

Angular 17: Concetti Avanzati



Composizione di componenti, gestione dati, eventi e lifecycle

Composizione di Componenti (1/2)

Cosa fa?

- Tecnica per costruire UI complesse combinando componenti più piccoli
- Esempio: un componente `ProductList` che contiene molti `ProductCard`

Perché si usa?

- ✓ Migliora la modularità
- ✓ Favorisce il riutilizzo del codice
- ✓ Semplifica la manutenzione

Composizione di Componenti (2/2)

Come si usa?

```
@Component({  
  selector: 'app-parent',  
  template: `  
    <app-header></app-header>  
    <app-product-list></app-product-list>  
    <app-footer></app-footer>  
  `,  
})
```

Quando si usa?

- In tutte le applicazioni Angular non banali
- Quando si vogliono creare componenti riutilizzabili

Passare Dati a Componenti (1/2)

Cosa fa?

- Meccanismo per comunicare dati da padre a figlio via `@Input`

Perché è importante?

- Crea componenti dinamici e riutilizzabili
- Separa chiaramente logica e presentazione

Passare Dati a Componenti (2/2)

Implementazione

```
// Padre
@Component({
  template: `<app-child [items]="products"></app-child>`
})

// Figlio
@Input() items: Product[];
```

Scenario tipico

- Lista di prodotti che riceve dati da un servizio
- Componenti UI generici (es. card, tabelle)

Interazione Utente (1/2)

Cosa gestisce?

- Click, hover, input, form submission
- Tutti gli eventi del DOM

Esempi comuni

- Pulsanti di azione
- Campi di input form
- Gestione drag-and-drop

Interazione Utente (2/2)

Implementazione

```
template: `  
  <button (click)="addToCart()">Aggiungi</button>  
  <input (keyup.enter)="search()">  
  `
```



```
// Metodo nel componente  
addToCart() {  
  this.cartService.add(product);  
}
```

Generazione Eventi (1/2)

Cosa fa?

- Comunica dal componente figlio al padre via `@Output`

Pattern fondamentale

- Flusso unidirezionale dei dati
- Architettura a componenti scalabile

Generazione Eventi (2/2)

Implementazione

```
// Figlio
@Output() selected = new EventEmitter<Product>();

selectItem(product: Product) {
  this.selected.emit(product);
}

// Padre
<app-product (selected)="onSelect($event)"></app-product>
```

Lifecycle Hooks (1/3)

Fasi principali

1. Creazione (`ngOnInit`)
2. Aggiornamenti (`ngOnChanges`)
3. Distruzione (`ngOnDestroy`)

Perché sono utili?

- Controllo preciso sull'inizializzazione
- Gestione efficiente delle risorse

Lifecycle Hooks (2/3)

Hook più usati

```
ngOnInit() {  
  // Caricamento dati iniziali  
  this.loadData();  
}  
  
ngOnChanges(changes: SimpleChanges) {  
  // Reagisce ai cambi degli input  
}  
  
ngOnDestroy() {  
  // Pulizia subscription  
  this.sub.unsubscribe();  
}
```

Lifecycle Hooks (3/3)

Quando usarli?

| Hook | Scenario Tipico |
|-------------|------------------------------|
| ngOnInit | Fetch dati, inizializzazioni |
| ngOnChanges | Reazione a input dinamici |
| ngOnDestroy | Pulizia memory leaks |

Esempio Integrato





Componente Product Card

```
@Component({
  selector: 'app-product-card',
  template: `
    <div (click)="select()">
      <h3>{{ product.name }}</h3>
      <button (click)="addToCart($event)">+</button>
    </div>
  `
})
export class ProductCard {
  @Input() product: Product;
  @Output() selected = new EventEmitter();

  ngOnInit() { /* ... */ }
}
```

Best Practices

Regole d'oro

1.  Componenti piccoli e focalizzati
2.  Input/Output per comunicazione padre-figlio
3.  Usare ngOnDestroy per evitare memory leaks
4.  Nomi espliciti per eventi (es. `productSelected` non `selected`)

Domande?

Grazie per l'attenzione!

 Q&A Icon

Esempi pratici o chiarimenti?