Composition

MUI tries to make composition as easy as possible.

Wrapping components

To provide maximum flexibility and performance, MUI needs a way to know the nature of the child elements a component receives. To solve this problem, we tag some of the components with a `muiName` static property when needed.

You may, however, need to wrap a component in order to enhance it, which can conflict with the `muiName` solution. If you wrap a component, verify if that component has this static property set.

If you encounter this issue, you need to use the same tag for your wrapping component that is used with the wrapped component. In addition, you should forward the props, as the parent component may need to control the wrapped components props.

Let's see an example:

```
const WrappedIcon = (props) => <Icon {...props} />;
WrappedIcon.muiName = Icon.muiName;
```

{{"demo": "Composition.js"}}

Component prop

MUI allows you to change the root element that will be rendered via a prop called `component`.

How does it work?

The custom component will be rendered by MUI like this:

```
return React.createElement(props.component, props);
```

For example, by default a `List` component will render a `<ul>` element. This can be changed by passing a React component to the `component` prop. The following example will render the `List` component with a `<nav>` element as root element instead:

```
<List component="nav">
  <ListItem button>
    <ListItemText primary="Trash" />
  </ListItem>
  <ListItem button>
    <ListItemText primary="Spam" />
  </ListItem>
</List>
```

This pattern is very powerful and allows for great flexibility, as well as a way to interoperate with other libraries, such as your favorite routing or forms library. But it also **comes with a small caveat!**

Caveat with inlining

Using an inline function as an argument for the `component` prop may result in **unexpected unmounting**, since a new component is passed every time React renders. For instance, if you want to create a custom `ListItem` that acts as a link, you could do the following:

```
import { Link } from 'react-router-dom';

function ListItemLink(props) {
  const { icon, primary, to } = props;

  const CustomLink = (props) => <Link to={to} {...props} />;

  return (
    <li>
      <ListItem button component={CustomLink}>
        <ListItemIcon>{icon}</ListItemIcon>
        <ListItemText primary={primary} />
      </ListItem>
    </li>
  );
}
```

⚠️ However, since we are using an inline function to change the rendered component, React will unmount the link every time `ListItemLink` is rendered. Not only will React update the DOM unnecessarily, the ripple effect of the `ListItem` will also not work correctly.

The solution is simple: **avoid inline functions and pass a static component to the `component` prop** instead. Let's change the `ListItemLink` component so `CustomLink` always reference the same component:

```
import { Link, LinkProps } from 'react-router-dom';

function ListItemLink(props) {
  const { icon, primary, to } = props;

  const CustomLink = React.useMemo(
    () =>
      React.forwardRef<HTMLAnchorElement, Omit<RouterLinkProps, 'to'>>(function
Link(
        linkProps,
        ref,
      ) {
        return <Link ref={ref} to={to} {...linkProps} />;
      }),
    [to],
  );

  return (
    <li>
      <ListItem button component={CustomLink}>
        <ListItemIcon>{icon}</ListItemIcon>
```

```
          <ListItemText primary={primary} />
        </ListItem>
      </li>
    );
  }
```

Caveat with prop forwarding

You can take advantage of the prop forwarding to simplify the code. In this example, we don't create any intermediary component:

```
import { Link } from 'react-router-dom';

<ListItem button component={Link} to="/">
```

⚠️ However, this strategy suffers from a limitation: prop collisions. The component providing the `component` prop (e.g. ListItem) might not forward all the props (for example dense) to the root element.

With TypeScript

Many MUI components allow you to replace their root node via a `component` prop, this is detailed in the component's API documentation. For example, a Button's root node can be replaced with a React Router's Link, and any additional props that are passed to Button, such as `to`, will be spread to the Link component. For a code example concerning Button and react-router-dom checkout these demos.

To be able to use props of such a MUI component on their own, props should be used with type arguments. Otherwise, the `component` prop will not be present in the props of the MUI component.

The examples below use `TypographyProps` but the same will work for any component which has props defined with `OverrideProps`.

The following `CustomComponent` component has the same props as the `Typography` component.

```
function CustomComponent(props: TypographyProps<'a', { component: 'a' }>) {
  /* ... */
}
```

Now the `CustomComponent` can be used with a `component` prop which should be set to `'a'`. In addition, the `CustomComponent` will have all props of a `<a>` HTML element. The other props of the `Typography` component will also be present in props of the `CustomComponent`.

It is possible to have generic `CustomComponent` which will accept any React component, custom, and HTML elements.

```
function GenericCustomComponent<C extends React.ElementType>(
  props: TypographyProps<C, { component?: C }>,
) {
  /* ... */
}
```

If the `GenericCustomComponent` will be used with a `component` prop provided, it should also have all props required by the provided component.

```
function ThirdPartyComponent({ prop1 }: { prop1: string }) {
  return <div />;
}
// ...
function ThirdPartyComponent({ prop1 }: { prop1: string }) {
  return <div />;
}
// ...
```

The `prop1` became required for the `GenericCustomComponent` as the `ThirdPartyComponent` has it as a requirement.

Not every component fully supports any component type you pass in. If you encounter a component that rejects its `component` props in TypeScript, please open an issue. There is an ongoing effort to fix this by making component props generic.

Caveat with refs

This section covers caveats when using a custom component as `children` or for the `component` prop.

Some of the components need access to the DOM node. This was previously possible by using `ReactDOM.findDOMNode` . This function is deprecated in favor of `ref` and ref forwarding. However, only the following component types can be given a `ref` :

- Any MUI component
- class components i.e. `React.Component` or `React.PureComponent`
- DOM (or host) components e.g. `div` or `button`
- React.forwardRef components
- React.lazy components
- React.memo components

If you don't use one of the above types when using your components in conjunction with MUI, you might see a warning from React in your console similar to:

> *Function components cannot be given refs. Attempts to access this ref will fail. Did you mean to use React.forwardRef()?*

Note that you will still get this warning for `lazy` and `memo` components if their wrapped component can't hold a ref. In some instances, an additional warning is issued to help with debugging, similar to:

> *Invalid prop `component` supplied to `ComponentName` . Expected an element type that can hold a ref.*

Only the two most common use cases are covered. For more information see this section in the official React docs.

```diff
-const MyButton = () => <div role="button" />;
+const MyButton = React.forwardRef((props, ref) =>
+  <div role="button" {...props} ref={ref} />);

 <Button component={MyButton} />;
```

```diff
-const SomeContent = props => <div {...props}>Hello, World!</div>;
+const SomeContent = React.forwardRef((props, ref) =>
+  <div {...props} ref={ref}>Hello, World!</div>);

 <Tooltip title="Hello again."><SomeContent /></Tooltip>;
```

To find out if the MUI component you're using has this requirement, check out the props API documentation for that component. If you need to forward refs the description will link to this section.

Caveat with StrictMode

If you use class components for the cases described above you will still see warnings in `React.StrictMode`. `ReactDOM.findDOMNode` is used internally for backwards compatibility. You can use `React.forwardRef` and a designated prop in your class component to forward the `ref` to a DOM component. Doing so should not trigger any more warnings related to the deprecation of `ReactDOM.findDOMNode`.

```diff
 class Component extends React.Component {
   render() {
-    const { props } = this;
+    const { forwardedRef, ...props } = this.props;
     return <div {...props} ref={forwardedRef} />;
   }
 }

-export default Component;
+export default React.forwardRef((props, ref) => <Component {...props} forwardedRef=
{ref} />);
```