tema externo. Você pode estender o tema externo fornecendo uma função:

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

Sobrescrevendo estilos - propriedade `classes`

As APIs `makeStyles` (hook generator) e `withStyles` (HOC) permitem a criação de várias regras de estilos por folha de estilo. Cada regra de estilo tem seu próprio nome de classe. Os nomes das classes são fornecidos para o componente com a variável `classes`. Isso é particularmente útil ao estilizar elementos aninhados em um componente.

```
// Uma folha de estilo
const useStyles = makeStyles({
  root: {}, // uma regra de estilo
  label: {}, // uma regra de estilo aninhada
});
```

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

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

No entanto, os nomes das classes geralmente não são determinísticos. Como um componente pai pode substituir o estilo de um elemento aninhado?

`withStyles`

Esta é a maneira mais simples. O componente encapsulado aceita a propriedade `classes`, ele simplesmente mescla os nomes de classes fornecidos com a folha de estilo.

```
const Nested = withStyles({
  root: {}, // uma regra de estilo
  label: {}, // uma regra de estilo aninhada
})(({ classes }) => (
  <button className={classes.root}>
    <span className={classes.label}> // 'jss2 my-label' Aninhado</span>
  </button>
));

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

`makeStyles`

A API hook requer um pouco mais de trabalho. Você tem que encaminhar as propriedades do pai para o hook como primeiro argumento.

```
const useStyles = makeStyles({
  root: {}, // uma regra de estilo
  label: {}, // uma regra de estilo aninhada
});

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' }} />;
}
```

Plugins JSS

JSS usa plugins para estender sua essência, permitindo que você escolha os recursos que você precisa, e somente pague pela sobrecarga de desempenho para o que você está usando.

Nem todos os plugins estão disponíveis por padrão no Material-UI. Os seguintes (que são um subconjunto de jss-preset-default) estão incluídos:

- 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

Claro, você é livre para usar plugins adicionais. Aqui está um exemplo com o plugin jss-rtl.

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

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

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

String templates

Se você preferir a sintaxe CSS sobre o JSS, você pode usar o plugin jss-plugin-template .

```
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 que isto não suporta seletores, ou regras aninhadas.

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

Ordem de injeção do CSS

> *É **realmente importante** entender como a especificidade do CSS é calculada pelo navegador, como um dos elementos chave para saber quando sobrescrever estilos. Recomendamos que você leia este parágrafo do MDN: [Como a especificidade é calculada?]*

Por padrão, os estilos são inseridos **por último** no elemento `<head>` da sua página. Eles ganham mais especificidade do que qualquer outra folha de estilo em sua página, por exemplo, módulos CSS, componentes estilizados (styled components).

injectFirst

O componente `StylesProvider` tem uma propriedade `injectFirst` para injetar as tags de estilo em **primeiro** no cabeçalho (menor prioridade):

```
import { StylesProvider } from '@material-ui/styles';

<StylesProvider injectFirst>
  {/* Sua árvore de componentes.
      Componentes com estilo podem sobrescrever os estilos de Material-UI. */}
</StylesProvider>;
```

`makeStyles` / `withStyles` / `styled`

A injeção de tags de estilo acontece na **mesma ordem** das invocações de `makeStyles` / `withStyles` / `styled`. Por exemplo, a cor vermelha ganha maior especificidade neste caso:

```
import clsx from 'clsx';
import { makeStyles } from '@material-ui/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} />;
}
```

A ordem de chamada do hook e a ordem de concatenação da classe **não importam**.

Ponto de inserção (insertionPoint)

JSS fornece um mecanismo para controlar esta situação. By adding an `insertionPoint` within the HTML you can control the order that the CSS rules are applied to your components.

Comentário HTML

A abordagem mais simples é adicionar um comentário HTML no `<head>` que determina onde o JSS vai injetar os estilos:

```
<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>;
}
  insertionPoint: 'jss-insertion-point',
});

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

Outros elementos HTML

Create React App remove comentários em HTML ao criar a compilação de produção. Para contornar esse comportamento, você pode fornecer um elemento DOM (diferente de um comentário) como o ponto de inserção do JSS, por exemplo, um elemento `<noscript>`:

```
<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: 'jss-insertion-point',
});

export default function App() {
  return <StylesProvider jss={jss}>...</StylesProvider>;
}
  insertionPoint: document.getElementById('jss-insertion-point'),
});

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

JS createComment

codesandbox.io impede o acesso ao elemento `<head>`. Para contornar esse comportamento, você pode usar a API JavaScript `document.createComment()`:

```
insertionPoint: document.getElementById('jss-insertion-point'),
});

export default function App() {
  return <StylesProvider jss={jss}>...</StylesProvider>;
}
  import { create } from 'jss';
import { StylesProvider, jssPreset } from '@material-ui/styles';

const jss = create({
  ...jssPreset(),
  // Defina um ponto de inserção customizado que o JSS irá procurar para injetar os
estilos no DOM.
  insertionPoint: 'jss-insertion-point',
});
```

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

Renderização do lado servidor

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 '@material-ui/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>
  `;
}
```

Você pode seguir o guia de renderização no servidor para um exemplo mais detalhado, ou leia o `ServerStyleSheets` na documentação da API.

Gatsby

Existe um plugin oficial Gatsby que permite a renderização do lado do servidor para `@material-ui/styles`. Consulte a página do plugin para obter instruções de configuração e uso.

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

Next.js

Você precisa ter um `pages/_document.js` customizado, então copie esta lógica para injetar os estilos renderizados no lado do servidor no elemento `<head>`.

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

Nomes de classes

Os nomes de classes são gerados pelo gerador de nome de classe.

Padrão

Por padrão, os nomes de classes gerados por `@material-ui/core/styles` são **não determinísticos**; você não pode confiar que eles irão permanecer os mesmos. Vejamos o seguinte estilo como um exemplo:

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

Isto irá gerar um nome de classe como `makeStyles-root-123`.

Você tem que usar a propriedade `classes` de um componente para sobrescrever os estilos. O comportamento não determinístico dos nomes de classes permitem o isolamento de estilos.

- Em **desenvolvimento**, o nome da classe é: `.makeStyles-root-123` seguindo esta lógica:

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

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

- Em **produção**, o nome da classe é: `.jss123` seguindo esta lógica:

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

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

Quando as seguintes condições são atendidas, os nomes das classes são **determinísticos**:

- Apenas um provedor de tema é usado (**Sem aninhamento de tema **)
- A folha de estilo tem um nome que começa com `Mui` (todos os componentes do Material-UI).
- A opção `disableGlobal` do gerador de nome de classe é `false` (o padrão).

CSS global

`jss-plugin-global`

O plugin `jss-plugin-global` é instalado na predefinição padrão. Você pode usá-lo para definir nomes de classes globais.

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

Híbrido

Você também pode combinar nomes de classe gerados pelo JSS com nomes globais.

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

Prefixos CSS

O JSS usa recursos de detecção para aplicar os prefixos corretos. Não fique surpreso se você não conseguir ver um prefixo específico na versão mais recente do Chrome. Seu navegador provavelmente não precisa disso.

TypeScript usage

Utilizando `withStyles` no TypeScript pode ser um pouco complicado, mas há alguns utilitários que tornam a experiência menos dolorosa possível.

Utilizando `createStyles` para evitar a ampliação de tipo (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);
//         ^^^^^^
//         Os tipos de propriedade 'flexDirection' são incompatíveis.
//           Tipo 'string' não pode ser atribuído para o tipo '"-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. Para corrigir isto, você pode passar o objeto de estilos diretamente para `withStyles`:

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

No entanto, a ampliação de tipos continuará a causar dores de cabeça se você tentar fazer com que os estilos dependam do tema:

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

Isso ocorre pois o TypeScript amplia o retorno de tipos de expressões de função.

Por causa disso, é recomendado usar a função utilitária `createStyles` para construir seu objeto de regras de estilo:

```
// Estilos sem dependência
const styles = createStyles({
  root: {
    display: 'flex',
    flexDirection: 'column',
  },
});

// Estilos com dependência do tema
const styles = ({ palette, spacing }: Theme) =>
  createStyles({
    root: {
      display: 'flex',
      flexDirection: 'column',
      padding: spacing.unit,
      backgroundColor: palette.background.default,
      color: palette.primary.main,
    },
  });
```

`createStyles` é apenas a identidade da função; ela não "faz nada" em tempo de execução, apenas auxilia a inferência de tipos em tempo de compilação.

Consultas de Mídia (Media queries)

`withStyles` permite utilizar um objeto de estilos de nível superior com consultas de mídia assim:

```
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.

```
// erro porque TypeScript acha que `@media (min-width: 960px)` é o nome da classe
// e `content` é a propriedade css
const ambiguousStyles = createStyles({
```

```
  content: {
    minHeight: '100vh',
  },
  '@media (min-width: 960px)': {
    content: {
      display: 'flex',
    },
  },
});

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

Incrementando suas propriedades utilizando `WithStyles`

Desde que um componente seja decorado com `withStyles(styles)`, ele recebe uma propriedade injetada `classes`, você pode querer definir estas propriedades de acordo com:

```
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;
  };
}
```

No entanto isto não é muito elegante de acordo com o princípio de software DRY, porque requer que você mantenha os nomes das classes (`'root'` , `'paper'` , `'button'` , ...) em dois locais diferentes. Nós fornecemos um operador de tipo `WithStyles` para ajudar com isso, assim você pode apenas escrever:

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

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

interface Props extends WithStyles<typeof styles> {
  foo: number;
  bar: boolean;
}
```

Decorando componentes

Aplicando `withStyles(styles)` como uma função, nos dá o resultado como o esperado:

```
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>
      );
    }
  },
);
```

Infelizmente devido a uma limitação atual dos decoradores do TypeScript, `withStyles(styles)` não pode ser usado como decorador no TypeScript.