Angular 16 : Online Class

Component Lifecycle Hook

By: Sahosoft Solutions

Presented by : Ajeet Kumar

lifecycle hook

The number of methods called in a sequence to execute a component at specific moments is known as lifecycle hook.

COMPONENT LIFECYCLE HOOKS OVERVIEW

In Angular, every component has a life-cycle, a number of different stages it goes through.

There are 8 different stages in the component lifecycle. Every stage is called as lifecycle hook event. So, we can use these hook events in different phases of our application ***to obtain control of the components***.

lifecycle hook

Basically, what Angular does with a component is that it creates the component, renders the component, creates and renders the component’s children, checks when the component's data-bound properties change and destroys the component before removing it from the DOM. Means

Lifecycle of a component includes:

- Creating a component
- Rendering a component
- Creating and rendering its child components
- Checking data-bound properties
- Destroying and removing it from DOM

lifecycle hook

Constructor

Since a component is a Typescript class, every component must have a constructor method. The constructor of the component class executes, first, before the execution of any other lifecycle hook events. If we need to inject any dependencies into the component, then the constructor is the best place to inject those dependencies**. After executing the constructor, Angular executes its lifecycle hook methods in a specific order.**

We can say that once a new component is an instantiation, Angular goes through a couple of different phases in this creation process and it will actually give us a chance to hook into these phases by implementing some methods as per our requirement.

lifecycle hook

Lifecycle Sequence

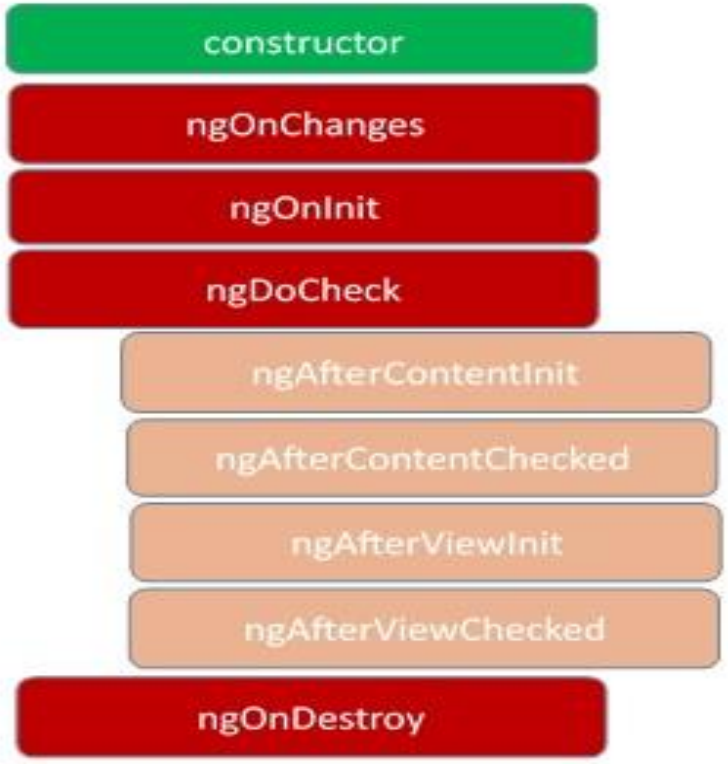
OnChange - OnInit - DoCheck - AfterContentInit - AfterContentChecked - AfterViewInit - AfterViewChecked - OnDestroy.

lifecycle hook

Now, try to learn all these hooks and learn how can we put our code between these phases.
We need to learn only 3 steps to use lifecycle hooks, they are:

- Import Hook interfaces from '@angular/core' library
- Declare that component/directive implements lifecycle hook interface
- Create the hook method and define the functionality of that method.

Let's start with these hooks:



lifecycle hook

These stages are mainly divided into two phases –

- one is linked to the component itself
- and another is linked to the children of that component.

lifecycle hook

Constructor

Whenever working with components, the first step is the calling of the constructor. This happens even before the implementation of any lifecycle hook. Also, if our component has any dependencies, a constructor is where we can inject those.

lifecycle hook

ngOnChange:

It is always called whenever one of our bound input changes; i.e., changes occur on those properties which are decorated with @Input() method.

Actually whenever we communicate between parent and child component, then the passing property from parent component is always received by child component with @Input decorator.

When the parent changes the Input properties, then this hook is invoked in the child component and we can easily find out the details about which input properties have changed and how they have changed.

ngOnChanges()

- Used in pretty much any component that has an input.
- Called whenever an input value changes
- Is called the first time before ngOnInit

lifecycle hook

ngOnInit():

ngOnInit() methods can be executed once component has been initialized, i.e. whenever component is created it triggers it.

This hook is fired after ngOnChanges() and before any of the child directive properties are initialized. There is a place where we will put logic related to initialization of properties.

ngOnInit()

- Used to initialize data in a component.
- Called after input values are set when a component is initialized.
- Added to every component by default by the Angular CLI.
- Called only once

lifecycle hook

ngDoCheck():

Whenever something changes on the template of a component or inside the component then ngDoCheck() executes. We can also say that it is called during every change detection run. This hook is called after ngOnInit().

Actually, this is very similar to ngOnChanges() hook, the major difference is that ngOnChanges() does not detect all the changes made to the input properties. It detects changes for those properties which are passed by value. However, ngDoCheck() detects changes for those properties also which are passed by reference such as arrays.

ngDoCheck()

- Called during all change detection runs
- A run through the view by Angular to update/detect changes

lifecycle hook

ngAfterContentInit():

This is called whenever the content which is projected through ng-content has been initialized.

Here, content projected means a way to import HTML content from outside the component and insert that content into the content's template in a designated spot and this spot is identified by Angular with the help of <ng-content>, i.e. it can put the content's template where <ng-content></ng-content> is present.

ngAfterContentInit()

- Called only once after first ngDoCheck()
- Called after the first run through of initializing content

@ContentChild

@ContentChild gives the first element or directive matching the selector from the content DOM. If new child element replaces the old one matching the selector in content DOM, then property will also be updated. @ContentChild has following metadata properties.

lifecycle hook

ngAfterContentChecked():

It is executed whenever the change detection occurs i.e. content will be projecting into our component. It is called immediately after the ngAfterContentInit and after every subsequent ngDoCheck().

ngAfterContentChecked()

- Called after every ngDoCheck()
- Waits till after ngAfterContentInit() on first run through

lifecycle hook

ngAfterViewInit():

Angular **ngAfterViewInit()** is the method of **AfterViewInit** interface.

ngAfterViewInit() is a lifecycle hook that is called after Angular has fully initialized a component's views. **ngAfterViewInit()** is used to handle any additional initialization tasks.

It is called after the component's view (and child view) has been initialized. It is very similar to **ngAfterContentInit**, but it works on view that is whenever view is initialize it fires.

ngAfterViewInit()

1. Called after Angular initializes component and child component content.
2. Called only once after view is initialized

ngAfterViewInit() is used to access properties annotated with **@ViewChild()** and **@ViewChildren()** decorators.

lifecycle hook

ngAfterViewInit():

@ViewChild()

@ViewChild() can be used for component communication. A component will get instance of another component inside it using @ViewChild(). In this way parent component will be able to access the properties and methods of child component. The child component selector will be used in parent component HTML template.

@Common Error when use ngAfterViewInit():

This error can be fixed two ways,

1.By changing the **ViewChild** property in **ngAfterContentInit** life cycle hook
2.Manually calling change detection using **ChangeDetectorRef**

To fix it in **ngAfterContentInit** life cycle hook you need to implement **AfterContentInit** interface.

```
ngAfterContentInit() {
    this.messageViewChild.message = 'Passed as View Child';
}
```

lifecycle hook

ngAfterViewInit():

Only problem with this approach is when you work with more than one ViewChild also known as ViewChildren. Reference of ViewChildren is not available in ngAfterContnetInit life cycle hook. In that case, to fix the above error, you will have to use a change detection mechanism. To use the change detection mechanism:

1.Import ChangeDetectorRef from @angular/core
2.Inject it to the constructor of Component class
3.Call detectChanges() method after ViewChild property is changed

You can use manual change detection like shown in below listing:

```
constructor(private cd: ChangeDetectorRef) {}

ngAfterViewInit() {
    console.log(this.messageViewChild);
    this.messageViewChild.message = 'Passed as View Child';
    this.cd.detectChanges();
}
```

Manually calling change detection will fix “Expression has changed after it was last checked.” error and it

lifecycle hook

ngAfterViewChecked():

It fires after ngAfterViewInit() and every time the view (and child view) have been checked. This hook is fired after the ngAfterViewInit and after that for every subsequent ngAfterContentChecked hook.

As per the name it both will run with View i.e. whenever view (template) is initialized and if some changes occur.

ngAfterViewChecked()

- Called after all the content is initialized and checked. (Component and child components).
- First call is after ngAfterViewInit()
- Called after every ngAfterContentChecked() call is completed

lifecycle hook

ngOnDestroy():

This hook is called once the component or directive is about to destroy.

Generally, we put logic related to clean up here. This is the right place where we would like to Unsubscribe Observable and detach event handlers to avoid memory leaks.

ngOnDestroy()
Used to clean up any necessary code when a component is removed from the DOM.
Fairly often used to unsubscribe from things like services.
Called only once just before component is removed from the DOM.

lifecycle hook

Interfaces

We can define the lifecycle hook methods directly on the component class, but we can aslo use the advantage of the interface, since each of these lifecycle hook methods has an associated TypeScript interface. The name of those interfaces is the same name as the method, just without the ng prefix.
For example:- **ngOnInit** has an interface called OnInit.

Each interface defines just one lifecycle hook method.

One more important note is that the browser-based TypeScript compiler does not raise a compilation error when we don’t implement interface functions in our class. But, in the compiling time of TypeScript code, it will throw an error.

It is optional to implement **life cycle hook** interface. Angular looks for only **life cycle hook** method name to run this lifecycle hook. Implementing interfaces are optional because JavaScript language doesn't have interfaces and Angular can't see TypeScript interfaces at runtime because they disappear from the transpiled JavaScript.

lifecycle hook

Interfaces

But it is good practice to add interfaces to TypeScript component/directive classes in order to benefit from strong typing and editor tooling.

```
import {
    OnChanges,
    OnInit,
    DoCheck,
    AfterContentInit,
    AfterContentChecked,
    AfterViewInit,
    AfterViewChecked,
    OnDestroy
} from '@angular/core';
class SampleComponent implements
    OnChanges,
    OnInit,
    DoCheck,
    AfterContentInit,
    AfterContentChecked,
    AfterViewInit,
    AfterViewChecked,
    OnDestroy {
  ...
}
```