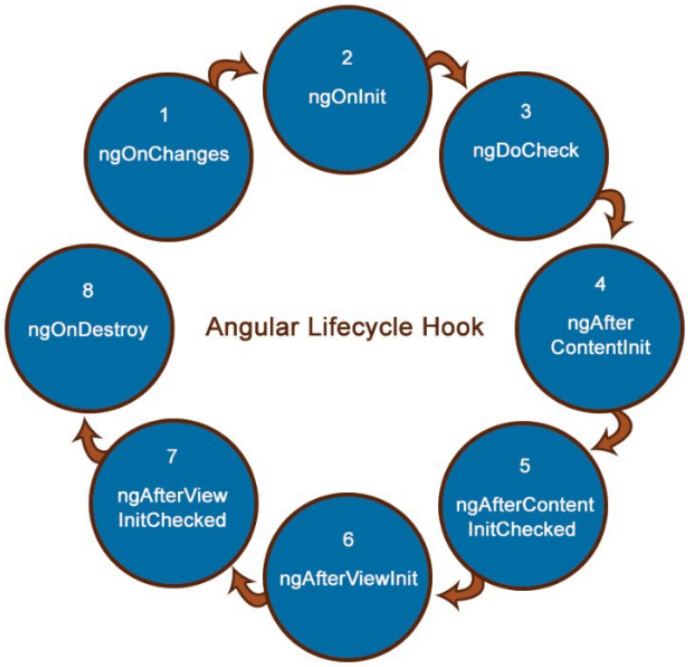
Mudit Manucha
Mar 31 · 5 min read · Listen

Actual Use Of Angular Lifecycle Hooks

We Will be Learning:

- Different lifecycle hooks for Angular components
- What triggers lifecycle hooks and the order in which they are called
- How to hook into component lifecycles and when to use it

These are the hooks for components or directives, in call order:



1. **ngOnChanges** : Called before ngOninit() (if the component has bound inputs) and whenever one or more data-bound input properties change
2. **ngOninit** : Called only once at time of components initialization
3. **ngDoCheck** : Called immediately after ngOnchanges() on every change detection run, and immediately after ngOninit() on the first run
4. **ngAfterContentInit*** : Called only once after ngDoCheck()
5. **ngAfterContentChecked*** : Called after ngAfterContentInit() and every subsequent ngDocheck()
6. **ngAfterViewInit*** : Called only once after ngAfterContentChecked()
7. **ngAfterViewChecked*** : Called after the ngAfterViewInit() and every subsequent ngAfterContentChecked()
8. **ngOnDestroy** : Called only once before the component is destroyed

Note: If component has no data-bound properties (Inputs()), the framework will not call ngOnChanges.

'' : These hooks are for handling events due to components child components*

OnInit

This lifecycle hook is triggered after angular has initialized all data-bound property.

OnInit is called only once. It is best practice to initialize functional logic in this block.

```
@Component({
 //...
})
export class AComponent implements OnInit {
 ngOnInit() {
 console.log("ngOnInit called")
 }
}
```

OnChanges

OnChanges is a lifecycle hook that is called when any data-bound property of a directive changes.

It should be used in child component when you have data-bound properties (using Input()) and child component require to perform some functionality when data-bound property is changed in its parent component.

```
@Component({
 // ...
 template: `
 <bcomp [games]="games"></bcomp>

<button (click)="updateGames()">Click to update</button> `
})
export class App {
 games = ['chess', 'cricket'];

 updateGames(){
  this.games = ['hockey', 'chess']
 }
}

@Component({
 selector: 'bcomp',
 template: `
 <div *ngFor="let game of games">
 {{game}}
 </div>
 `
})
export class BComponent implements OnChanges {
 @Input() games: string[];
 ngOnChanges() {
 console.log("The game property changed")
 }
}
```

In above example ngOnchanges will be triggered of Bcomponent if “Click to update” button is triggered in app component.

Note: ngOnchanges will only trigger if reference of data-bound value is changed and not in case if just its property is changed.

DoCheck

Docheck is used to add custom code to perform a custom change detection.

With reference to above example :

```
@Component({
 // ...
 template: `
 <bcomp [games]="games"></bcomp>

<button (click)="updateGames()">Click to update</button> `
})
export class App {
 games = ['chess','cricket'];

updateGames(){
  //updating only property and not changing reference
  this.games[0] ='hockey';
 }
}

@Component({
 selector: 'bcomp',
 template: `
 <div *ngFor="let game of games">
 {{game}}
 </div>
 `
})
export class BComponent implements OnChanges {
 @Input() games: string[];
 ngOnChanges() {
 console.log("The game property changed")
 }
 ngDocheck(){
  console.log("Docheck called, games might have got updated")
 }
}
```

In above case ngOnchanges will not trigger when button is clicked as reference of data-bound property is not changed.

In such situations ngDocheck can be used as it gets triggered for any CD/event triggered on component.

Note: Be careful not to write complex code in ngDocheck as it will get triggered for any CD/event changes on component and keep a proper if conditions to make sure that when the code should run inside its block.

AfterContentInit

AftercontentInit is called when the content that is projected between tags: `<ng-content></ng-content>` of a component has been initialized.

```
@Component({
 selector: 'bcomp',
 template: `
 <div>This is a BComponnet</div>
 `
})
export class BComponent {}
@Component({
 selector: 'acomp',
 template: `
 <ng-content></ng-content>
 `
})
export class AComponent implements AfterContentInit {
 ngAfterContentInit() {
 // ...
 }
}
@Component({
 template: `
 <acomp>
 <bcomp></bcomp>
 </acomp>
 `
```

```
})
export class App {}
```

The AComponent will project in between its tag `<acomp></acomp>` inside the `ng-content` tag. Now, in the App component, the BComponent is projected in the `acomp` tag. When the BComponent is being initialized, `ngAfterContentInit` hook will be called in AComponent.

AfterContentChecked

This hook is called into a component via the `ng-content` tag has completed its check:

```
@Component({
 selector: 'acomp',
 template: `
 <ng-content></ng-content>
 `
})
export class AComponent implements AfterContentInit {
 ngAfterContentInit() {
 // ...
 }
}
@Component({
 template: `
 <acomp>
 {{data}}
 </acomp>
 `
})
export class App implements AfterContentChecked {
 data: any
 ngAfterCOntentChecked() {
 //...
 }
}
```

The `App` component has a `data` property that is inside the `AComponent`. When the `data` property changes, the `ngAfterContentChecked` method will be called.

AfterViewInit

This hook is called after a component's view and its children's views have been created and fully initialized.

This hook comes in handy when we want to reference a component/directive instance in our component using `ViewChild` / `ViewChildren`.

Using it in any other method might cause an error because the component/directive might not have been initialized at the time of use:

```
@Component({
 selector: 'acomp',
 // ...
})
export class AComponent {
 aCompMethod() {
 }
}
@Component({
 template: `
 <acomp #acomp></acomp>
 `
})
export class App implements AfterViewInit {
 @ViewChild('acomp') acomp: AComponent
 ngAfterViewInit() {
 //...
 this.acomp.aCompMethod();
```

```
  }
}
```

If we try to get *'acomp'* reference before ngAfterViewInit than it might throw a reference error because AComponent has not been initialized in that time.

AfterViewChecked

This hook is called after the change detector of a components child component has been run for checks.

```
@Component({
 selector: 'acomp',
 // ...
})
export class AComponent {
 @Input() data
 aCompMethod() {
 return this.data * 9
 }
}
@Component({
 template: `
 <div>
 {{aCompData}}
 <acomp [data]="data" #acomp></acomp>
 <button (click)="changeData()"></button>
 </div>
 `
})
export class App implements AfterViewChecked {
 @ViewChild('acomp') acomp: AComponent
 data: any
 acompData:any
 changeData() {
 this.data = Date.now()
 }
 ngAfterViewChecked() {
 //...
 this.acompData = this.acomp.aCompMethod()
 }
}
```

`AComponent` has an input binding `data`, and we change the data binding when we click the `changeData` button, as we want to display the result of `AComponent`'s `aCompMethod` result in the `aCompData`.

If we call the method in any other place, we might hit a reference error because Angular might not be done with CD run on the `AComponent`. So we implemented the `AfterViewChecked` interface and added `ngAfterViewInit` method because the method will be run when the `AComponent` has completed its CD run.

In this case, we won't get an incorrect data value when we call the `aCompMethod`, because the data will have the current value.

Note: Be careful not to set any variables bound to the template here. If you do, you'll receive the `"Expression has changed after it was checked"` error.

OnDestroy

This is lifecycle hook that is called when a directive, pipe, or service is destroyed.

Use this for any custom cleanup that needs to occur when the instance is destroyed.

This hook is mostly used to unsubscribe from our observable streams and detach

event handlers to avoid memory leaks.