Unstyled components

Implementing a custom design system using MUI.

MUI Base provides a set of components without any styles. These can be used to implement a custom design system that is not based on Material Design.

So far, just a few components have been created, but we intend to focus on the `@mui/base` package extensively in the upcoming months.

Use cases

Is the Base package good for you?

If you:

- need to implement a design system that significantly differs from Material Design,
- need precise control over the rendered HTML of the components,
- or need to implement custom components but don't want to deal with accessibility features on your own,

then you will find `@mui/base` useful.

If, however, you:

- need components that look good out of the box,
- need to customize Material Design with your brand colors,
- or need to implement a design system based on Material Design,

then you may be better off using the `@mui/material` package and customizing it.

Components vs. hooks

The Base package has two kinds of building blocks:

- unstyled components
- hooks

Most of the unstyled components are implemented with the help of a hook (where it makes sense). Hooks encapsulate logic, while components provide structure.

When creating custom components based on the unstyled ones, you can use either unstyled components or hooks.

With components, you will usually be able to write less code, but with hooks you have the ultimate control over the structure of the rendered HTML. However, to make the resulting component accessible, you need to create the components the hook expects and apply props returned by the hook. Concrete examples are provided in each hook's documentation.

Anatomy of an unstyled component

While each component has its own API, there are a few props common to all of them:

- `components` - an object that allows you to override subcomponents (slots) used by the unstyled "host" component. Each host component will at least have the `Root` slot. Many complex components have more slots. You can either provide a custom component or an HTML tag there.

  ```
  <BadgeUnstyled components={{ Root: 'div', Badge: MyCustomBadge }} />
  ```

- `component` - a shortcut to `components.Root`. Note that if both `components.Root` and `component` are specified, `component` takes precedence.

  ```
  <BadgeUnstyled component="div" />
  ```

- `componentsProps` - an object with each slot's props. Even if you don't override a specific slot (with `components` or `component`), you can provide additional props for the default components. For example, if you want to add a custom CSS class to one of the slots' components, specify it in the `componentsProps`.

  ```
  <BadgeUnstyled componentsProps={{ input: { className: 'my-badge' } }} />
  ```

Additionally, all the extra props placed on the host component are propagated into the root component (just as if they were placed in `componentsProps.root`). These two examples are equivalent:

```
<BadgeUnstyled id="badge1">
```

```
<BadgeUnstyled componentsProps={{ root: { id: 'badge1' } }}>
```

Styling the unstyled

There are several levels of customization available:

Applying custom CSS rules to unstyled components

If you're happy with the structure of the HTML rendered by the unstyled component by default, you can apply custom styles to the classes used by the component. Each component has its own set of classes. Some are static - present on different parts of the component all the time, and some are applied conditionally (like `Mui-disabled`, applied when a component is disabled).

Each component's API documentation lists all classes that the component uses. Additionally, you can import a `[componentName]Classes` object that describes all the classes a given component uses (see the demo below for an example).

{{"demo": "StylingCustomCss.js"}}

Overriding the unstyled components' slots

If you don't want to rely on default components, you can override them with your own. This makes it possible to provide a styled component. Each unstyled component has a specific set of "slots" - that is subcomponents that you can override.

Let's take a SwitchUnstyled as an example. It has three slots: `Root`, `Thumb`, and `Input`. The demo below shows how to create a styled component (using System in this case, but it could well be any other solution)

{{"demo": "StylingSlots.js"}}

The components you pass in the `components` prop receive the `ownerState` prop from the top-level component (host). By convention, it contains all props passed to the host, merged with its rendering "state".

For example:

```
<SwitchUnstyled components={{ Thumb: MyCustomThumb }} data-foo="42" />
```

In this case, `MyCustomThumb` component will receive the `ownerState` object with the following data:

```
{
  checked: boolean;
  disabled: boolean;
  focusVisible: boolean;
  readOnly: boolean;
  'data-foo': string;
}
```

You can use this object to style your component.

Creating custom components using hooks

If you need to create your own component structure, you can use the provided hooks. They encapsulate the logic of the components while not enforcing structure.

Hooks return the current state of the component (`checked`, `disabled`, `open`, etc., depending on the component) and provide functions that return props you can apply to your components.

Again, let's take a Switch as an example. The corresponding hook is called `useSwitch`. It returns the following object:

```
{
  checked: Readonly<boolean>;
  disabled: Readonly<boolean>;
  readOnly: Readonly<boolean>;
  focusVisible: Readonly<boolean>;
```

```
  getInputProps: (otherProps?: object) => SwitchInputProps;
}
```

The `checked`, `disabled`, `readOnly`, and `focusVisible` fields represent the state of the switch. Use them to apply styling to your HTML elements. The `getInputProps` function can be used to get the props to place on an HTML `input` to make the switch accessible.

{{“demo”: “StylingHooks.js”}}