API Design Approach

We have learned a great deal regarding how MUI is used, and the v1 rewrite allowed us to completely rethink the component API.

> *API design is hard because you can make it seem simple but it's actually deceptively complex, or make it actually simple but seem complex.*
>
> [@sebmarkbage]

As Sebastian Markbage pointed out, no abstraction is superior to wrong abstractions. We are providing low-level components to maximize composition capabilities.

Composition

You may have noticed some inconsistency in the API regarding composing components. To provide some transparency, we have been using the following rules when designing the API:

1. Using the `children` prop is the idiomatic way to do composition with React.
2. Sometimes we only need limited child composition, for instance when we don't need to allow child order permutations. In this case, providing explicit props makes the implementation simpler and more performant; for example, the `Tab` takes an `icon` and a `label` prop.
3. API consistency matters.

Rules

Aside from the above composition trade-off, we enforce the following rules:

Spread

Props supplied to a component which are not explicitly documented are spread to the root element; for instance, the `className` prop is applied to the root.

Now, let's say you want to disable the ripples on the `MenuItem`. You can take advantage of the spread behavior:

```
<MenuItem disableRipple />
```

The `disableRipple` prop will flow this way: `MenuItem` > `ListItem` > `ButtonBase`.

Native properties

We avoid documenting native properties supported by the DOM like `className`.

CSS Classes

All components accept a `classes` prop to customize the styles. The classes design answers two constraints: to make the classes structure as simple as possible, while sufficient to implement the Material Design guidelines.

- The class applied to the root element is always called `root`.
- All the default styles are grouped in a single class.
- The classes applied to non-root elements are prefixed with the name of the element, e.g. `paperWidthXs` in the Dialog component.

- The variants applied by a boolean prop **aren't** prefixed, e.g. the `rounded` class applied by the `rounded` prop.
- The variants applied by an enum prop **are** prefixed, e.g. the `colorPrimary` class applied by the `color="primary"` prop.
- A variant has **one level of specificity**. The `color` and `variant` props are considered a variant. The lower the style specificity is, the simpler it is to override.
- We increase the specificity for a variant modifier. We already **have to do it** for the pseudo-classes (`:hover` , `:focus` , etc.). It allows much more control at the cost of more boilerplate. Hopefully, it's also more intuitive.

```
const styles = {
  root: {
    color: green[600],
    '&$checked': {
      color: green[500],
    },
  },
  checked: {},
};
```

Nested components

Nested components inside a component have:

- their own flattened props when these are key to the top level component abstraction, for instance an `id` prop for the `Input` component.
- their own `xxxProps` prop when users might need to tweak the internal render method's sub-components, for instance, exposing the `inputProps` and `InputProps` props on components that use `Input` internally.
- their own `xxxComponent` prop for performing component injection.
- their own `xxxRef` prop when you might need to perform imperative actions, for instance, exposing an `inputRef` prop to access the native `input` on the `Input` component. This helps answer the question "How can I access the DOM element?"

Prop naming

The name of a boolean prop should be chosen based on the **default value**. This choice allows:

- the shorthand notation. For example, the `disabled` attribute on an input element, if supplied, defaults to `true` :

  ```
  <Input enabled={false} /> ❌
  <Input disabled /> ✅
  ```

- developers to know what the default value is from the name of the boolean prop. It's always the opposite.

Controlled components

Most of the controlled component are controlled via the `value` and the `onChange` props, however, the `open` / `onClose` / `onOpen` combination is used for display related state. In the cases where there are more events, we

put the noun first, and then the verb, for example: `onPageChange`, `onRowsChange`.

boolean vs. enum

There are two options to design the API for the variations of a component: with a *boolean*; or with an *enum*. For example, let's take a button that has different types. Each option has its pros and cons:

- Option 1 *boolean*:

  ```tsx
  type Props = {
    contained: boolean;
    fab: boolean;
  };
  ```

  This API enables the shorthand notation: `<Button>`, `<Button contained />`, `<Button fab />`.

- Option 2 *enum*:

  ```tsx
  type Props = {
    variant: 'text' | 'contained' | 'fab';
  };
  ```

  This API is more verbose: `<Button>`, `<Button variant="contained">`, `<Button variant="fab">`.

  However, it prevents an invalid combination from being used, bounds the number of props exposed, and can easily support new values in the future.

The MUI components use a combination of the two approaches according to the following rules:

- A *boolean* is used when **2** possible values are required.
- An *enum* is used when **> 2** possible values are required, or if there is the possibility that additional possible values may be required in the future.

Going back to the previous button example; since it requires 3 possible values, we use an *enum*.

Ref

The `ref` is forwarded to the root element. This means that, without changing the rendered root element via the `component` prop, it is forwarded to the outermost DOM element which the component renders. If you pass a different component via the `component` prop, the ref will be attached to that component instead.

Glossary

- **host component**: a DOM node type in the context of `react-dom`, e.g. a `'div'`. See also React Implementation Notes.
- **host element**: a DOM node in the context of `react-dom`, e.g. an instance of `window.HTMLDivElement`.
- **outermost**: The first component when reading the component tree from top to bottom i.e. breadth-first search.
- **root component**: the outermost component that renders a host component.
- **root element**: the outermost element that renders a host component.