Abordagem do Design da API

Nós aprendemos bastante como o Material-UI é usado e a refatoração da v1 permitiu-nos repensar completamente a API dos componentes.

> *O design da API é difícil porque você pode fazer com que pareça simples, mas na verdade é extremamente complexo ou simples, mas parece complexo.*

@sebmarkbage

Como Sebastian Markbage apontou, nenhuma abstração é superior a abstrações erradas. Estamos fornecendo componentes de baixo nível para maximizar os recursos de composição.

Composição

Você deve ter notado alguma inconsistência na API em relação à composição de componentes. Para fornecer alguma transparência, usamos as seguintes regras ao projetar a API:

1. Usando a propriedade `children` é a maneira idiomática de fazer composição com React.
2. Às vezes, precisamos apenas de uma composição limitada ao elemento filho, por exemplo, quando não precisamos permitir ordem de permutações com um elemento filho. Nesse caso, fornecer propriedades explícitas torna a implementação mais simples e com maior desempenho; por exemplo, o componente `Tab` recebe uma propriedade `icon` e `label`.
3. A consistência da API é importante.

Regras

Além do trade-off da composição acima, aplicamos as seguintes regras:

Propagação

Propriedades fornecidas para um componente que não estão explicitamente documentadas são propagadas para o elemento raiz; por exemplo, a propriedade `className` é aplicada no elemento raiz.

Agora, digamos que você queira desabilitar o efeito cascata do `MenuItem`. Você pode aproveitar o comportamento da propagação:

```
<MenuItem disableRipple />
```

A propriedade `disableRipple` propagará desta maneira: `MenuItem` > `ListItem` > `ButtonBase`.

Propriedades nativas

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

Classes CSS

Todos os componentes aceitam uma propriedade `classes` para customizar os estilos. The classes design answers two constraints: to make the classes structure as simple as possible, while sufficient to implement the Material Design guidelines.

- A classe aplicada ao elemento raiz é sempre chamada de `root`.
- Todos os estilos padrão são agrupados em uma única classe.

- As classes aplicadas a elementos não-raiz são prefixadas com o nome do elemento, por exemplo, `paperWidthXs` no componente Dialog.
- As variantes aplicadas por uma propriedade booleana **não são** prefixadas, por exemplo, a classe `rounded` aplicada pela propriedade `rounded`.
- As variantes aplicadas por uma propriedade enum **são** prefixadas, por exemplo, a classe `colorPrimary` aplicada pela propriedade `color="primary"`.
- Uma variante tem **um nível de especificidade**. As propriedades `color` e `variant` são consideradas uma variante. Quanto menor a especificidade de estilo, mais simples é sobrescrever.
- Aumentamos a especificidade de um modificador variante. Nós já **temos que fazer isso** para as pseudo-classes (`:hover`, `:focus`, etc.). Permite muito mais controle ao custo de mais trabalho. Esperamos que também seja mais intuitivo.

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

Componentes aninhados

Os componentes aninhados dentro de um componente possuem:

- suas próprias propriedades niveladas quando estas são chaves para a abstração do componente de nível superior, por exemplo, uma propriedade `id` para o componente `input`.
- suas próprias propriedades `xxxProps`, quando os usuários podem precisar ajustar os subcomponentes do método de renderização interno, por exemplo, expondo as propriedades `inputProps` e `InputProps` em componentes que usam `Input` internamente.
- suas próprias propriedades `xxxComponent` para executar a injeção de componentes.
- suas próprias propriedades `xxxRef`, quando o usuário precisar executar ações imperativas, por exemplo, expondo uma propriedade `inputRef` para acessar nativamente o `input` no componente `Input`. Isso ajuda a responder a pergunta "Como posso acessar o elemento DOM?"

Prop naming

O nome de uma propriedade booleana deve ser escolhido com base no **valor padrão**. Essa escolha permite a notação abreviada:

- the shorthand notation. Por exemplo, o atributo `disabled` em um elemento de entrada, se fornecido, é padronizado para `true`:

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

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

Componentes controlados

A maior parte de componentes controlados, é controlado pelas propriedades `value` e `onChange`, no entanto, o `open` / `onClose` / `onOpen` é uma combinação usada para o estado relacionado à exibição. Nos casos em que há mais eventos, colocamos o substantivo em primeiro lugar e depois o verbo, por exemplo: `onPageChange`, `onRowsChange`.

boolean vs. enum

Existem duas opções para projetar a API para as variações de um componente: com um *booleano*; ou com um *enumerador*. Por exemplo, vamos pegar um botão que tenha tipos diferentes. Cada opção tem seus prós e contras:

- Opção 1 *booleano*:

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

  Esta API ativou a notação abreviada: `<Button>`, `<Button contained />`, `<Button fab />`.

- Opção 2 *enumerador*:

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

  Esta API é mais verbosa: `<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.

Os componentes do Material-UI usam uma combinação das duas abordagens de acordo com as seguintes regras:

- Um *booleano* é usado quando **2** valores possíveis são necessários.
- **elemento hospedeiro**: um nó DOM no contexto de `react-dom`, por exemplo, uma instância de `window.HTMLDivElement`.

Voltando ao exemplo do botão anterior; ele requer 3 valores possíveis, usamos um *enumerador*.

Ref

O `ref` é encaminhado para o elemento raiz. Isso significa que, sem alterar o elemento raiz renderizado através da propriedade `component`, ele é encaminhado para o elemento DOM mais externo para que o componente renderize. Se você passar um componente diferente através da propriedade `component`, o ref será anexado para esse componente.

Glossário

- **componente hospedeiro**: um tipo de nó DOM no contexto de `react-dom`, por exemplo, um `'div'`. Veja também as Notas de implementação do React.
- **elemento hospedeiro**: um nó DOM no contexto de `react-dom`, por exemplo, uma instância de `window.HTMLDivElement`.
- **mais externo**: O primeiro componente ao ler a árvore de componentes de cima para baixo, ou seja, busca em largura (breadth-first search).
- **componente raiz**: o componente mais externo que renderiza um componente do hospedeiro.
- **elemento raiz**: o elemento mais externo que renderiza um componente hospedeiro.