

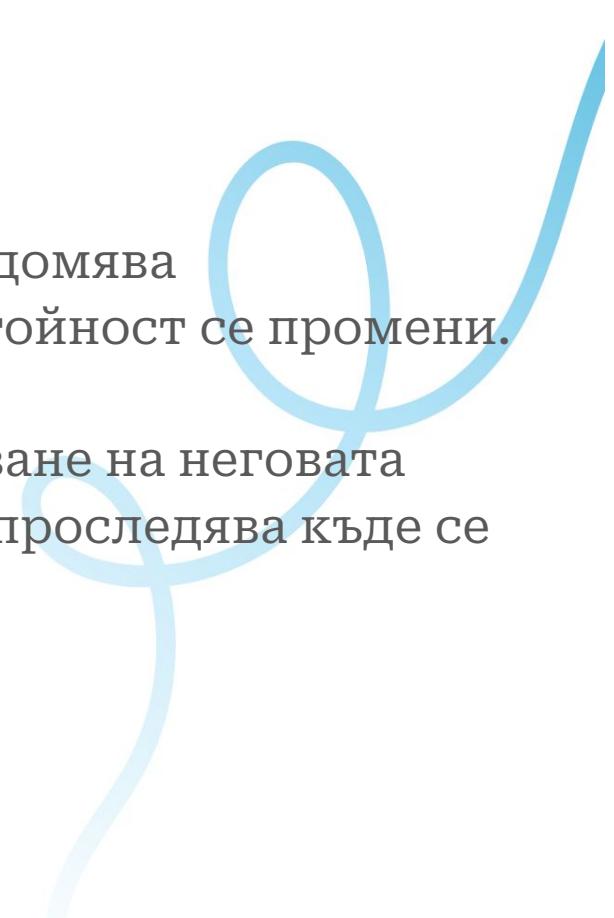
# Angular Signals

Лектор: Петър Маламов

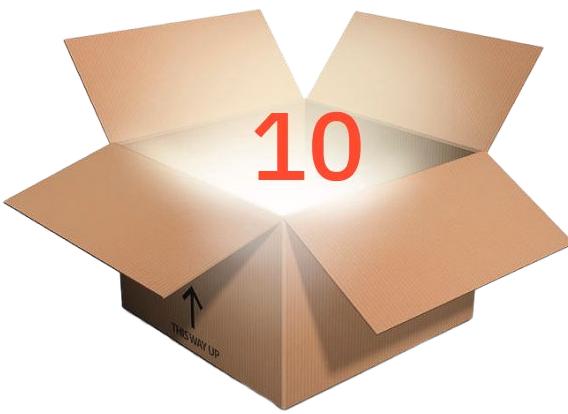
# Какво са Signals ?

Сигналът е обвивка около стойност, която уведомява заинтересованите потребители, когато тази стойност се промени.

Стойността на сигнала се прочита чрез извикване на неговата getter функция, което позволява на Angular да проследява къде се използва сигналът.



**Signal = value + change notification**



```
const count = 0;  
console.log(count);
```

```
const count = signal(0);  
console.log(count());
```

**Signal getter  
function**

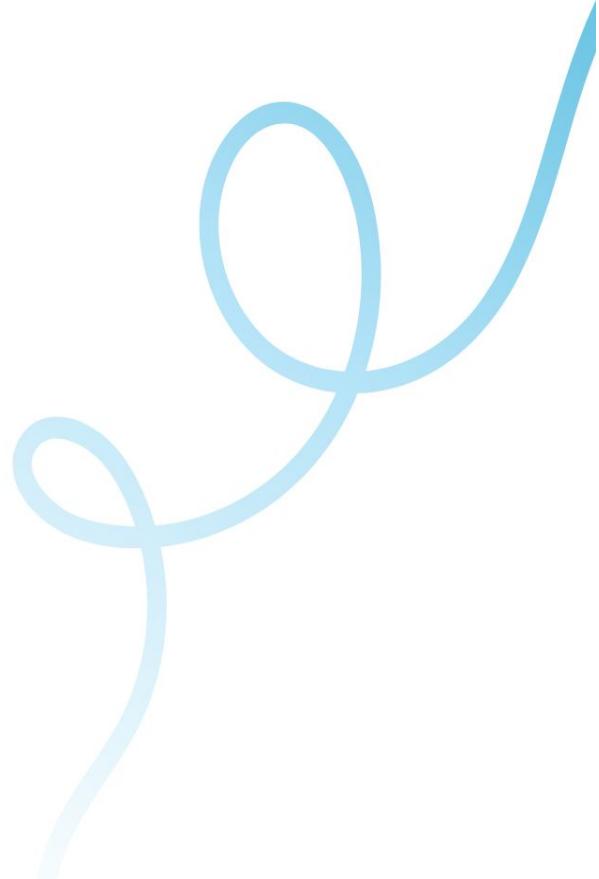
# Важно!

- Сигналите са reactive
- Сигналите винаги имат стойност
- Сигналите могат да съдържат всякакви стойности – от примитиви до сложни структури от данни.

```
const count = signal<number[]>([1,2,3,4]);
```

# Типове сигнали

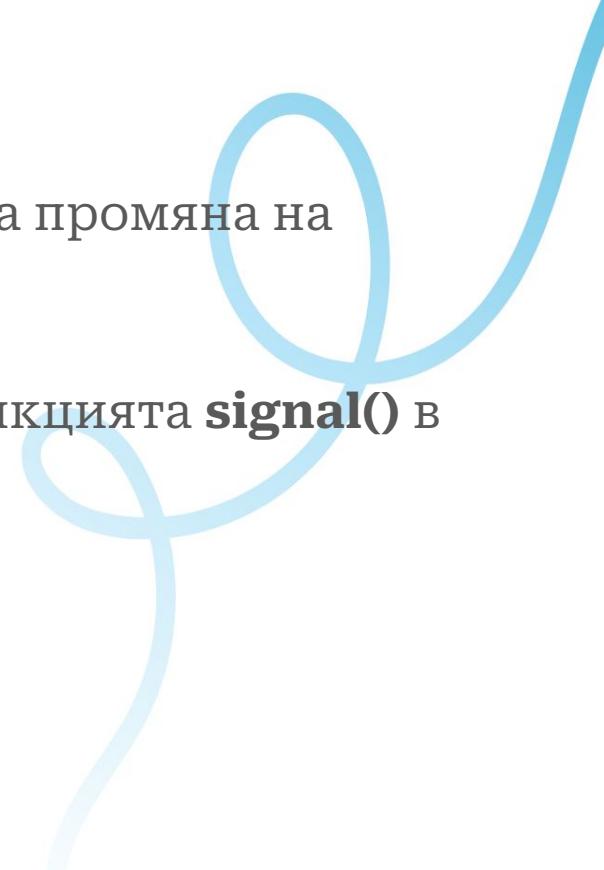
- Writable signals
- Read-only



# Writable signals

Сигнали, които предоставят начин за директна промяна на тяхната стойност.

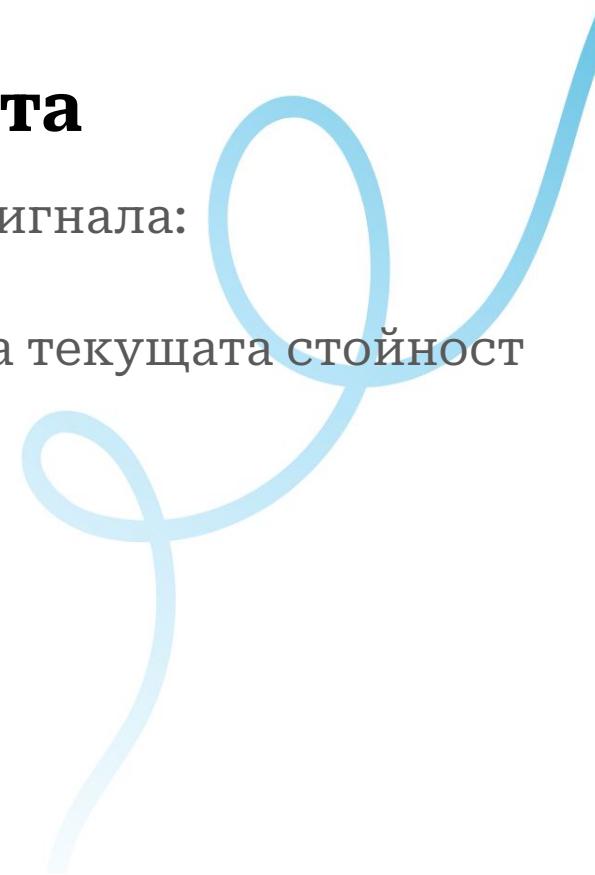
Writable signal може да бъде създаден чрез функцията **signal()** в Angular.



# Начини за промяна на стойността

Има два начина за промяна на стойността на сигнала:

- **set** – директно променя стойността
- **update** – променя стойността въз основа на текущата стойност на сигнала.



TS app.component.ts

```
@Component({
  selector: 'app-root',
  standalone: true,
  imports: [RouterOutlet],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})
export class AppComponent {
  title = '06-signals';

  count = signal(0);
```

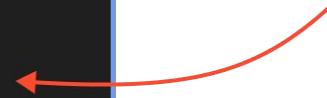
```
    resetValue() {
      this.count.set(0);
    }
```

```
    increment() {
      this.count.update((n) => n + 1);
    }
}
```

Директна промяна



Промяна въз основа  
на текущата стойност



# **Важно!**

Когато сигнал се използва в темплейт, той се регистрира като зависимост на изгледа. Ако сигналът се промени, изгледът се променя също (rereder).

Ако стойността на сигнала се промени няколко пъти, преди да бъде прочетена, потребителите ще получат само последната стойност.

## Промяна

```
TS app.component.ts

  @Component([
    selector: 'app-root',
    standalone: true,
    imports: [RouterOutlet],
    templateUrl: './app.component.html',
    styleUrls: ['./app.component.css'],
  ])
  export class AppComponent {
    title = '06-signals';

    count = signal(0);

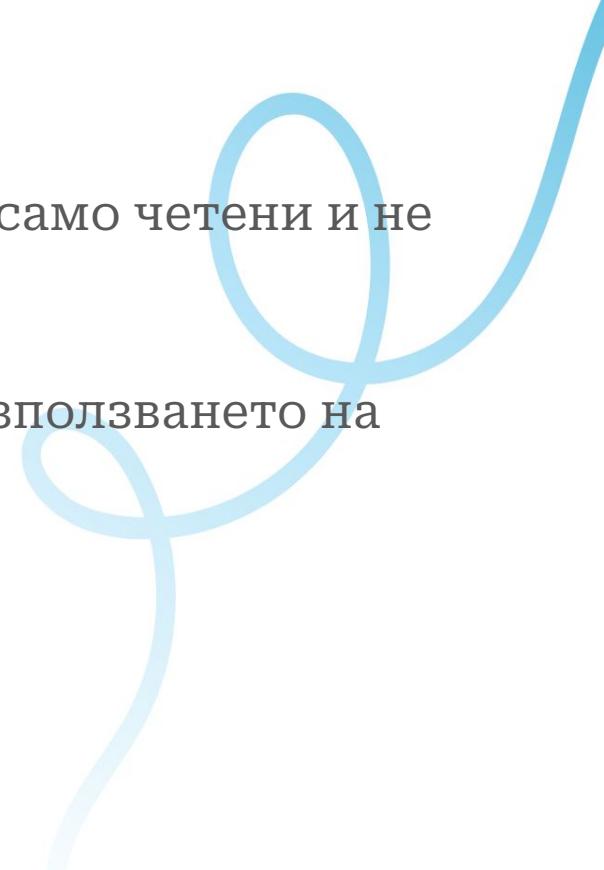
    setCount() {
      this.count.set(0);
      this.count.set(5);
      this.count.set(10);
    }

    increment() {
      this.count.update((n) => n + 1);
    }
  }
```

# Read-only signals

Представляват сигнали, които могат да бъдат само четени и не позволяват директна промяна на стойността.

Read-only signal може да бъде създаден чрез използването на  
**computed signal**

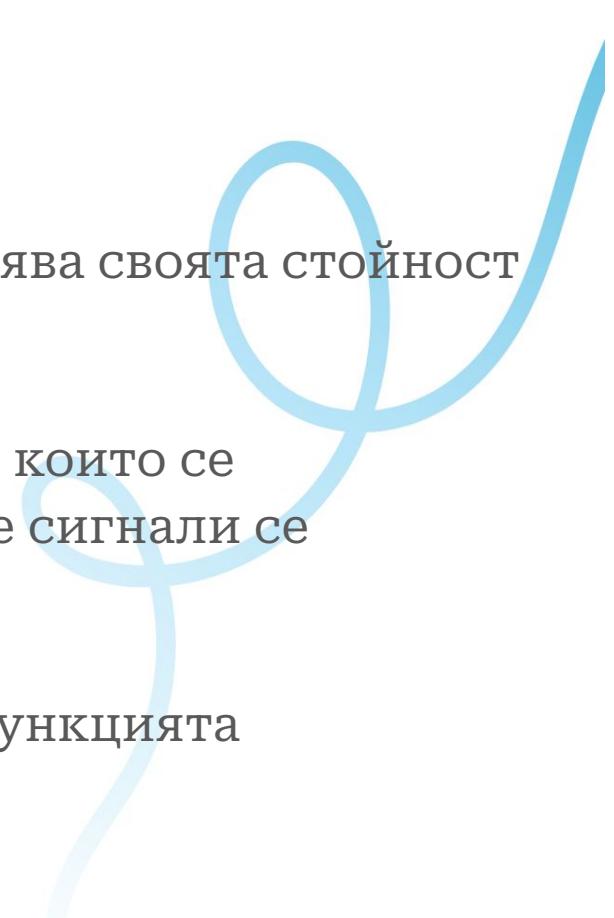


# Computed signal

Сигнал, който автоматично изчислява и обновява своята стойност на база стойностите на други сигнали.

Той позволява създаването на нови стойности, които се актуализират автоматично, когато зависимите сигнали се променят.

Computed signal може да бъде създаден чрез функцията **computed()** в Angular.



TS app.component.ts

```
@Component({
  selector: 'app-root',
  standalone: true,
  imports: [RouterOutlet],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})
export class AppComponent {
  title = '06-signals';

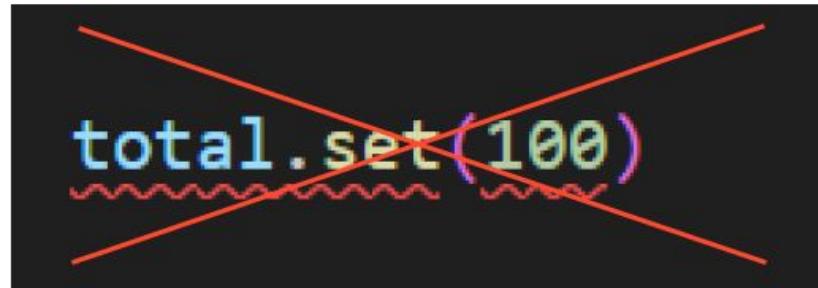
  price = signal<number>(99.99);
  quantity = signal<number>(1);

  total = computed(() => this.price() * this.quantity());
}
```



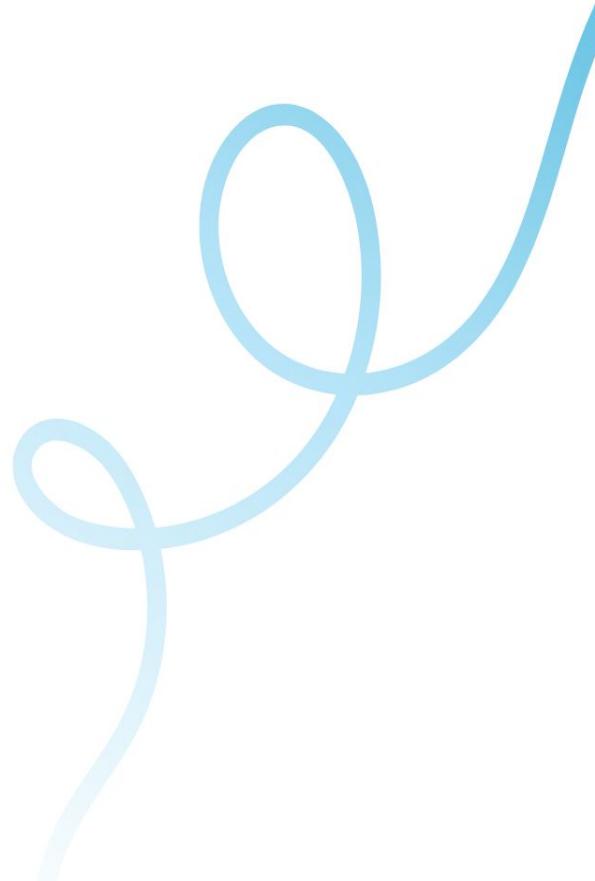
www.diadraw.com

Стойностите в computed signals не могат да бъдат променяни ръчно!



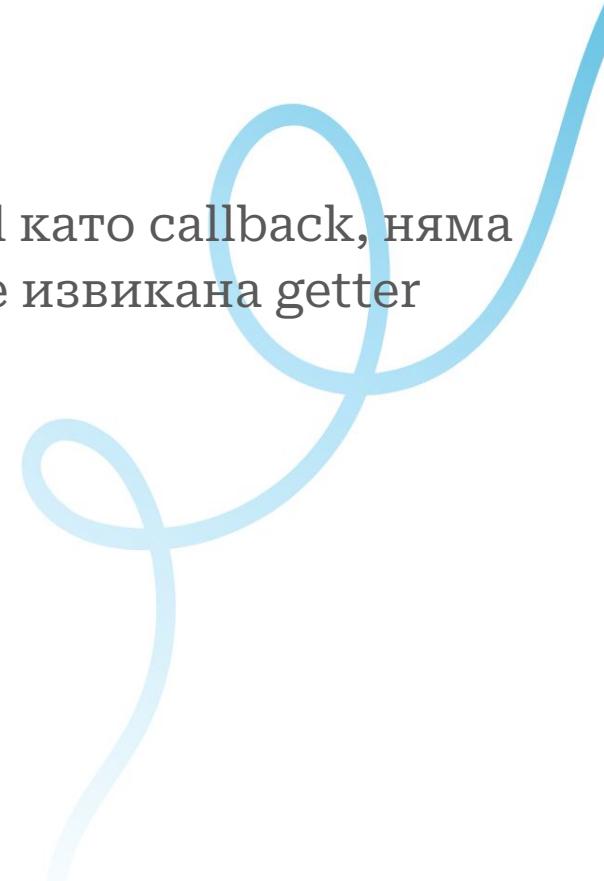
# Computed signal ca:

- Lazily evaluated
- Memoized



# Lazily evaluated

Функцията, която подаваме на computed signal като callback, няма да бъде извикана/калкулирана, докато не бъде извикана getter функцията на сигнала.



TS app.component.ts

```
count = signal(0);
double = computed(() => {
  console.log('in computed signal');
  return this.count() * 2;
});
```

Console



<> app.component.html

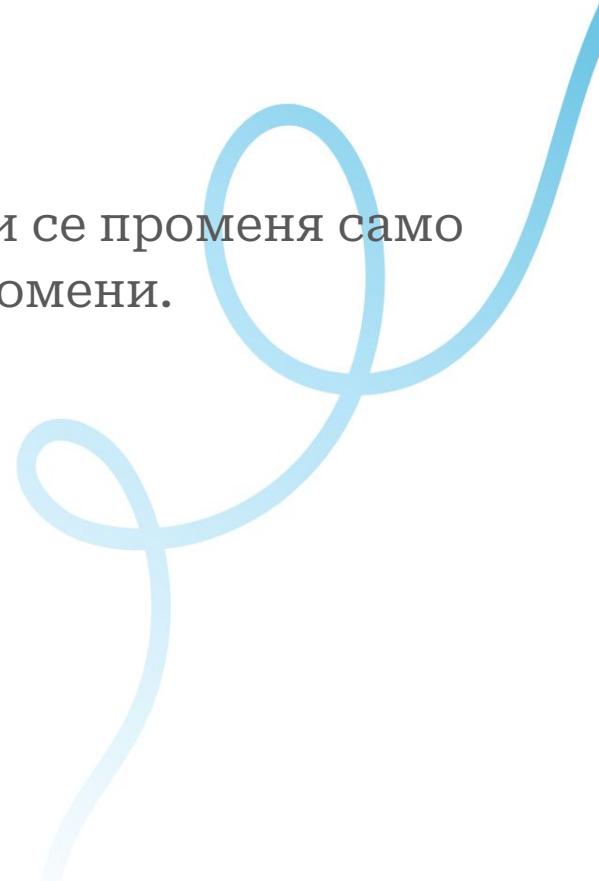
```
{{double()}}
```

Console

in computed signal

# Memoized

Калкулираната стойност в сигнала се кешира и се променя само ако някой от сигналите, от които зависи, се промени.



TS app.component.ts

```
count = signal(0);
double = computed(() => {
  console.log('in computed signal');
  return this.count() * 2;
});
```

Console



<> app.component.html

```
{{double()}}
{{double()}}
{{double()}}
```

Console

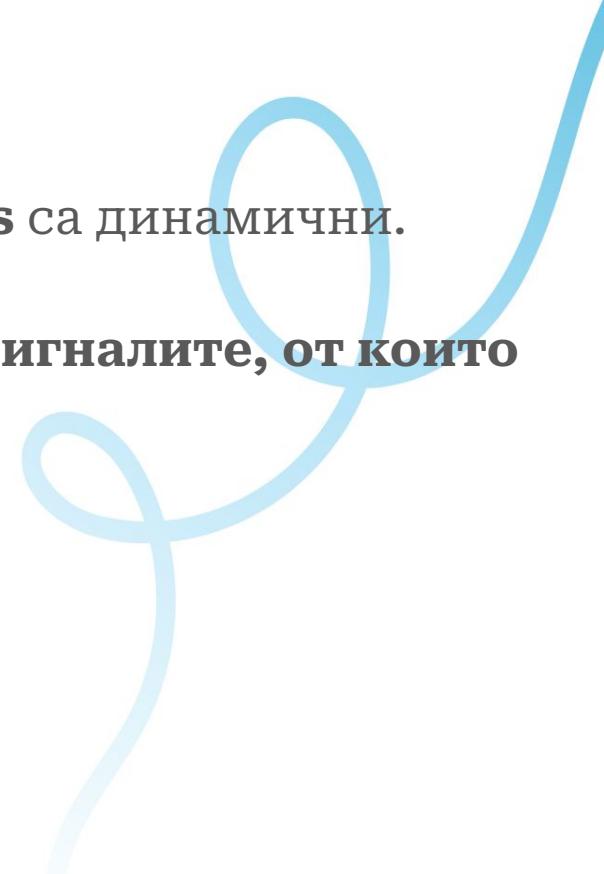
in computed signal



# Важно!

Зависимостите използвани в **computed signals** са динамични.

**Computed signals** следят за промени само в сигналите, от които са прочели стойност.



TS app.component.ts

```
export class AppComponent {
  title = '06-signals';

  quantity = signal(0);
  price = signal(99.99);

  itemLabel = computed(() => {
    if (this.quantity() < 1) {
      return 'There are no items';
    }

    return `The price is ${this.price()}`;
  });
}
```

Ако quantity остане по-малко от 1,  
тогава сигналът няма да прочете  
price.

Ако сигналът не прочете price,  
той няма да следи за промени в  
него.

# Как се определя промяна в signal ?

По подразбиране, сравнението, което се извършва, за да се определи дали даден сигнал е променен, се прави по референция с помощта на **Object.is()**.

Angular ни дава възможност да предоставим собствена функция за сравнение, която да определи дали има промяна в сигналите.

TS app.component.ts

```
export class AppComponent {
  title = '06-signals';

  count = signal(0, {
    equal: (a, b) => a === b,
  });

  updateCount() {
    this.count.update((c) => c + 1);
  }
}
```

# Важно!

Няма да отрази промяната, тъй като референцията остава същата.

TS app.component.ts

```
export class AppComponent {
  title = '06-signals';

  arr = signal([1, 2, 3, 4]);
  double = computed(() => {
    return this.arr().map((item) => item * 2);
  });

  addValue(newValue: number) {
    this.arr().push(newValue);
  }
}
```

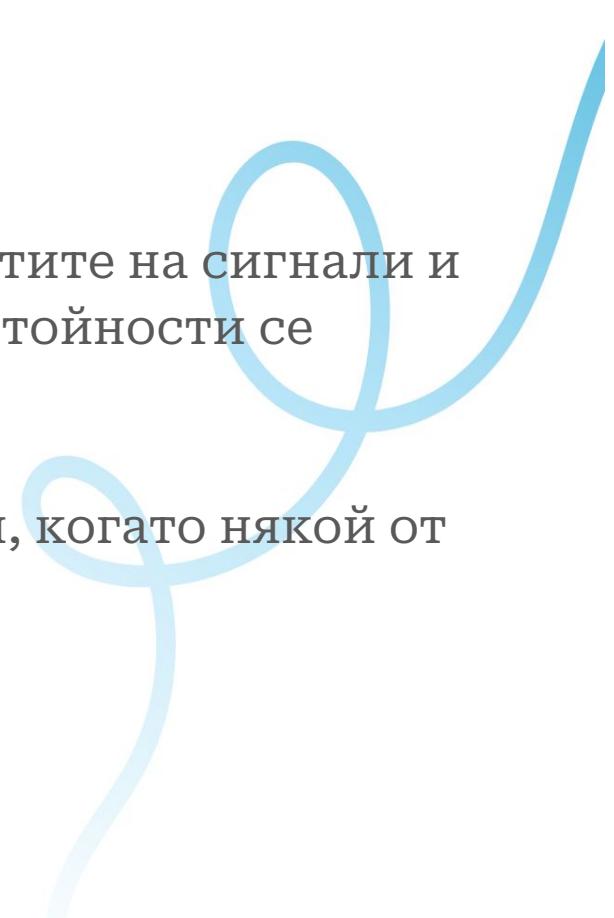


# Effects

# Какво е Effects?

Функция, която реагира на промени в стойностите на сигнали и изпълнява определени действия, когато тези стойности се променят.

Позволява извършването на странични ефекти, когато някой от следените сигнали се промени.



TS app.component.ts

```
export class AppComponent {  
  title = '06-signals';  
  
  count = signal(0);  
  
  constructor() {  
    effect(() => {  
      console.log(`The count is: ${this.count()}`);  
    });  
  }  
}
```

TS app.component.ts

```
export class AppComponent {  
  title = '06-signals';  
  
  count = signal(0);  
  
  loggingEffect: EffectRef = effect(() => {  
    console.log(`The count is: ${this.count()}`);  
  });  
}
```



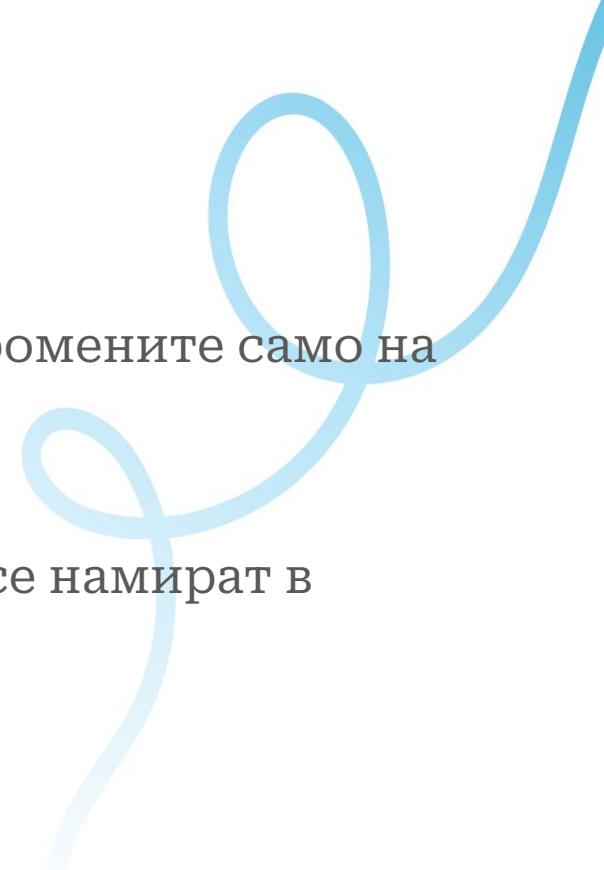
www.diadraw.com

# Важно!

**Effects** винаги се изпълняват поне веднъж.

Подобно на computed signals, **effects** следят промените само на сигналите, от които са прочели стойностите.

**Effects** могат да бъдат създавани само когато се намират в **Injection context!**



TS app.component.ts

```
export class AppComponent {  
  title = '06-signals';  
  
  count = signal(0);  
  
  constructor() {  
    effect(() => {  
      console.log(`The count is: ${this.count()}`);  
    });  
  }  
}
```

TS app.component.ts

```
export class AppComponent {  
  title = '06-signals';  
  
  count = signal(0);  
  
  loggingEffect: EffectRef = effect(() => {  
    console.log(`The count is: ${this.count()}`);  
  });  
}
```

Injection context

Можем да подадем Injector на effect функцията, за да може да работи извън извън Injection context!

```
TS app.component.ts
export class AppComponent {
  title = '06-signals';

  count = signal(0);

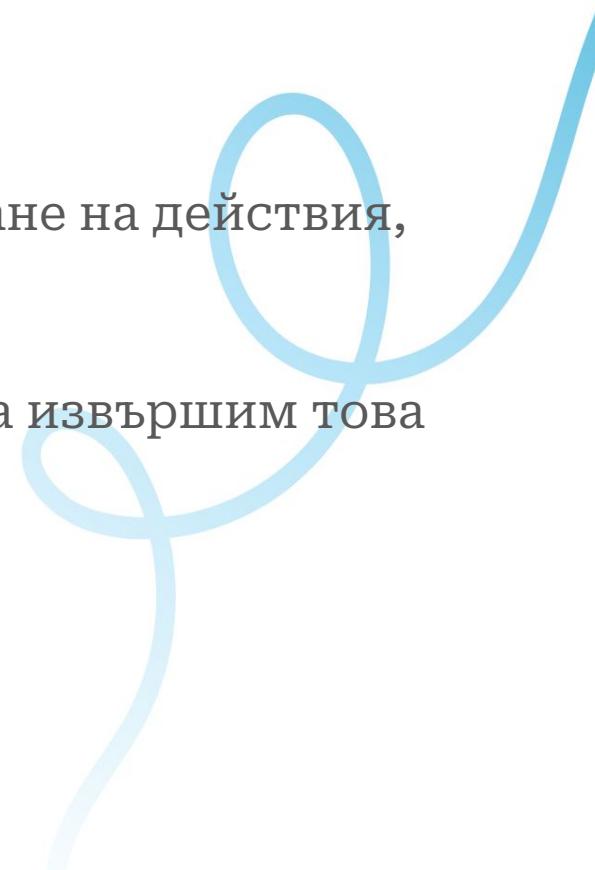
  constructor(private injector: Injector) {}

  someAction(){
    effect(() => {
      console.log(this.count());
    },{injector: this.injector})
  }
}
```

# Effect cleanup

Начин за почистване на ресурси или извършване на действия, когато един effect спре да бъде активен.

Angular предоставя функция, с която можем да извършим това почистване.



TS app.component.ts

```
export class AppComponent {
  title = '06-signals';

  userId = signal(0);
  someEffect = effect((onCleanup) => {
    const user = currentUser(this.userId());

    const timer = setTimeout(() => {
      console.log(`The current user is ${user}`);
    }, 1000);

    onCleanup(() => {
      clearTimeout(timer);
    });
  });
}
```

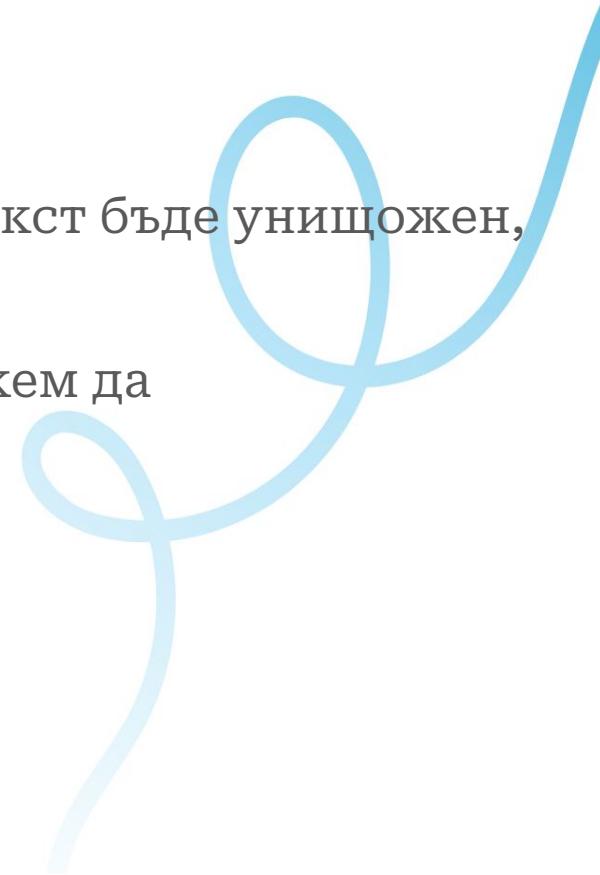


www.diadraw.com

# Премахване на effect

По подразбиране, когато обграждащият контекст бъде унищожен, effect-ът в него също се унищожава.

Angular ни предоставя начини, чрез които можем да контролираме кога effect-ът се премахва.



ts app.component.ts

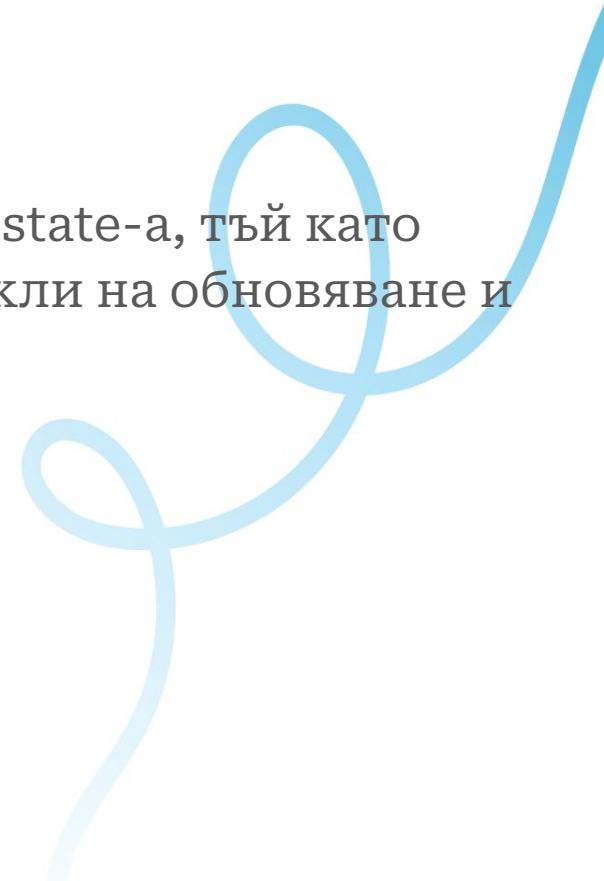
```
export class AppComponent {  
  title = '06-signals';  
  
  timerEffect = effect((onCleanup) => {  
    const timerId = setInterval(() => {  
      console.log('effect');  
    });  
  
    onCleanup(() => {  
      clearInterval(timerId);  
    });  
  }, {  
    manualCleanup: true  
  });  
  
  removeEffect() {  
    this.timerEffect.destroy();  
  }  
}
```

Отменя автоматичното премахване, когато компонентът бъде унищожен.

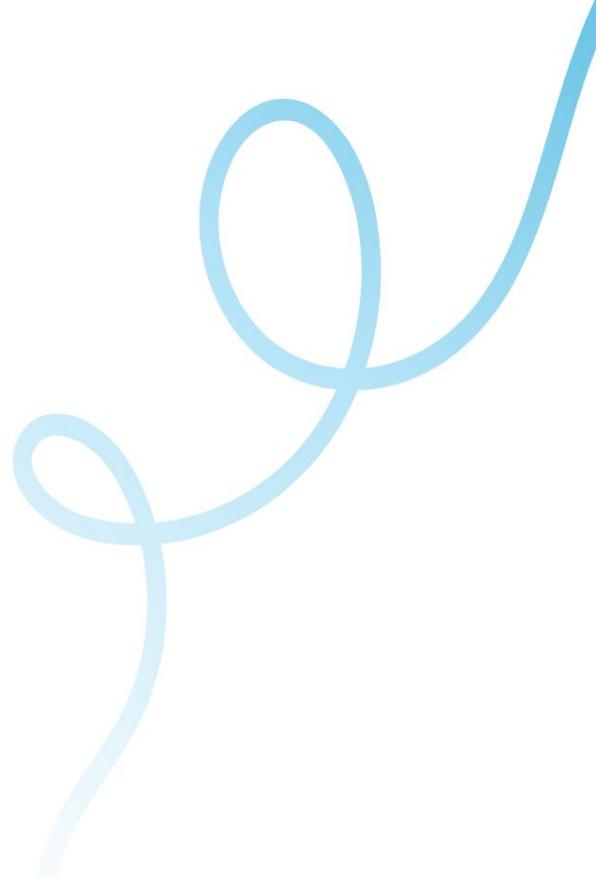
Ръчно премахване на effect чрез неговата референция.

# Важно!

Избягвайте да използвате effects за промени в state-а, тъй като това може да доведе до грешки, безкрайни цикли на обновяване и ненужни проверки за промени.



# Упражнение



# Signals vs RxJS

```
id = signal<number>(1);      =    id = new BehaviorSubject<number>(1); ?
```

```
id = signal<number>(1);
```

➤

```
id = new BehaviorSubject<number>(1);
```



[www.diadraw.com](http://www.diadraw.com)

```
template: `  
  {{ id() }}  
  {{ id$ | async }}  
`
```

Използване на signal

Използване на  
observable

TS app.component.ts

```
count = signal<number>(1);
doubleCount = computed(() => {
  return this.count() * 2;
});

countSubject = new BehaviorSubject<number>(1);
doubleCount$ = this.countSubject.pipe(
  map((value) => value * 2),
);
```

Представа да бъде  
**Subject** и се  
превръща в обикновен  
**observable**.

TS app.component.ts

```
subscriptions: Subscription[] = [];

countSubject = new BehaviorSubject<number>(1);
doubleCount$ = this.countSubject.pipe(
  map(value) => value * 2,
);

getObserverValue() {
  const sub = this.doubleCount$.subscribe(value => console.log(value));
  this.subscriptions.push(sub);
}

ngOnDestroy(): void {
  this.subscriptions.forEach(sub => sub.unsubscribe());
}
```

TS app.component.ts

```
count = signal<number>(1);
doubleCount = computed(() => {
  return this.count() * 2;
});

getSignalValue() {
  return this.doubleCount();
}
```

Много повече неща,  
за които трябва да  
се погрижим.



www.diadraw.com

TS app.component.ts

```
countOne = signal<number>(1);
countTwo = signal<number>(9);

derivedCount = computed(() => {
  return this.countOne() + this.countTwo();
});

changeValue() {
  this.countOne.set(2);
  this.countTwo.set(18);
}
```

Винаги ще има  
начална стойност.

Новата стойност ще  
се изчисли, след  
като сигналите се  
стабилизират.

TS app.component.ts

```
countOne = new BehaviorSubject<number>(1);
countTwo = new BehaviorSubject<number>(9);

derivedValue = combineLatest([this.countOne, this.countTwo]).pipe(
  map(([countOne, countTwo]) => countOne + countTwo)
);

changeValue(){
  this.countOne.next(2);
  this.countTwo.next(18);
}
```

Може да няма  
първоначална  
стойност

Поради начина, по  
който работи  
`combineLatest`,  
промяната на няколко  
стойности може да  
доведе до  
изчисляване на  
междинни стойности,  
които са некоректни

TS app.component.ts

```
count = signal<number>(1);
countEffect = effect(() => {
  console.log('The count now is ' + this.count());
});
```

Ще се изпълнява при  
всеки абонамент.

TS app.component.ts

```
count = new BehaviorSubject<number>(1);
count$ = this.count
  .asObservable()
  .pipe(tap((count) => console.log('The count now is ' + count)));
```

TS app.component.ts

```
template:
  {{ count$ | async }}
  {{ count$ | async }}
  {{ count$ | async }}  
,
```

# **Signals + RxJS**

# Създаване на Signal от Observable

Angular предоставя възможност за създаване на сигнал, който следи стойността на RxJS Observable, чрез функцията **toSignal**.

Подобно на AsyncPipe, toSignal автоматично се абонира за observable. Също така, когато компонентът бъде унищожен, абонаментът се премахва автоматично.

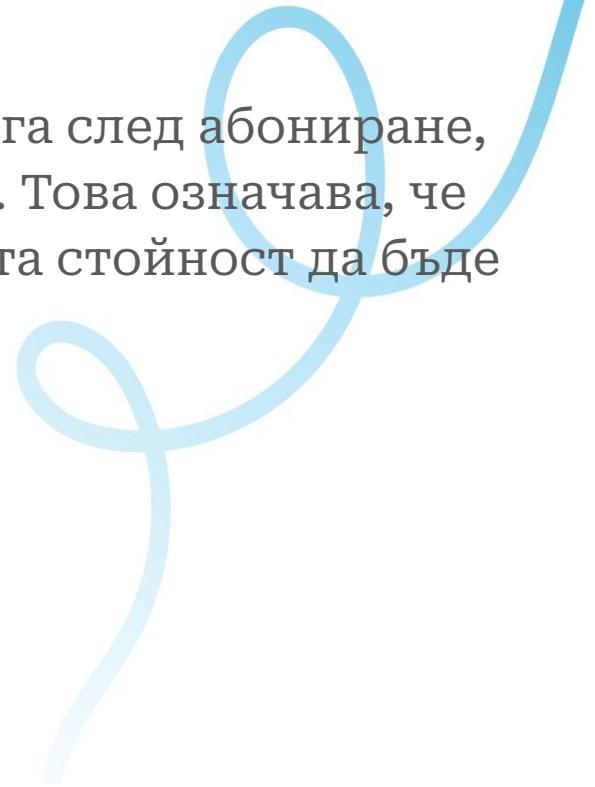
Тъй като toSignal създава нов абонамент, тряба да ограничим броя на пътите, в които я използваме за един и същи observable.

TS app.component.ts

```
export class AppComponent {  
    title = '06-signals';  
  
    id$ = of(1);  
  
    id = toSignal(this.id$);  
}
```

# Важно!

Observables могат да не върнат стойност веднага след аборниране, но сигналите винаги връщат стойност веднага. Това означава, че трябва да се справим с възможността началната стойност да бъде **undefined**.



# initialValue

TS app.component.ts

```
export class AppComponent {
  title = '06-signals';

  id$ = of(1);

  id = toSignal(this.id$, { initialValue: 0 });
}
```

# requireSync

TS app.component.ts

```
export class AppComponent {
  title = '06-signals';

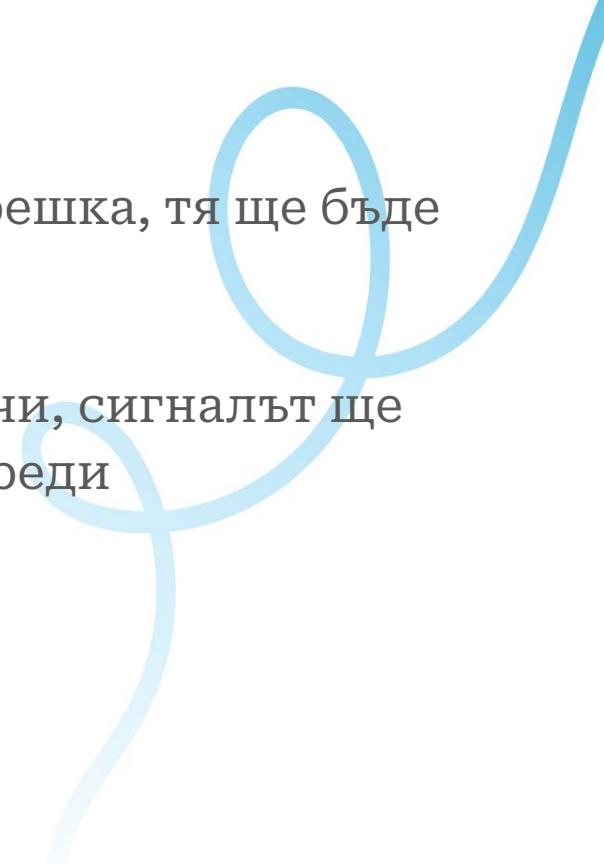
  idSubject = new BehaviorSubject(1);
  is$ = this.idSubject.asObservable();

  id = toSignal(this.is$, { requireSync: true });
}
```

# Важно!

Ако **Observable**, използван в **toSignal**, върне грешка, тя ще бъде хвърлена, когато сигналът бъде прочетен.

Ако **Observable**, използван в **toSignal**, приключи, сигналът ще продължи да използва последната стойност преди приключването.



TS app.component.ts

```
export class AppComponent {
  title = '06-signals';

  arr$ = of([1, 2, 3]);
  double = computed(() => {
    try {
      return toSignal<number[]>(this.arr$, { requireSync: true })().map(
        (v) => v * 2
      );
    } catch (error) {
      return [0];
    }
  });
}
```

# Създаване на Observable от Signal

Angular предоставя възможност за създаване на Observable, който следи стойността на сигнал чрез функцията **toObservable**.

Всеки път, когато сигналът промени стойността си, тя се добавя към потока от данни.

За да следи промяната на сигнала **toObservable** използва **effect**.

TS app.component.ts

```
export class AppComponent {
  title = '06-signals';

  constructor(private http: HttpClient) {}

  id = signal<number>(1);
  id$ = toObservable(this.id);

  search$ = this.id$.pipe(switchMap((id) => this.http.get(`/user/${id}`)));
}
```

# Важно!

**toObservable** използва **effect**, за да следи промяната на сигналите и съхранява стойностите в **ReplaySubject**.

За разлика от **observables**, ако променим стойността на сигнала няколко пъти последователно, това няма да доведе до няколко нови стойности в потока. Сигналът ще добави стойност в потока само когато се стабилизира.

TS app.component.ts

```
export class AppComponent {
  title = '06-signals';

  id = signal<number>(1);
  id$ = toObservable(this.id).subscribe((value) => console.log(value));

  changeId() {
    this.id.set(6);
    this.id.set(1);
    this.id.set(9);
  }
}
```

Console

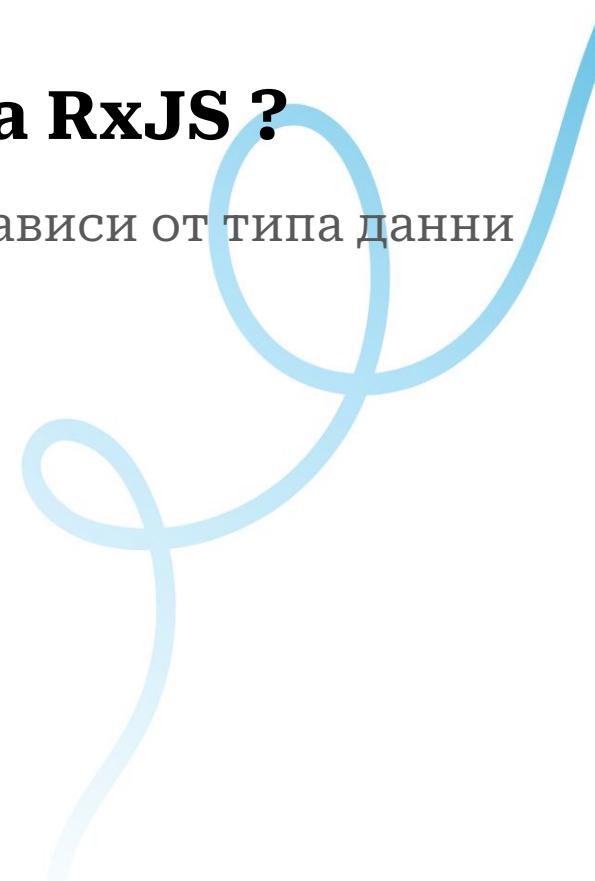
```
9
```



[www.diadraw.com](http://www.diadraw.com)

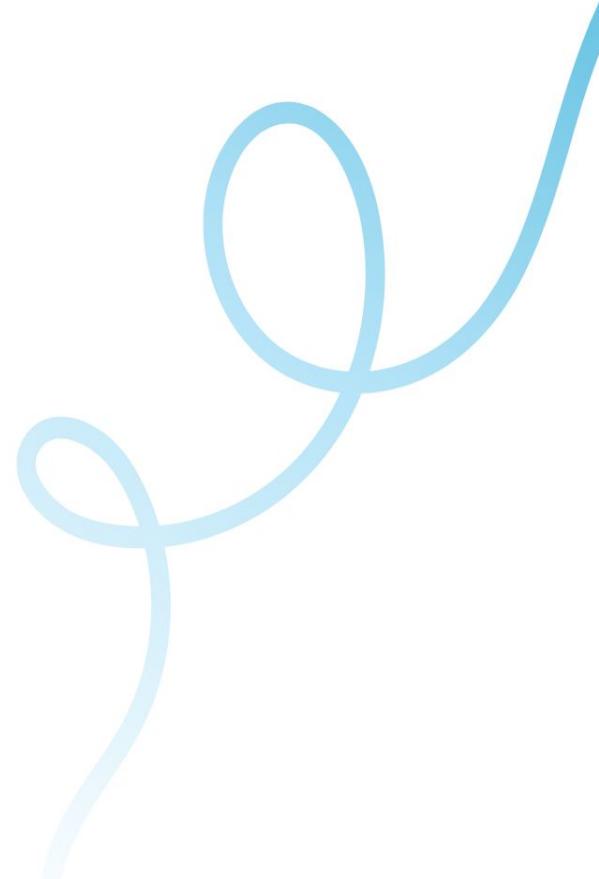
# Кога да използваме Signal и кога RxJS ?

Изборът между Signal и Observable в Angular зависи от типа данни и реактивност, които искате да постигнете



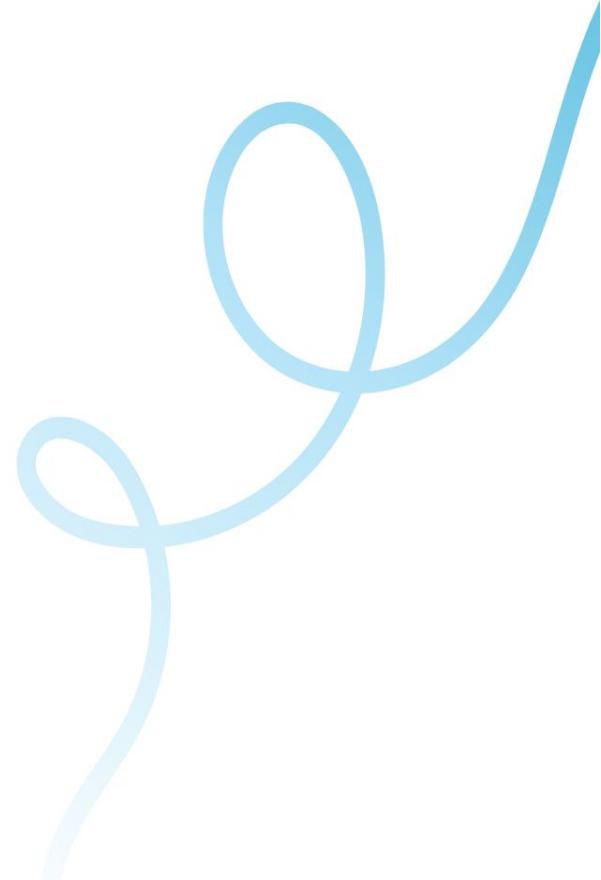
# Кога да използваме Signal

- UI state
- Синхронно изчисление на стойности
- Improved change detection

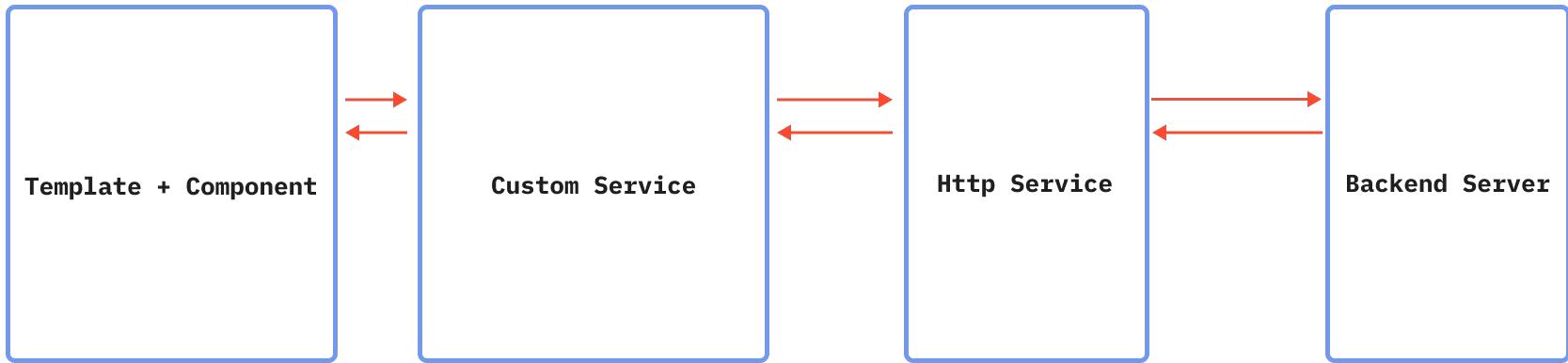


# Кога да използваме RxJS

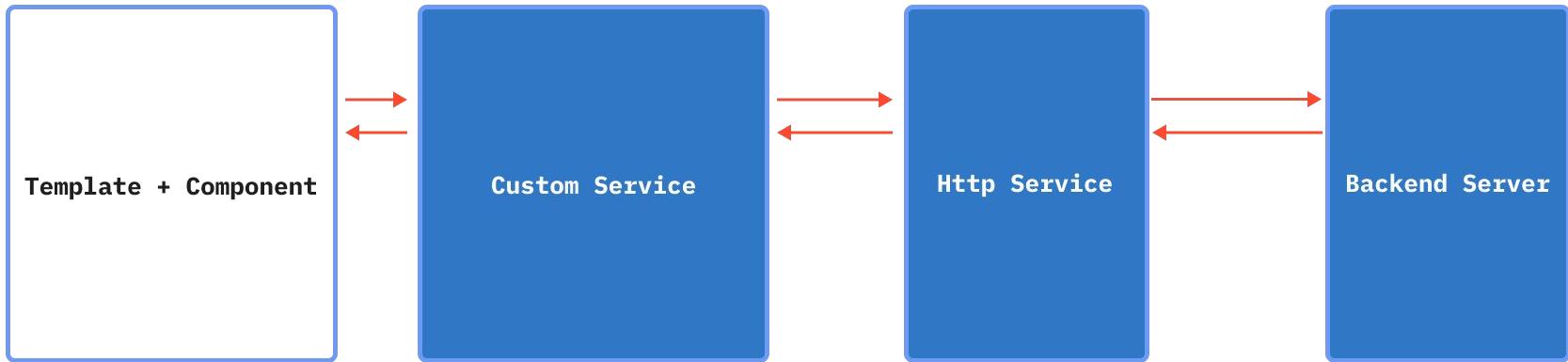
- Асинхронни операции
- HTTP requests
- Контрол върху жизнения цикъл на данни



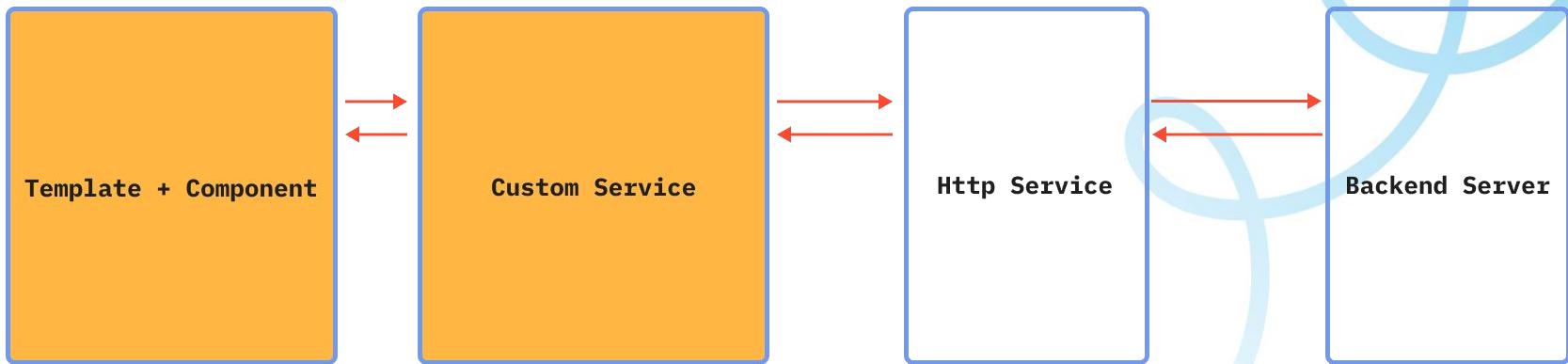
# Основна архітектура



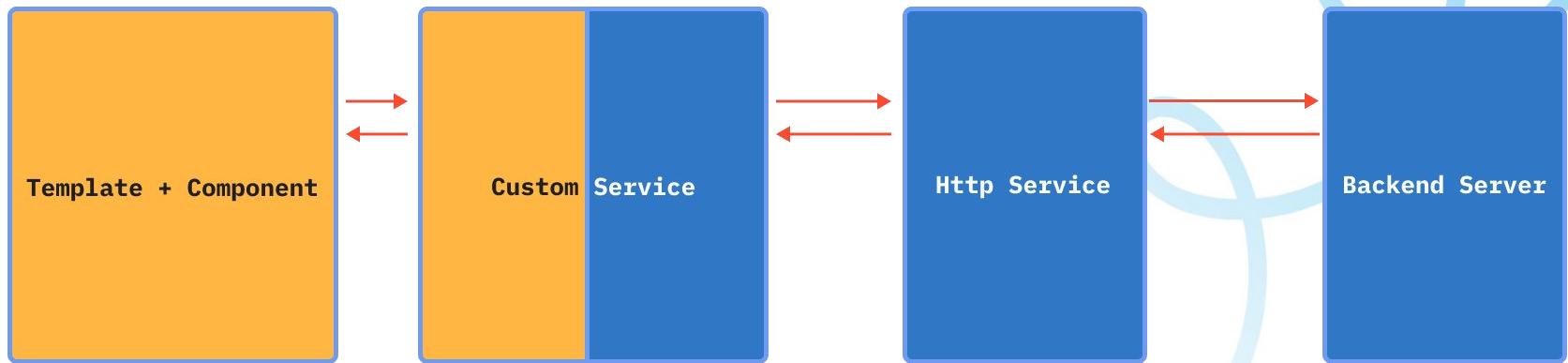
# Къде се ползва RxJS



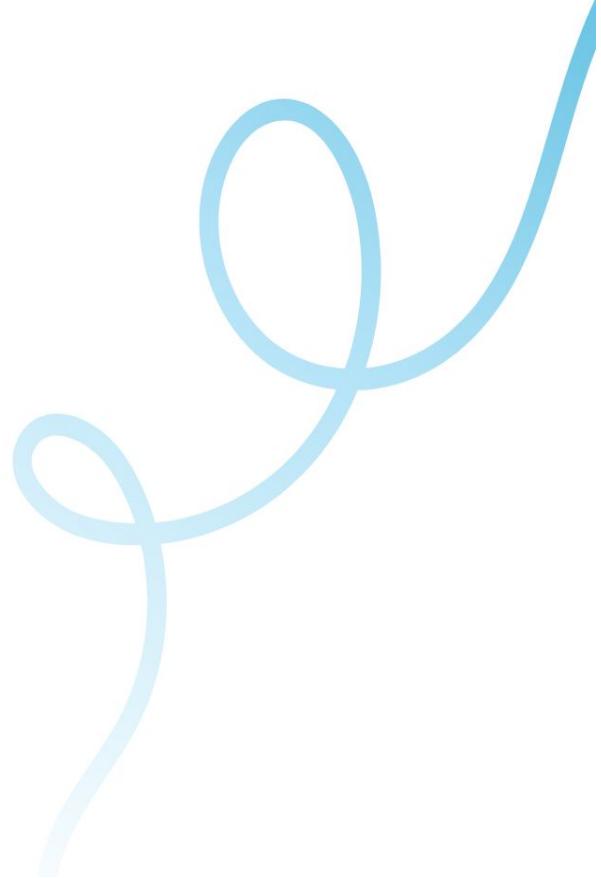
# Къде се ползват Signals



# Как взаимодействуют помежду си



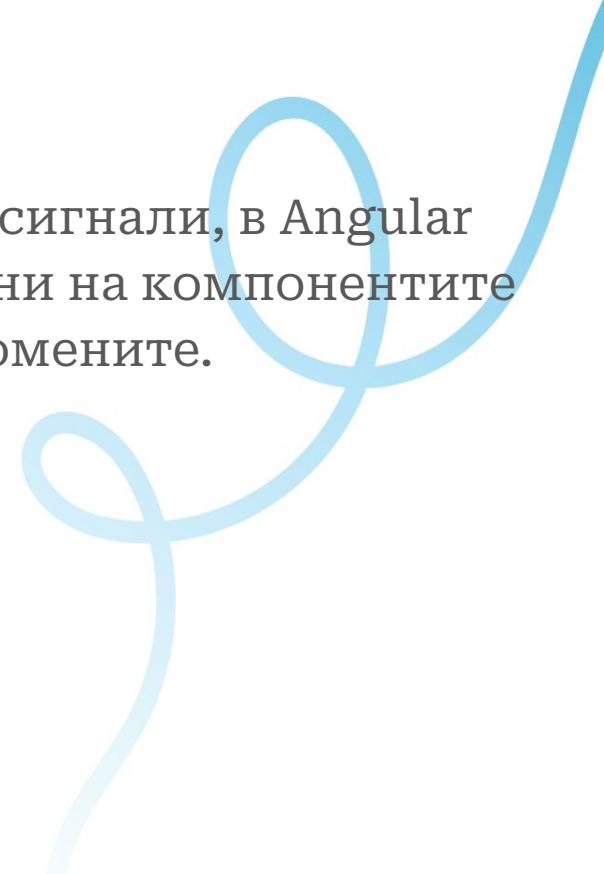
# Упражнение



# Inputs Signals

# Inputs като Signals

Освен стандартните начини за дефиниране на сигнали, в Angular вече е възможно да дефинираме входните данни на компонентите като сигнали, което улеснява следенето на промените.



# Как досега следим за промени в входните данни?

```
TS products.component.ts

@Component({
  selector: 'app-product',
  standalone: true,
  imports: [],
  templateUrl: './product.component.html',
})
export class ProductComponent {
  productData: any;

  @Input({ required: true }) set id(c: number) {
    // get data for product
    this.productData = { id: c };
  }
}
```

# Как досега следим за промени в входните данни?

```
TS products.component.ts

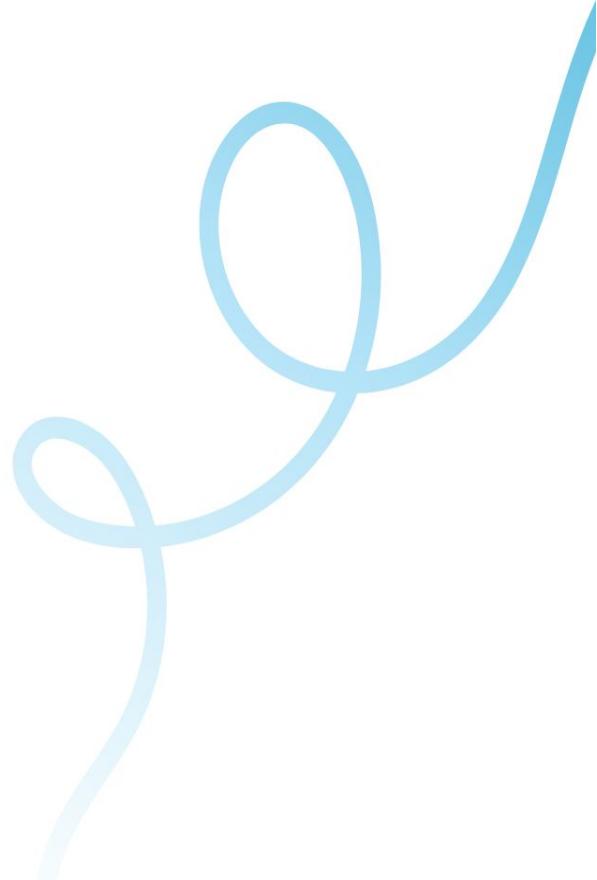
@Component({
  selector: 'app-product',
  standalone: true,
  imports: [],
  templateUrl: './product.component.html',
})
export class ProductComponent implements OnChanges {
  productData: any;

  @Input({ required: true }) id!: number;

  ngOnChanges(changes: SimpleChanges): void {
    // get data for product
    this.productData = { id: this.id };
  }
}
```

# Типове входните данни

- Signal input
- Model input



# Signal input

Позволяват стойностите на входните параметри да се следят реактивно и автоматично да се обновяват при промяна в родителския компонент.

Създават се чрез **input** функцията.



TS products.component.ts

```
@Component({
  selector: 'app-product',
  standalone: true,
  imports: [],
  templateUrl: './product.component.html',
})
export class ProductComponent {
  id = input<number>(0);

  productData = computed(() => {
    // get data for product
    return { id: this.id() };
  });
}
```

Дефиниране на  
входен параметър  
като Signal

id = input<number>(0);

productData = computed(() => {  
 // get data for product  
 return { id: this.id() };  
});

Следене на  
промени във  
входните данни

# Входните данни могат да бъдат:

- Опционални – ако входните данни са опционални, може да зададем начална стойност, която да се използва, или Angular ще зададе **undefined** по подразбиране.
- Задължителни – за да направим входните данни задължителни, трябва да използваме **input.required**. В този случай не е необходимо да декларираме начална стойност.

ts products.component.ts

```
@Component({
  selector: 'app-product',
  standalone: true,
  imports: [],
  templateUrl: './product.component.html',
})
export class ProductComponent {
  id = input<number>(0);
  productData = computed(() => {
    // get data for product
    return { id: this.id() };
  });
}
```

Опционални  
входни  
данни

ts products.component.ts

```
@Component({
  selector: 'app-product',
  standalone: true,
  imports: [],
  templateUrl: './product.component.html',
})
export class ProductComponent {
  id = input.required<number>();
  productData = computed(() => {
    // get data for product
    return { id: this.id() };
  });
}
```

Задължителни  
входни  
данни

Следене на  
промени във  
входните данни

```
TS products.component.ts

@Component({
  selector: 'app-product',
  standalone: true,
  imports: [],
  templateUrl: './product.component.html',
})
export class ProductComponent {
  id = input.required<number>();

  constructor() {
    effect(() => {
      // do some action
      console.log(this.id());
    });
  }
}
```

# Трансформация на входни данни

При използването на `input signals` в Angular, може да се приложи трансформация на входната стойност преди тя да бъде съхранена в сигнала.

Това може да се постигне чрез задаване на опцията `transform` при създаване на сигнала, като стойността ѝ трябва да бъде `pure` функция.

TS products.component.ts

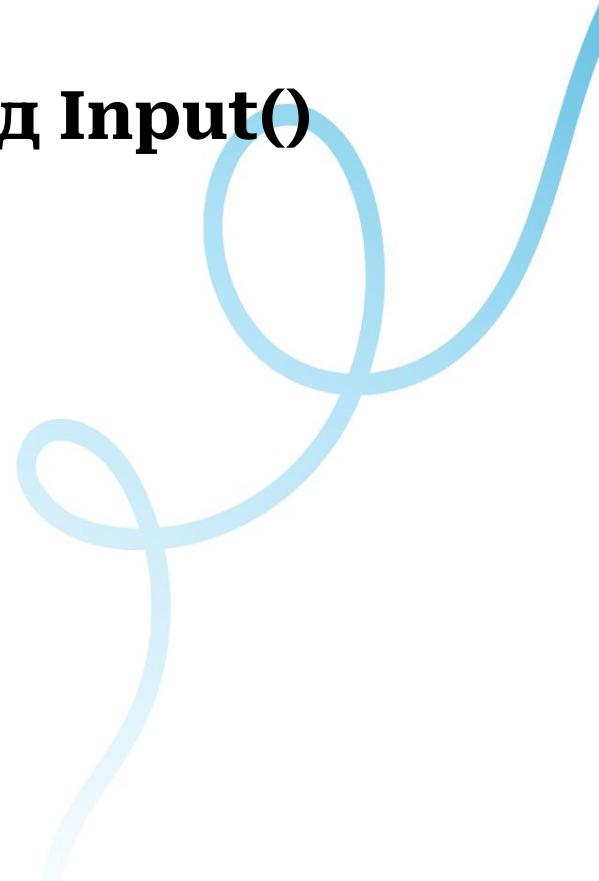
```
@Component({
  selector: 'app-product',
  standalone: true,
  imports: [],
  templateUrl: './product.component.html',
})
export class ProductComponent {
  disabled = input(false, {
    transform: (value: boolean | string) =>
      typeof value === 'string' ? value === '' : value,
  });
}
```

# Важно!

- Избягвайте промени в смисъла на входа – Преобразуванията не трябва да променят значението на данните, за да не създават объркване.
- Алтернативи за комплексни трансформации – За по-сложни или изчислени стойности, използвайте computed сигнали или effect, вместо да разчитате на transform.

# Предимства на signal inputs пред Input()

- По-добра типизация
- Подобрена реактивност
- Няма нужда да работим с ngOnChanges

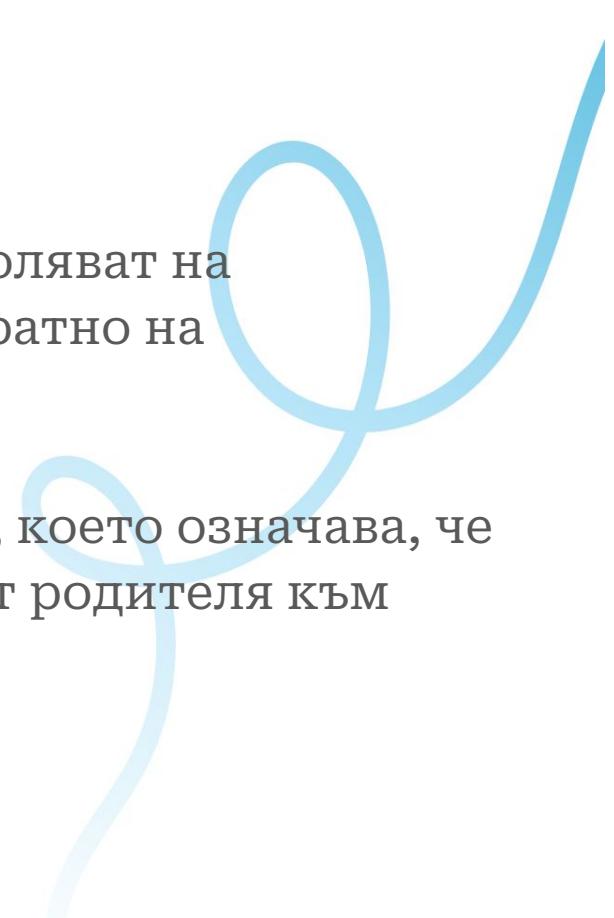


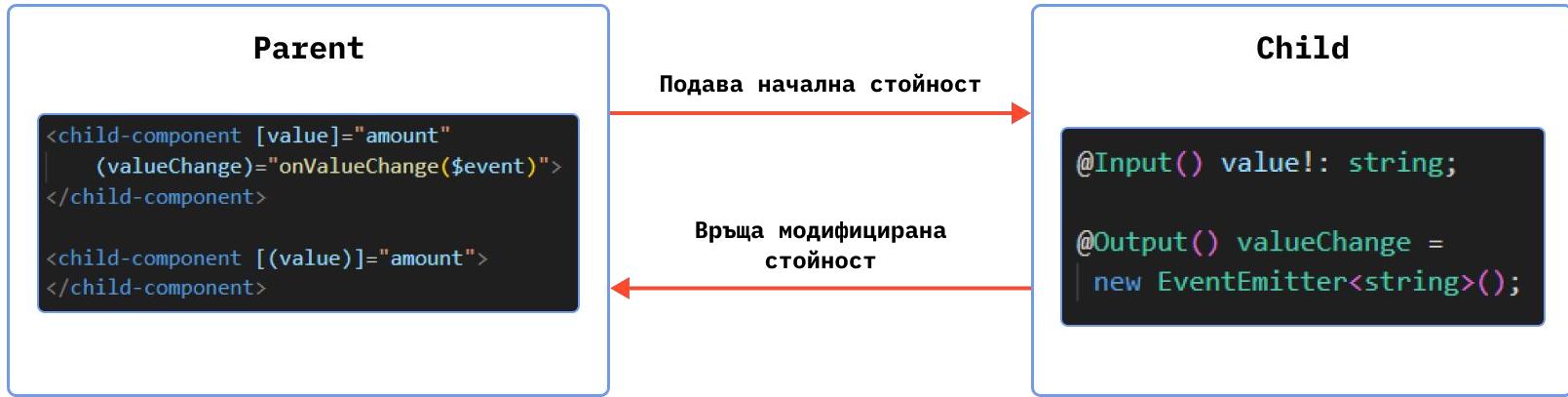
# Model input

Специален тип входни параметри, които позволяват на компонентите да предават нови стойности обратно на родителските компоненти.

Те осигуряват двупосочко свързване на данни, което означава, че стойностите могат да текат в двете посоки – от родителя към децата и обратно(**Two-way binding**)

Създават се чрез **model** функцията.





## Parent

```
<child-component [value]="amount"  
|   (valueChange)="onValueChange($event)">  
</child-component>  
  
<child-component [(value)]="amount">  
</child-component>
```

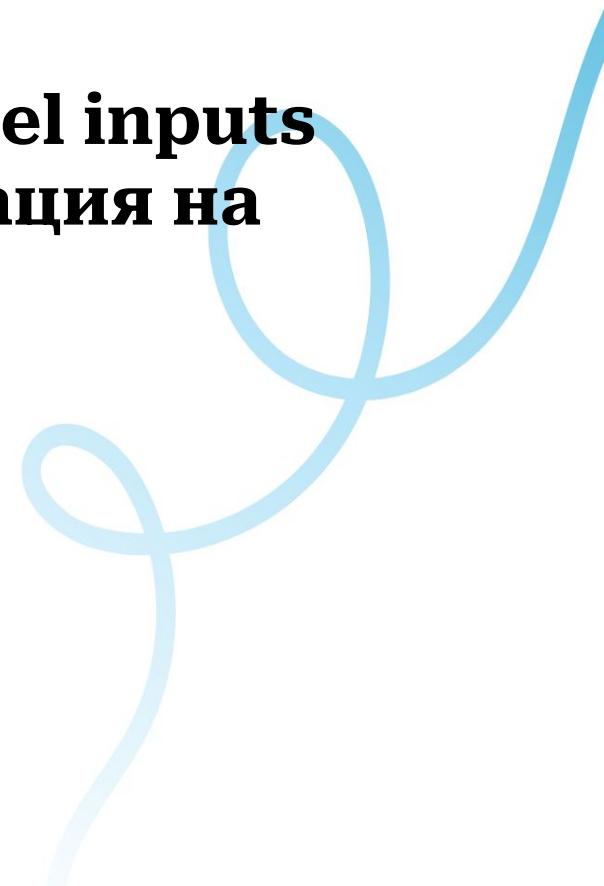
Подава начална стойност

Връща модифицирана  
стойност

## Child

```
value = model('');
```

**За разлика от signal inputs, model inputs позволяват директна модификация на стойността в тях.**



TS products.component.ts

```
@Component({
  selector: 'app-product',
  standalone: true,
  imports: [],
  templateUrl: './product.component.html',
})
export class ProductComponent {
  value = model('');

  onSomeAction() {
    this.value.set('new value');
  }
}
```

Позволява  
директна промяна  
на стойността



# Two-way binding with model input

TS products.component.ts

```
@Component({
  selector: 'app-product',
  standalone: true,
  imports: [],
  template:
    <app-product-description
      [(description)]="description"
    ></app-product-description>
  ,
})
export class ProductComponent {
  description = signal('');
}
```

Two-way binding  
with signal

TS products.component.ts

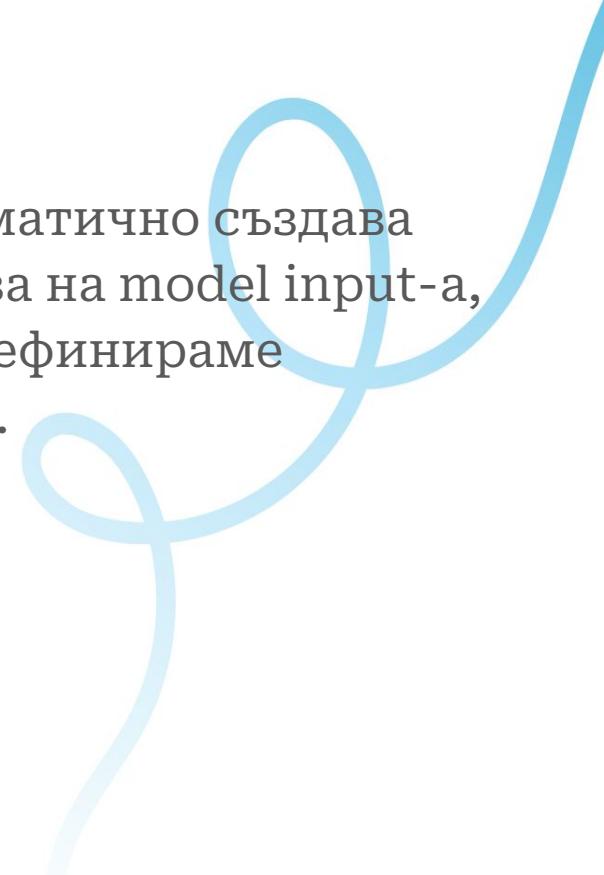
```
@Component({
  selector: 'app-product',
  standalone: true,
  imports: [],
  template:
    <app-product-description
      [(description)]="description"
    ></app-product-description>
  ,
})
export class ProductComponent {
  description = '';
}
```

Two-way binding  
with plain  
properties

Използваме  
сигнала, а не  
стойността му, за  
Two-way binding

# Важно!

Когато дефинираме **model input**, Angular автоматично създава output за него. Името на output-а съвпада с това на model input-а, но с наставка “Change”. Това ни позволява да дефинираме странични ефекти при промяна на стойността.



### TS products.component.ts

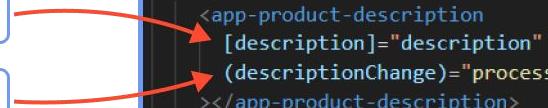
```
@Component({
  selector: 'app-product',
  standalone: true,
  imports: [],
  template: `
    <app-product-description
      [description]="description"
      (descriptionChange)="processEvent($event)"
    ></app-product-description>
  `,
})
export class ProductComponent {
  description = '';

  processEvent(newDescription: string) {
    this.description = newDescription;

    // do some effect
  }
}
```

**Input value**

**Output event**



# По-ефективен начин за извършване на странични ефекти

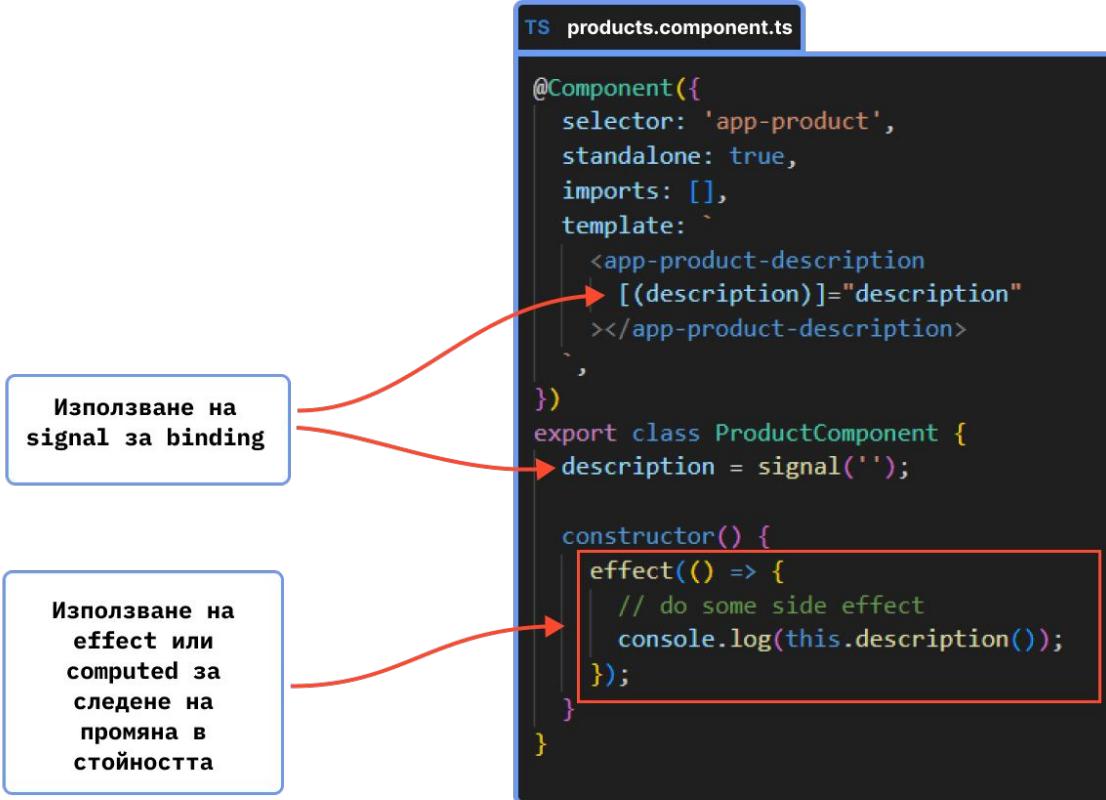
TS products.component.ts

```
@Component({
  selector: 'app-product',
  standalone: true,
  imports: [],
  template:
    <app-product-description
      [(description)]="description"
    ></app-product-description>
  ,
})
export class ProductComponent {
  description = signal('');

  constructor() {
    effect(() => {
      // do some side effect
      console.log(this.description());
    });
  }
}
```

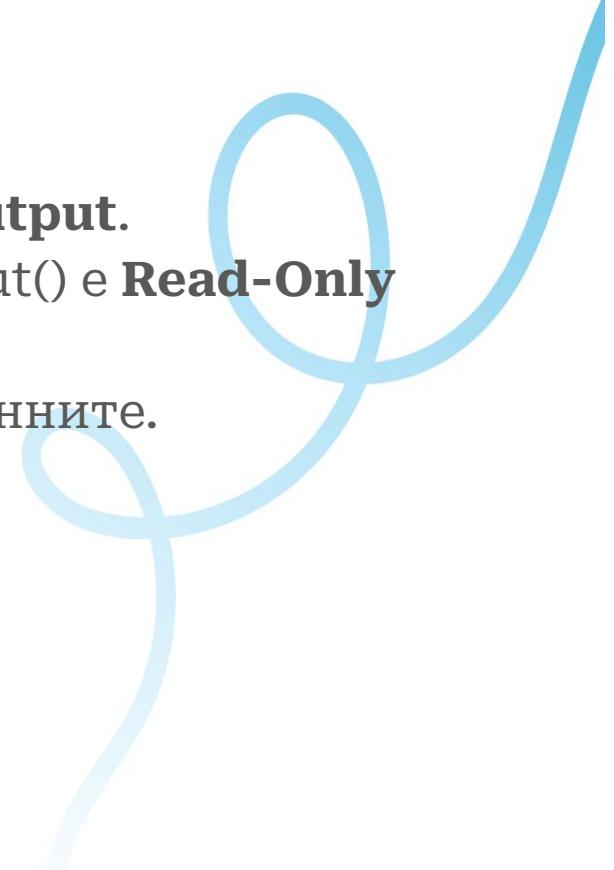
Използване на signal за binding

Използване на effect или computed за следене на промяна в стойността

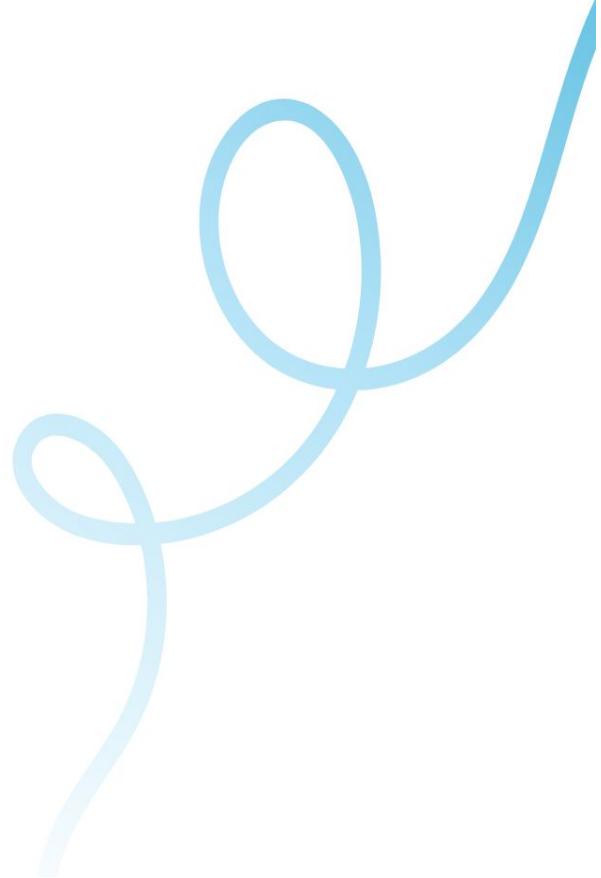


# model() vs input()

- model() дефинира едновременно **input** и **output**.
- model() сигналите са **Writable Signal**, а input() е **Read-Only Signal**
- model() не поддържа трансформация на данните.



# Упражнение



# Благодаря за вниманието!