

Luca Minotti

# ANGULAR SIGNALS

# AGENDA

- 1**    **Introduzione** ai Signals
- 2**    **Computed** signals
- 3**    **Effects**
- 4**    **Linked** signals
- 5**    **Component APIs** signal-based
- 6**    **Change detection** e vantaggi



# MATERIALE CORSO



Il codice e le slide sono disponibili in questo repository:

> <https://github.com/lminotti-reply/angular-signals-course>



# SIGNALS

Un segnale è un **contenitore** di un valore mutabile

```
quantity : WritableSignal<number> = signal( initialValue: 0)
```

Il valore può assumere **qualsiasi tipo**  
**number**, string, array, boolean, object, function

Richiesto Angular v17 (v16 in developer preview)



# SIGNALS

Un segnale è un **contenitore** di un valore mutabile

```
names : WritableSignal<string[]> = signal(['Luca', 'Matteo'])
```

Il valore può assumere **qualsiasi tipo**  
number, string, **array**, boolean, object, function

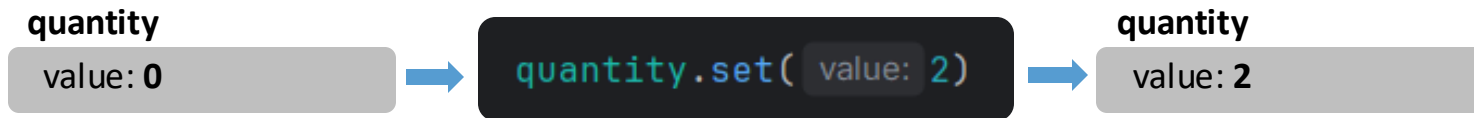
Richiesto Angular v17 (v16 in developer preview)



# SIGNALS

```
quantity : WritableSignal<number> = signal( initialValue: 0)
```

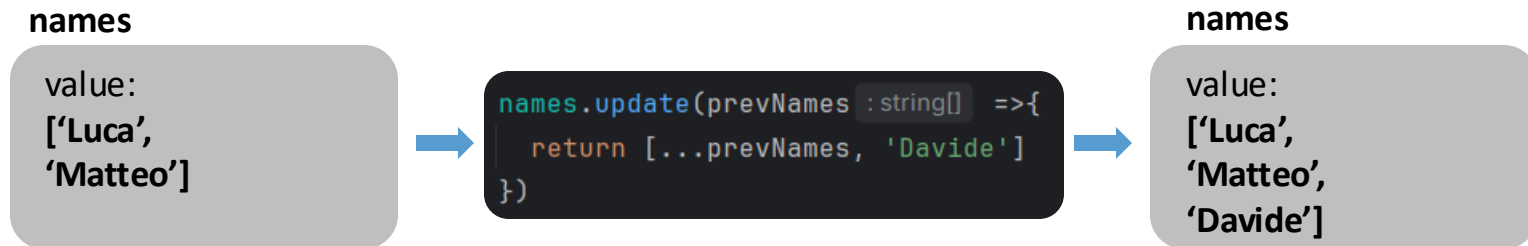
Il valore può essere **aggiornato** tramite apposite funzioni



# SIGNALS

```
names : WritableSignal<string[]> = signal(['Luca', 'Matteo'])
```

Il valore può essere **aggiornato** tramite apposite funzioni



# SIGNALS

Il valore può essere **letto** «invocando» il segnale

```
quantity.set( value: 2)  
quantity() // == 2
```



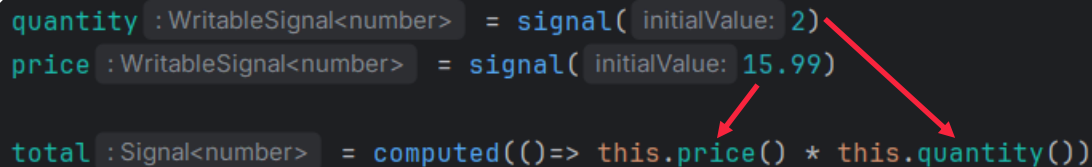


# COMPUTED SIGNALS

Possiamo **derivare** un segnale a partire da altri segnali

```
quantity : WritableSignal<number> = signal( initialValue: 2)
price : WritableSignal<number> = signal( initialValue: 15.99)

total : Signal<number> = computed(()=> this.price() * this.quantity())
```

A diagram showing the derivation of a computed signal. Three lines of code are shown in a dark box. The first line defines 'quantity' as a signal with an initial value of 2. The second line defines 'price' as a signal with an initial value of 15.99. The third line defines 'total' as a computed signal that takes the values of 'price' and 'quantity' and multiplies them. Red arrows point from the 'initialValue: 2' and 'initialValue: 15.99' in the first two lines to the 'this.price()' and 'this.quantity()' in the third line, respectively. A third red arrow points from the 'total' variable to the text below.

In questo momento varrebbe: 17,99

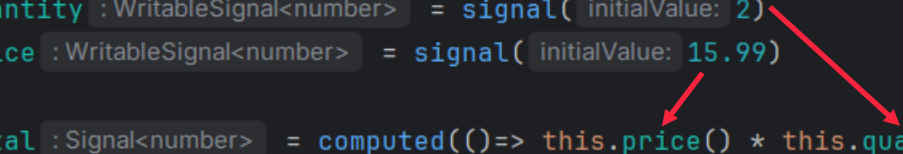


# COMPUTED SIGNALS

Possiamo **derivare** un segnale a partire da altri segnali

```
quantity : WritableSignal<number> = signal( initialValue: 2)
price : WritableSignal<number> = signal( initialValue: 15.99)

total : Signal<number> = computed(()=> this.price() * this.quantity())
```



In questo momento varrebbe: 31.98

```
function changeQuantity() {
  quantity.set( value: 1)
  total() // == 15.99
}
```

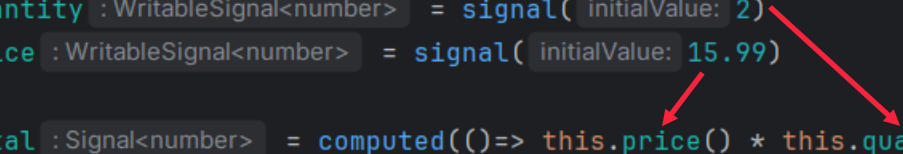
*total* si **aggiorna automaticamente** sulla base delle dipendenze



# COMPUTED SIGNALS

```
quantity : WritableSignal<number> = signal( initialValue: 2)
price : WritableSignal<number> = signal( initialValue: 15.99)

total : Signal<number> = computed(()=> this.price() * this.quantity())
```



In questo momento varrebbe: 31.98

```
function changeQuantity() {
  quantity.set( value: 1)
  total() // == 15.99
}
```

I computed signals sono **lazy** e **memoized**: il valore è calcolato solo quando qualcuno lo legge, ed è cached



# COMPUTED SIGNALS

```
1  const quantity : WritableSignal<number> = signal( initialValue: 2)
2  const price : WritableSignal<number> = signal( initialValue: 15.99)
3
4  const total : Signal<number> = computed(()=> {
5    console.log( message: "Ricalcolo")
6    return price() * quantity()
7  })
8
9  console.log(total()) // Il valore è calcolato solo in questo momento
10 console.log(total())
11 console.log( message: "Ora aggiorno la quantità")
12 quantity.set( value: 1)
13 console.log( message: "Quantità aggiornata")
14 setTimeout(() => {
15   console.log(total()) // Il valore è ri-calcolato solo in questo momento
16 }, delay: 5000)
```



# COMPUTED SIGNALS

```
1  const quantity : WritableSignal<number> = signal( initialValue: 2)
2  const price : WritableSignal<number> = signal( initialValue: 15.99)
3
4  const total : Signal<number> = computed(()=> {
5    console.log( message: "Ricalcolo")
6    return price() * quantity()
7  })
8
9  console.log(total()) // Il valore è calcolato solo in questo momento
10 console.log(total())
11 console.log( message: "Ora aggiorno la quantità")
12 quantity.set( value: 1)
13 console.log( message: "Quantità aggiornata")
14 setTimeout(() => {
15   console.log(total()) // Il valore è ri-calcolato solo in questo momento
16 }, delay: 5000)
```

Ricalcolo

31.98

31.98

Ora aggiorno la quantità

Quantità aggiornata



# COMPUTED SIGNALS

```
1  const quantity : WritableSignal<number> = signal( initialValue: 2)
2  const price : WritableSignal<number> = signal( initialValue: 15.99)
3
4  const total : Signal<number> = computed(()=> {
5    console.log( message: "Ricalcolo")
6    return price() * quantity()
7  })
8
9  console.log(total()) // Il valore è calcolato solo in questo momento
10 console.log(total())
11 console.log( message: "Ora aggiorno la quantità")
12 quantity.set( value: 1)
13 console.log( message: "Quantità aggiornata")
14 setTimeout(() => {
15   console.log(total()) // Il valore è ri-calcolato solo in questo momento
16 }, delay: 5000)
```

Ricalcolo

31.98

31.98

Ora aggiorno la quantità

Quantità aggiornata

Ricalcolo

15.99



# COMPUTED SIGNALS

Le **dipendenze** di un computed signal sono **dinamiche**: solo i segnali letti per calcolarne il valore sono effettivamente tracciati

```
1  const showCount : WritableSignal<boolean> = signal( initialValue: false);
2  const count : WritableSignal<number> = signal( initialValue: 0);
3
4  const conditionalCount : Signal<string> = computed(() => {
5      console.log( message: "Ricalcolo");
6
7      if (showCount()) {
8          return `Il count vale ${count()}.`; // count è tracciato solo se showCount è true
9      } else {
10         return 'Niente da mostrare!';
11     }
12 });
13
14 console.log("1." + conditionalCount());
15 count.set( value: 1); // Il valore non viene ricalcolato
16 console.log("2." + conditionalCount());
```

Ricalcolo

1.Niente da mostrare!

2.Niente da mostrare!



# COMPUTED SIGNALS

Ma cosa significa che una dipendenza **cambia valore**?

```
const showCount : WritableSignal<boolean> = signal( initialValue: false);  
showCount.set( value: false); // il valore non è cambiato per Angular  
showCount.set( value: true); // il valore è cambiato per Angular
```





# COMPUTED SIGNALS

Ma cosa significa che una dipendenza **cambia valore**?

```
const showCount : WritableSignal<boolean> = signal( initialValue: false);  
showCount.set( value: false); // il valore non è cambiato per Angular  
showCount.set( value: true); // il valore è cambiato per Angular
```

```
const obj : WritableSignal<{}> = signal({});  
obj.set(obj()) // il valore non è cambiato per Angular  
obj.set({}) // i due oggetti non sono lo stesso oggetto => il valore è cambiato per Angular
```

Angular utilizza **Object.is(prevVal, nextVal)** per determinare se il valore impostato è diverso dal precedente.



# COMPUTED SIGNALS

**Non includere** mai nell'espressione di un computed signal variabili che non siano segnali: aggiornamenti di queste variabili non sono tracciati e non determineranno un update del valore.



# COMPUTED SIGNALS

**Non includere** mai nell'espressione di un computed signal variabili che non siano segnali: aggiornamenti di queste variabili non sono tracciati e non determineranno un update del valore.

```
welcomeMessage : Signal<string> = computed(() => {  
  // Buongiorno o Buenasera a seconda dell'ora  
  const date = new Date();  
  const hour : number = date.getHours();  
  if (hour < 12) {  
    return `Buongiorno ${this.name()}`;  
  } else {  
    return `Buonasera ${this.name()}`;  
  }  
})
```

*L'utente entra nella pagina e non fa nulla per 5 minuti*

h. 11.59

Buongiorno Luca

h. 12.00

Buonasera Luca



# COMPUTED SIGNALS

**Non includere** mai nell'espressione di un computed signal variabili che non siano segnali: aggiornamenti di queste variabili non sono tracciati e non determineranno un update del valore.

```
time : WritableSignal<Date> = signal(new Date());

welcomeMessage : Signal<string> = computed(() => {
  const hour : number = this.time().getHours();
  if (hour < 12) {
    return `Buongiorno ${this.name()}`;
  } else {
    return `Buonasera ${this.name()}`;
  }
});

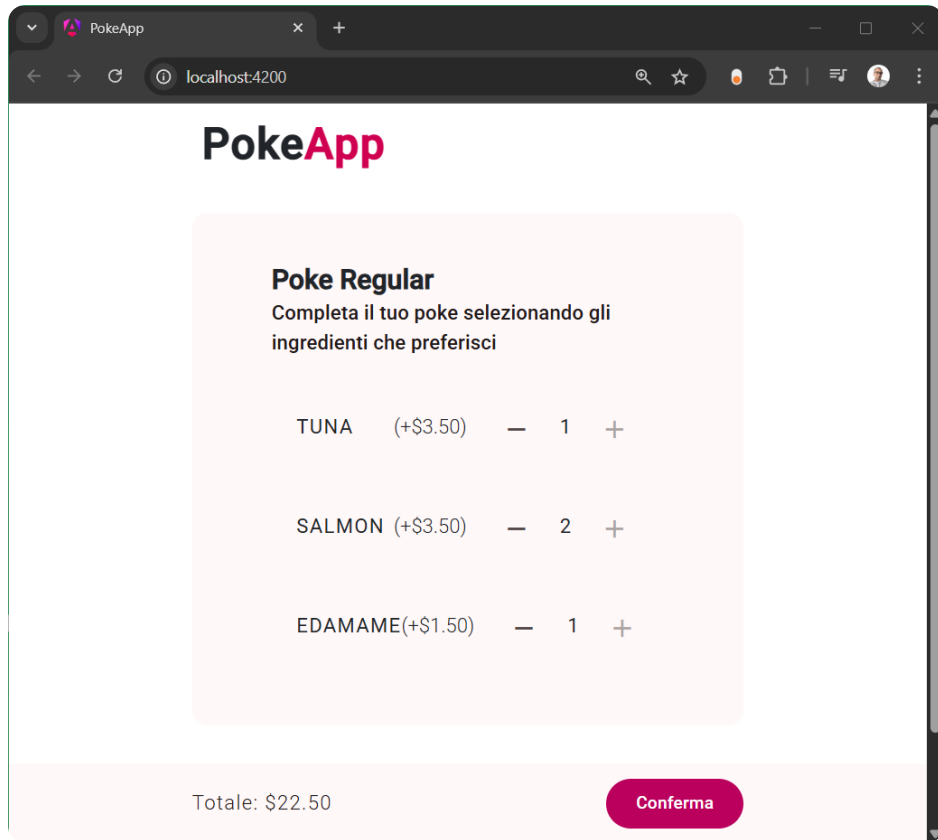
ngOnInit() {
  no usages new *
  setInterval(() => this.time.set(new Date()), delay: 60000); // Aggiorna l'ora ogni minuto
}
```



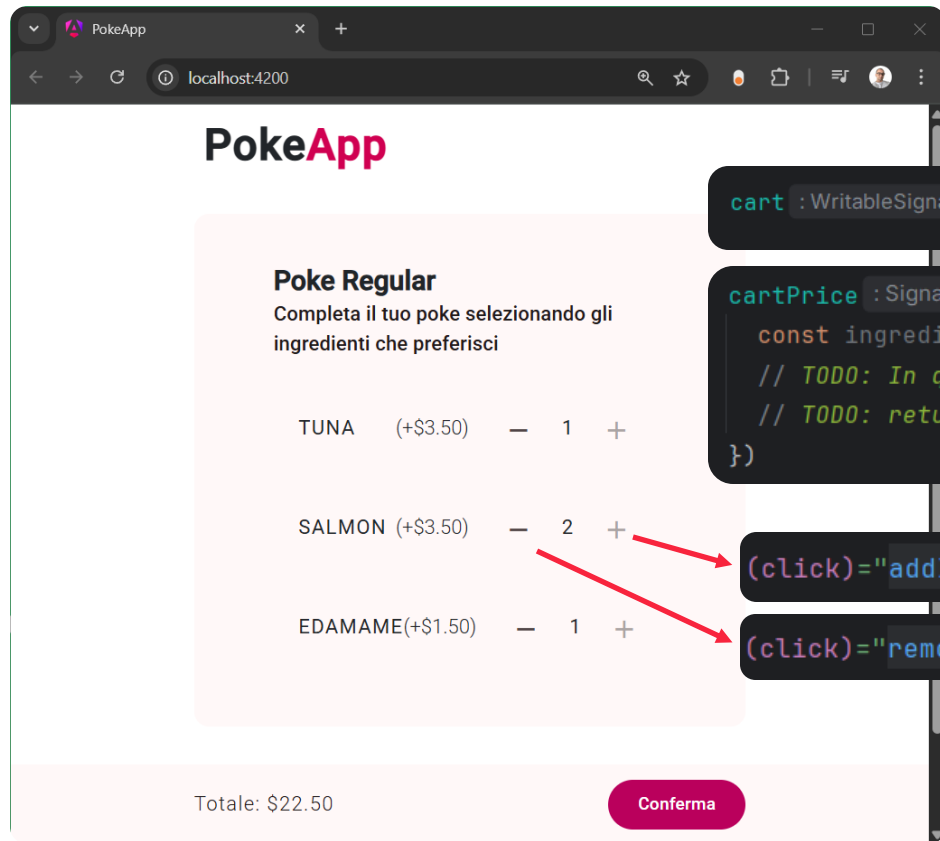
# ESEMPIO



## L'ESEMPIO CHE CONSIDEREREMO:



## L'ESEMPIO CHE CONSIDEREREMO:



```
cart : WritableSignal<PokeCart> = signal<PokeCart>({ingredients: []});
```

```
cartPrice : Signal<void> = computed(()=>{  
    const ingredientsSelected : PokeCartIngredient[] = this.cart().ingredients;  
    // TODO: In qualche modo calcolare il prezzo totale  
    // TODO: return price;  
})
```

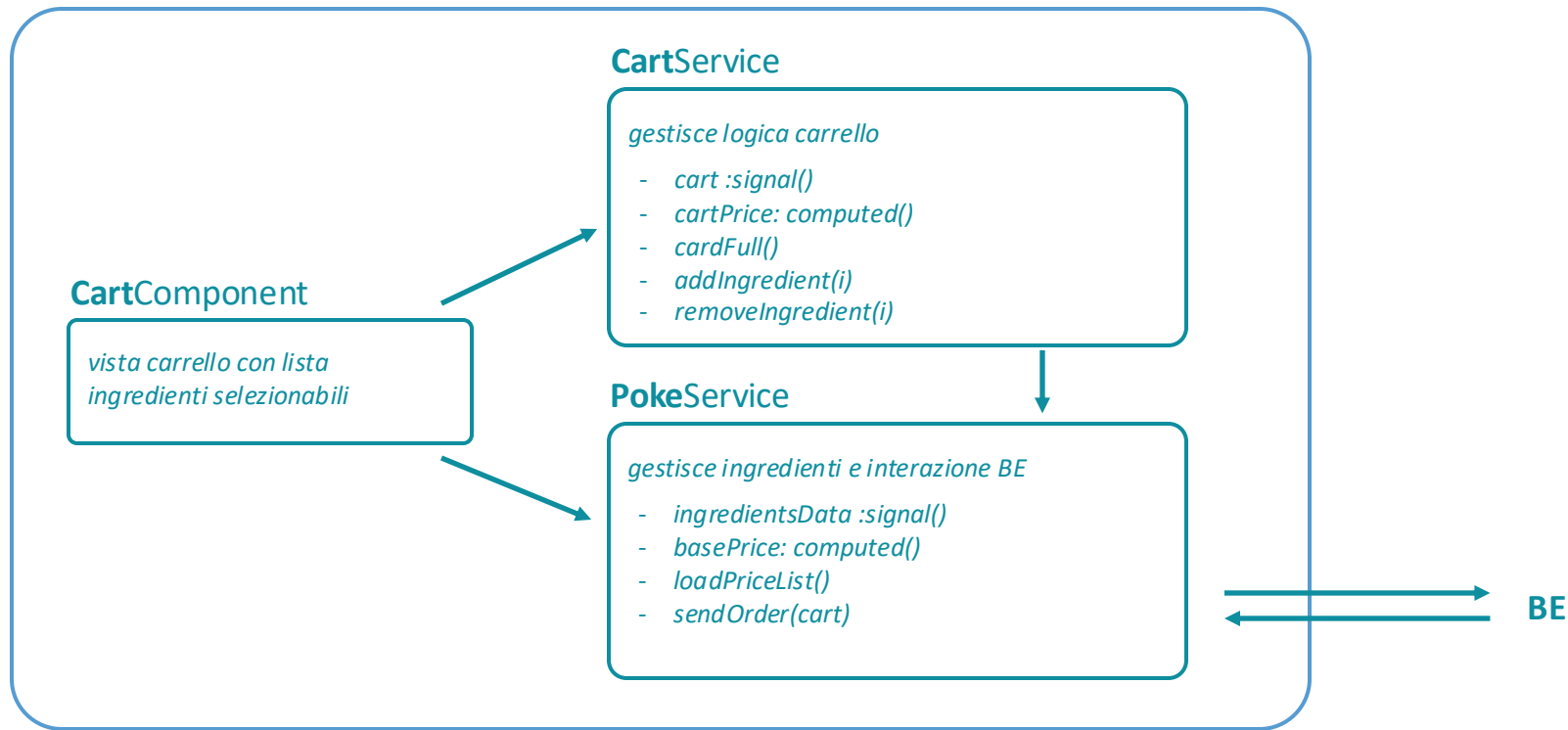
```
(click)="addIngredient(ingredient)"
```

```
(click)="removeIngredient(ingredient)"
```



## L'ESEMPIO CHE CONSIDEREREMO:

### Design generale





# DEEP DIVE



# NEXT UP: EFFECTS



# EFFECTS

Possiamo **tracciare** uno o più segnali anche in altri modi:

Un **effetto** è una operazione che viene eseguita ogni volta che uno o più segnali da cui dipende cambiano

```
effect(() => {  
  const cart :PokeCart = this.cartService.cart()  
  console.log( message: "Cart changed", cart)  
});
```

Effect function

Tracked dependency



# EFFECTS

```
effect(() => {  
  const cart :PokeCart = this.cartService.cart()  
  console.log( message: "Cart changed", cart)  
});
```

## Poke Regular

TUNA	(+\$3.50)	—	2	+
SALMON	(+\$3.50)	—	1	+
EDAMAME	(+\$1.50)			+

```
Cart changed {"ingredients":[{"id":1,"quantity":1}]}  
Cart changed {"ingredients":[{"id":1,"quantity":2}]}  
Cart changed {"ingredients":[{"id":1,"quantity":2},{"id":2,"quantity":1}]}
```



# EFFECTS

Per cosa usarli:

- Logging
- Sincronizzazioni «complicate» (ad esempio: forms)

```
// Servizio che gestisce le informazioni dell'utente
protected readonly userService :UserService = inject(UserService);

// Definizione campo nome della bowl
bowlNameCtrl :FormControl<string | null> = new FormControl( value: '');

constructor() { no usages ⓘiminotti-reply*
  // Sincronizza il campo nome della bowl con il nome del cliente
  effect(() => {
    const customerName :string = this.userService.customerName()
    this.bowlNameCtrl.setValue(customerName);
  });
}
```



# EFFECTS

- Vanno definiti in un injection context (tipicamente il **costruttore**)
- Sono sempre **eseguiti almeno una volta** (alla registrazione)
- **Effect batching**: aggiornamenti sincroni con i segnali tracciati ma solo una volta stabilizzate le dipendenze

```
a : WritableSignal<number> = signal( initialValue: 1);  
b : WritableSignal<number> = signal( initialValue: 2);  
  
constructor() { no usages new *  
  effect(() => {  
    console.log( message: '> Effetto che traccia A e B', this.a(), this.b());  
  });  
}  
  
myFunction(){ Show usages new *  
  console.log( message: '>MyFunction: START');  
  this.a.set( value: 2);  
  this.b.set( value: 3);  
  console.log( message: '>MyFunction: END');  
}
```

```
> Effetto che traccia A e B 1 2  
>MyFunction: START  
>MyFunction: END  
> Effetto che traccia A e B 2 3
```



# ESEMPIO



# EFFECTS

## Per cosa non usarli:

- Calcolare valori derivati → usa i computed signals
- Side effects complessi

In generale, **evitare** quando possibile di settare altri segnali all'interno di un effetto.

↳ Peggiora manutenibilità e rischio di creare **loop**





# LINKED SIGNALS

Un **linked signal** permette di creare un segnale «intrinsecamente collegato» ad un altro segnale.

- *Rappresenta un ibrido tra un signal e un computed signal*

```
// Possibili taglie della bowl
bowlSizes : WritableSignal<string[]> = signal(['small', 'medium', 'large']);

// Dimensione della bowl selezionata
selectedBowlSize : WritableSignal<string> = linkedSignal(()=>{
  const smallest : string = this.bowlSizes()[0]
  return smallest;
});
```

Reset function

Tracked dependency



# LINKED SIGNALS

```
1 // Possibili taglie della bowl
2 const bowlSizes : WritableSignal<string[]> = signal(['small', 'medium', 'large']);
3 // Dimensione della bowl selezionata
4 const selectedBowlSize : WritableSignal<string> = linkedSignal(()=>{
5   return bowlSizes()[0];
6 });
7
8 console.log(selectedBowlSize()); // 'small'
9
10 // Cambia la dimensione della bowl
11 selectedBowlSize.set( value: 'medium');
12 console.log(selectedBowlSize()); // 'medium'
13
14 // Cambia dimensioni disponibili
15 bowlSizes.set(['small', 'medium', 'large', 'extra-large']);
16 console.log(selectedBowlSize()); // 'small'
```



# LINKED SIGNALS

```
1 // Possibili taglie della bowl
2 const bowlSizes : WritableSignal<string[]> = signal(['small', 'medium', 'large']);
3 // Dimensione della bowl selezionata
4 const selectedBowlSize : WritableSignal<string> = linkedSignal(()=>{
5   return bowlSizes()[0];
6 });
7
8 console.log(selectedBowlSize()); // 'small'
9
10 // Cambia la dimensione della bowl
11 selectedBowlSize.set( value: 'medium');
12 console.log(selectedBowlSize()); // 'medium'
13
14 // Cambia dimensioni disponibili
15 bowlSizes.set(['small', 'medium', 'large', 'extra-large']);
16 console.log(selectedBowlSize()); // 'small'
```

Quando una delle dipendenze tracciate cambia, il linked signal viene «resettato» tramite la sua funzione di inizializzazione.



# **SIGNA-BASED COMPONENT API**



# *SIGNAL-BASED* **COMPONENT API**

**API basate sui segnali** che sostituiscono le precedenti basate sull'uso delle annotazioni (@Input, @Output, @ViewChild,...).

- ***input()***: gestione input
- ***output()***: gestione output
- ***model()***: gestione two-way binding
- ***viewChild()***: view queries
- ***contentChild()***: content queries



# @INPUT() → INPUT()

## VECCHIA SINTASSI

```
3  @Component({ Show usages new *
4  selector: 'hello',
5  template: `Hello {{ upper }}` // Hello LUCA
6  })
7  export class HelloComponent implements OnChanges {
8    @Input() name!: string;
9    upper : string = '';
10
11  ngOnChanges(changes: SimpleChanges) {
12    if (changes['name']) {
13      this.upper = this.name.toUpperCase();
14    }
15  }
16 }
```

Decorator

Lifecycle hook

Boilerplate

Derivazione manuale



# @INPUT() → INPUT()

NUOVA SINTASSI

```
3 @Component({ Show usages new *
4   selector: 'hello',
5   template: `Hello {{ upper() }}`
6 })
7 export class HelloComponent {
8   name : InputSignal<string | undefined> = input<string>(); // Signal<string>
9   upper : Signal<string | undefined> = computed(() => this.name()?.toUpperCase());
10 }
```

**input()** definisce un segnale opzionale *readonly* di input del tipo specificato

E' possibile passare in input anche un valore di default e opzioni, in maniera analoga a quanto fosse possibile con @Input.



# @INPUT() → INPUT()

NUOVA SINTASSI

```
3  @Component({ Show usages new *
4  selector: 'hello',
5  template: `Hello {{ upper() }}`
6  })
7  export class HelloComponent {
8    name : InputSignal<string> = input.required<string>(); // Signal<string>
9    upper : Signal<string> = computed(() => this.name().toUpperCase());
10 }
```

**input()** definisce un segnale opzionale *readonly* di input del tipo specificato

E' possibile passare in input anche un valore di default e opzioni, in maniera analoga a quanto fosse possibile con @Input.

**E' possibile anche dichiarare l'input come obbligatorio.**





# @INPUT() → INPUT()

## UTILIZZO

```
20 ⓘ selector: 'hello-container',  
21 ⓘ template: `<hello [name]="userName"/>`,  
22 ⓘ imports: [HelloComponent]  
23 })  
24 export class HelloContainerComponent {  
25   userName : string = 'Luca';
```



# @OUTPUT() → OUTPUT()

VECCHIA SINTASSI

```
3  @Component({ Show usages new *
4  ④↑ selector: 'counter',
5  ④↑ template: `<button (click)="inc()">+1</button>`
6  })
7  export class CounterComponent {
8    @Output() changed : EventEmitter<number> = new EventEmitter<number>();
9    count : number = 0;
10
11   inc() { Show usages new *
12     this.count++;
13     this.changed.emit(this.count);
14   }
15 }
```

Decorator



# @OUTPUT() → OUTPUT()

NUOVA SINTASSI

```
14 @Component({ Show usages new *
15   selector: 'counter',
16   template: `<button (click)="inc()">+1</button>`
17 })
18 export class CounterComponent {
19   count : WritableSignal<number> = signal( initialValue: 0);
20   changed : OutputEmitterRef<number> = output<number>();
21
22   inc() { Show usages new *
23     this.count.update(c : number => c + 1);
24     this.changed.emit(this.count());
25   }
26 }
```

**output()** definisce un  
OutputEmitterRef

Non è un segnale!

Come prima



# @OUTPUT() → OUTPUT()

UTILIZZO

```
0  @Component({ no usages new *
1  selector: 'counter-container',
2  template: `<counter (changed)="doSomething()" />`,
3  imports: [CounterComponent]
4  })
5  export class HelloContainerComponent {
6    doSomething() { Show usages new *
7      // TODO
8    }
9  }
```



# @INPUT()+@OUTPUT() → MODEL()

VECCHIA SINTASSI

```
14 @Component({ no usages new *
15   selector: 'toggle',
16   template: `<button (click)="toggle()">{{ checked ? 'ON' : 'OFF' }}</button>`
17 })
18 export class ToggleComponent {
19   @Input() checked : boolean = false;
20   @Output() checkedChange : EventEmitter<boolean> = new EventEmitter<boolean>();
21
22   toggle() { Show usages new *
23     this.checked = !this.checked;
24     this.checkedChange.emit(this.checked);
25   }
26 }
```

Decorator

Decorator



# @INPUT()+@OUTPUT() → MODEL()

NUOVA SINTASSI

```
14 @Component({ no usages new *
15   selector: 'toggle',
16   template: `<button (click)="toggle()">{{ checked() ? 'ON' : 'OFF' }}</button>`
17 })
18 export class ToggleComponent {
19   checked : ModelSignal<boolean> = model( initialValue: false); // Signal + paired output
20
21   toggle() { Show usages new *
22     this.checked.update(v : boolean => !v);
23   }
24 }
```




**model()** definisce un segnale *modificabile*

Crea automaticamente l'output associato al modello ed emette automaticamente l'evento di modifica.



# @INPUT()+@OUTPUT() → MODEL()

## UTILIZZO

```
30 @Component({ no usages new *
31  selector: 'toggle-container',
32  template: `
33     <toggle [(checked)]="checked" />
34     <toggle [(checked)]="checkedSignal" />
35     `
36  imports: [ToggleComponent]
37 })
38 export class HelloContainerComponent {
39     checked : boolean =false
40     checkedSignal : WritableSignal<boolean> = signal( initialValue: false)
41 }
```

← checkedSignal()



# @VIEWCHILD() → VIEWCHILD()

VECCHIA SINTASSI

```
15  ✓ @Component({ no usages new *
16  Ⓢ selector: 'search',
17  Ⓢ template: `<input #q :HTMLInputElement><button (click)="focus()">Focus</button>`
18  })
19  ✓ export class SearchComponent implements AfterViewInit {
20    @ViewChild( selector: 'q', { static: true }) q!: ElementRef<HTMLInputElement>;
21
22  Ⓢ ngAfterViewInit() { no usages new *
23    |   this.q.nativeElement.focus();
24    | }
25
26  ✓ focus() { Show usages new *
27    |   this.q.nativeElement.focus();
28    | }
29  }
```

Decorator

Lifecycle hook





# @VIEWCHILD() → VIEWCHILD()

NUOVA SINTASSI

```
15 @Component({ no usages new *
16   selector: 'search',
17   template: `<input #q :HTMLInputElement><button (click)="focus()">Focus</button>`
18 })
19 export class SearchComponent {
20   q : Signal<ElementRef<HTMLInputElement>> = viewChild.required<ElementRef<HTMLInputElement>>( locator: 'q');
21
22   constructor() { no usages new *
23     effect(() => this.q()?.nativeElement.focus());
24   }
25
26   focus() { Show usages new *
27     this.q().nativeElement.focus();
28   }
29 }
```

Se il target non è mostrato condizionatamente (fuori da @if)

**viewChild()** definisce un segnale aggiornato con il risultato della view query  
Analogamente per query con risultati multipli, **@viewChildren** -> **viewChildren()**



# @CONTENTCHILD() → CONTENTCHILD()

## UTILIZZO

```
56 @Component({ no usages new *
57   ↑ selector: 'app',
58   ↑ template: `
59     <app-panel>
60       <h2 #titleRef :HTMLHeadingElement >Angular Signals</h2>
61       <p>Altro contenuto</p>
62     </app-panel>
63   `,
64   ↑ imports: [PanelComponent]
65 })
66 export class AppComponent {}
```



# @CONTENTCHILD() → CONTENTCHILD()

VECCHIA SINTASSI

```
32  ✓ @Component({ Show usages new *
33  Ⓢ↑ selector: 'app-panel',
34  Ⓢ↑ template: `
35  ✓   <div class="body">
36  |   <ng-content><!-- Il contenuto verrà proiettato qua --></ng-content>
37  |   </div>
38  |   `
39  })
40  ✓ export class PanelComponent implements AfterContentInit {
41  |   @ContentChild(selector: 'titleRef') title?: ElementRef;
42  |
43  Ⓢ↑ ngAfterContentInit() { no usages new *
44  |   if (this.title)
45  |   |   console.log(message: 'Title projected:', this.title.nativeElement.textContent);
46  |   }
47  }
```

Decorator

Lifecycle hook



# @CONTENTCHILD() → CONTENTCHILD()

NUOVA SINTASSI

```
33 @Component({ Show usages new *
34 selector: 'app-panel',
35 template: `
36   <div class="body">
37     <ng-content><!-- Il contenuto verrà proiettato qua --></ng-content>
38   </div>
39 `
40 })
41 export class PanelComponent {
42   title : Signal<ElementRef<any> | undefined> = contentChild<ElementRef>({ locator: 'titleRef' });
43
44   constructor() { no usages new *
45     effect(() => {
46       const el : ElementRef<any> | undefined = this.title();
47       if (el) console.log( message: 'Title projected:', el.nativeElement.textContent);
48     });
49   }
50 }
```

**contentChild()** definisce un segnale aggiornato con il risultato della content query

Analogamente per query con risultati multipli,  
**@contentChildren** -> **contentChildren()**



# WRAP UP

- Reattività **nativa** e **leggibile**: niente più subscribe() o async pipe

```
// Prima (Observable)
count$ = new BehaviorSubject(0);
ngOnInit() {
  this.count$.subscribe(v => this.value = v);
}

// Dopo (Signal)
count = signal(0);
```



# WRAP UP

- Reattività **nativa** e **leggibile**: niente più subscribe() o async pipe
- Meno **boilerplate**: meno decoratori, niente lifecycle extra



# WRAP UP

- Reattività **nativa** e **leggibile**: niente più subscribe() o async pipe
- Meno **boilerplate**: meno decoratori, niente lifecycle extra
- **Change detection** più efficiente



# CHANGE DETECTION

La change detection è il «processo tramite il quale Angular mantiene **l'interfaccia utente sincronizzata con i dati**».

Basato su **zone.js** (intercetta tutti gli eventi asincroni del browser quali click, timer,... e avvisa Angular).

**Evento click → zone.js intercetta → Angular controlla *tutto* il tree**





# CHANGE DETECTION

Con i segnali:

```
<div>{{ count() }}</div>
```

```
count = signal(0);  
increment() { this.count.update(c => c + 1); }
```

**Evento:** cambia count()

**Effetto:** Solo il `<div>{{ count() }}</div>` viene aggiornato, non il resto dell'app



