O que é o `@Output`?

O decorator `@Output` é usado em Angular para permitir que um componente filho envie eventos ou dados para seu componente pai. Ele funciona em conjunto com a classe `EventEmitter` para emitir eventos personalizados que podem ser manipulados pelo componente pai.

Como funciona?

1. **Definição do evento no componente filho:**
    - Usa-se o decorator `@Output` para criar um emissor de eventos (`EventEmitter`).
    - Esse evento pode carregar um dado ou simplesmente notificar o componente pai.
2. **Emissão do evento:**
    - O evento é disparado usando o método `.emit(value)` do `EventEmitter`.
3. **Escuta do evento no componente pai:**
    - O componente pai escuta o evento através de uma binding no HTML, usando a sintaxe `(eventName)="parentMethod($event)"`.

Exemplo Básico

1. Componente Filho

```
import { Component, EventEmitter, Output } from '@angular/core';

@Component({
  selector: 'app-child',
  template: `
    <button (click)="sendMessage()">Enviar Mensagem</button>
  `,
})
```

```
export class ChildComponent {
  // Define o evento personalizado com o decorator @Output
  @Output() messageEvent = new EventEmitter<string>();

  // Método que emite o evento com um valor
  sendMessage() {
    this.messageEvent.emit('Olá, Pai!');
  }
}
```

2. Componente Pai

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-parent',
  template: `
    <app-child (messageEvent)="receiveMessage($event)"></app-child>
    <p>Mensagem do Filho: {{ message }}</p>
  `,
})
export class ParentComponent {
  message: string = '';

  // Método que será chamado quando o evento for emitido
  receiveMessage(event: string) {
    this.message = event;
  }
}
```

O que acontece?

- Quando o botão no componente `ChildComponent` é clicado, o método `sendMessage()` é chamado.
- Esse método emite o evento `messageEvent` com o valor `'Olá, Pai!'`.
- O componente pai (`ParentComponent`) escuta esse evento e executa o método `receiveMessage()`, atualizando a propriedade `message`.

Outros Exemplos e Usos

1. Emitindo Objetos Complexos

Você não precisa emitir apenas strings. Qualquer tipo de dado (números, objetos, arrays, etc.) pode ser enviado.

Componente Filho

```
@Output() dataEvent = new EventEmitter<{ id: number; name: string }>();

sendData() {
  this.dataEvent.emit({ id: 1, name: 'Angular' });
}
```

Componente Pai

```
receiveData(event: { id: number; name: string }) {
  console.log('Dados recebidos:', event);
}
```

---

2. Trabalhando com Vários Eventos

Você pode definir múltiplos eventos no mesmo componente filho.

Componente Filho

```
@Output() incrementEvent = new EventEmitter<number>();
@Output() decrementEvent = new EventEmitter<number>();

increment() {
  this.incrementEvent.emit(1);
}

decrement() {
  this.decrementEvent.emit(1);
}
```

Componente Pai

```
<app-child
```

```
  (incrementEvent)="incrementCounter($event)"
  (decrementEvent)="decrementCounter($event)">
</app-child>
```

Métodos no Pai

```
incrementCounter(value: number) {
  this.counter += value;
}

decrementCounter(value: number) {
  this.counter -= value;
}
```

---

3. Emitindo Eventos sem Dados

Se você só quer notificar o componente pai, pode emitir eventos sem carregar valores.

Componente Filho

```
@Output() notifyEvent = new EventEmitter<void>();

notify() {
  this.notifyEvent.emit();
}
```

Componente Pai

```
<app-child (notifyEvent)="handleNotification()"></app-child>
```

Método no Pai

```
handleNotification() {
  console.log('Notificado pelo filho!');
}
```

Boas Práticas com `@Output`

1. **Nomeclatura clara:**
    - Use nomes descritivos para eventos, como `buttonClicked`, `dataUpdated`, ou `userLoggedOut`.
2. **Evite lógica complexa no filho:**
    - O componente filho deve emitir eventos e deixar a maior parte da lógica para o componente pai.
3. **Evite poluição do pai:**
    - Se o componente pai precisa tratar muitos eventos do filho, reavalie a separação de responsabilidades.

Conclusão

O decorator `@Output` e o `EventEmitter` são ferramentas essenciais em Angular para comunicação de baixo para cima (filho -> pai). Eles permitem que os componentes interajam de forma eficiente e ajudam a manter o código limpo e modular.