Signals

SD 555 – Web Application Development III

Maharishi International University

Department of Computer Science

Associate Professor Asaad Saad

Maharishi International University - Fairfield, Iowa

signal

A signal is a callable object (can be invoked like a function), and when called will return its latest value. This object has several methods to update the value.

There are two essential types of signals:

- **`WritableSignal`** is a type whose value can be changed.
- **`Signal`** is a type whose values are immutable. (**readonly**)

Both **`Signal`** and **`WritableSignal`** accept a generic type, and their type argument indicates the value that is stored inside the signal.

Signal Features

- Signals must have a **default value**
- Signal operations are **synchronous**
- We can perform **side effects** based on changes in the value
- We can create new reactive values from existing ones (**computed signals**)
- No subscription is needed, and un-subscription will be automatic

Updating Signals

There are 2 ways of changing a signal: setting a value directly, and updating the value using the previous value with a callback function, the callback returns the new value.

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

$count.set(1);
console.log($count()); // 1

$count.update(prev_value => prev_value + 1);
console.log($count()); // 2
```

Computed Signals

Computed signals re-run their computation only when it is read for the first time, and then it will only execute when the signal it depends on changes. Reading the computed value will not trigger recalculation.

```
const a = signal(5);
const b = signal(9);

const sum = computed(() => {
  console.log('Recalculating');
  return a() + b();
});

sum();
sum();
sum();
```

When to Create a Computed Signal

Computed signals are readonly, create them when you need a filtered or mapped set of the signal data.

```
export class MyComponent {

  tickets = signal<Tickets[]>([]);
  resolvedTickets = computed(() => this.tickets().filter((r) => r.status !== 'Pending'));

}
```

Now we have a signal that only contains resolved requests (ones that are no longer "Pending").

```
<h3>Resolved {{ resolvedTickets().length }} / {{ tickets().length }} Unresolved </h3>
```

Effects

Signal effects are always **asynchronous**, no matter if the code inside the callback does not deal with any async logic. This means that if during one function execution the value of a signal that an effect watches changes several times, the effect's logic is performed once at the end.

- You cannot set or update a signal inside an effect since that could potentially result in infinite circular updates.
- `effect()` can only be used within an injection context.

Effect Example

As opposed to **`computed`** signals, **`effects`** re-run are closely tied to the change detection mechanism. This means that the following code will:

- print **`0`** when it starts.
- print **`1110`** when the **`update()`** method is called.

```
export class SomeComponent {
  $counter = signal(0);

  constructor() {
    effect(() => {
      console.log(`Value is ${this.$counter()}`);
    });
  }

  update() {
    this.$counter.set(10);
    this.$counter.update(count => count + 100);
    this.$counter.update(count => count + 1000);
  }
}
```

When to Use Effects

- Writing to external storage
- Calling third-party APIs
- Performing UI updates that cannot be expressed via Angular’s template syntax (page title)

Effects can be Destroyed Earlier

```
export class SomeComponent {

  $count = signal(0);
  log_effect = effect(() => {
    console.log(`Count is ${this.$count()}`);
  });

  constructor() {
    setInterval(() => this.$count.update(c => c + 1), 1_000);
    setTimeout(()=> this.log_effect.destroy(), 10_000);
  }

}
```

Signal Best Practices

- Always use computed signals for variations of state.
- Do not write to signals in a computed or effect callbacks.
- Only use signals in the computed callback. Using non-signal properties in a computed callback is a sign of problems with the component structure.

Template Local Variables (Refs)

```
@Component({
  selector: 'app-root',
  template: `
    <div #header>Hello</div>
  `,
})
export class AppComponent {}
```

We created a template variable **header** which is a direct reference to the **div** DOM element.

Access Template Local Variables

```
@Component({
  selector: 'app-root',
  template: `
    <div #header>Hello</div>
  `,
})
export class AppComponent {
    $header = viewChild.required<ElementRef<HTMLDivElement>>('header');
}
```

ngAfterView** Lifecycle Hook

You can read the template variable value starting with **ngAfterView**** lifecycle hook.

```
@Component({
  selector: 'app-root',
  template: `
    <div #header>Hello</div>
  `,
})
export class AppComponent {
  $header = viewChild.required<ElementRef<HTMLDivElement>>('header');

  ngAfterViewChecked(){ console.log(this.$header().nativeElement.innerHTML ) }

}
```

Custom Pipes

To create a custom pipe from CLI: **`ng g p myPipe`**

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({ name: 'double' , standalone: true })
export class DoublePipe implements PipeTransform {
   transform(value: unknown, ...args?: unknown[]): unknown { return value * 2; }
}
```

Creating our Custom Directive

To create a new custom Directive class from Angular CLI we use:

```
ng g d directiveName
```

Notice

- Angular creates a Directive using the **@Directive()** decorator
- Directives don't have template or styles metadata
- Directives bind to all host element attributes (inputs)
- Assign the selector as a **CSS selector [attribute]**

Host Element Binding

Reference to the **Host Element** we are applying the directive on

```
#element = inject(ElementRef) // provide access to the host element
#renderer2 = inject(Renderer2) // utility API to perform DOM changes

ngOnInit(){
  #element.nativeElement.style.fontSize = '22px'; // DOM Dependent
  #renderer2.setStyle(#element.nativeElement, 'font-size', '22px'); // DOM Independent
}
```

```
@HostListener('click') handleClick(){
        this.#renderer2.setStyle(this.#element.nativeElement, 'color', 'red')
}
```

Listening to Host Events