Lifecycle Hooks

A component has a lifecycle managed by Angular.

Angular creates it, renders it, creates and renders its children, checks it when its data-bound properties change, and destroys it before removing it from the DOM.

Angular offers **lifecycle hooks** that provide visibility into these key life moments and the ability to act when they occur.

A directive has the same set of lifecycle hooks, minus the hooks that are specific to component content and views.

{@a hooks-overview}

Component lifecycle hooks overview

Directive and component instances have a lifecycle as Angular creates, updates, and destroys them. Developers can tap into key moments in that lifecycle by implementing one or more of the *lifecycle hook* interfaces in the Angular `core` library.

Each interface has a single hook method whose name is the interface name prefixed with `ng`. For example, the `OnInit` interface has a hook method named `ngOnInit()` that Angular calls shortly after creating the component:

No directive or component will implement all of the lifecycle hooks and some of the hooks only make sense for components. Angular only calls a directive/component hook method *if it is defined*.

{@a hooks-purpose-timing}

Lifecycle sequence

After creating a component/directive by calling its constructor, Angular calls the lifecycle hook methods in the following sequence at specific moments:

Hook	Purpose and Timing
`ngOnChanges()`	Respond when Angular (re)sets data-bound input properties. The method receives a `SimpleChanges` object of current and previous property values. Called before `ngOnInit()` and whenever one or more data-bound input properties change.
`ngOnInit()`	Initialize the directive/component after Angular first displays the data-bound properties and sets the directive/component's input properties. Called _once_, after the _first_ `ngOnChanges()`.
`ngDoCheck()`	Detect and act upon changes that Angular can't or won't detect on its own. Called during every change detection run, immediately after `ngOnChanges()` and `ngOnInit()`.
`ngAfterContentInit()`	Respond after Angular projects external content into the component's view. Called _once_ after the first `ngDoCheck()`. _A component-only hook_.
`ngAfterContentChecked()`	Respond after Angular checks the content projected into the component. Called after the `ngAfterContentInit()` and every subsequent `ngDoCheck()`. _A component-only hook_.
`ngAfterViewInit()`	Respond after Angular initializes the component's views and child views. Called _once_ after the first `ngAfterContentChecked()`. _A component-only hook_.
`ngAfterViewChecked()`	Respond after Angular checks the component's views and child views. Called after the `ngAfterViewInit` and every subsequent `ngAfterContentChecked()`. _A component-only hook_.
`ngOnDestroy`	Cleanup just before Angular destroys the directive/component. Unsubscribe Observables and detach event handlers to avoid memory leaks. Called _just before_ Angular destroys the directive/component.

{@a interface-optional}

Interfaces are optional (technically)

The interfaces are optional for JavaScript and Typescript developers from a purely technical perspective. The

JavaScript language doesn't have interfaces. Angular can't see TypeScript interfaces at runtime because they disappear from the transpiled JavaScript.

Fortunately, they aren't necessary. You don't have to add the lifecycle hook interfaces to directives and components to benefit from the hooks themselves.

Angular instead inspects directive and component classes and calls the hook methods *if they are defined*. Angular finds and calls methods like `ngOnInit()`, with or without the interfaces.

Nonetheless, it's good practice to add interfaces to TypeScript directive classes in order to benefit from strong typing and editor tooling.

{@a other-lifecycle-hooks}

Other Angular lifecycle hooks

Other Angular sub-systems may have their own lifecycle hooks apart from these component hooks.

3rd party libraries might implement their hooks as well in order to give developers more control over how these libraries are used.

{@a the-sample}

Lifecycle examples

The demonstrates the lifecycle hooks in action through a series of exercises presented as components under the control of the root `AppComponent`.

They follow a common pattern: a *parent* component serves as a test rig for a *child* component that illustrates one or more of the lifecycle hook methods.

Here's a brief description of each exercise:

Component	Description
Peek-a-boo	Demonstrates every lifecycle hook. Each hook method writes to the on-screen log.
Spy	Directives have lifecycle hooks too. A `SpyDirective` can log when the element it spies upon is created or destroyed using the `ngOnInit` and `ngOnDestroy` hooks. This example applies the `SpyDirective` to a `<div>` in an `ngFor` *hero* repeater managed by the parent `SpyComponent`.
OnChanges	See how Angular calls the `ngOnChanges()` hook with a `changes` object every time one of the component input properties changes. Shows how to interpret the `changes` object.
DoCheck	Implements an `ngDoCheck()` method with custom change detection. See how often Angular calls this hook and watch it post changes to a log.
AfterView	Shows what Angular means by a *view*. Demonstrates the `ngAfterViewInit` and `ngAfterViewChecked` hooks.
AfterContent	Shows how to project external content into a component and how to distinguish projected content from a component's view children. Demonstrates the `ngAfterContentInit` and `ngAfterContentChecked` hooks.
Counter	Demonstrates a combination of a component and a directive each with its own hooks. In this example, a `CounterComponent` logs a change (via `ngOnChanges`) every time the parent component increments its input counter property. Meanwhile, the `SpyDirective` from the previous example is applied to the `CounterComponent` log where it watches log entries being created and destroyed.

The remainder of this page discusses selected exercises in further detail.

{@a peek-a-boo}

Peek-a-boo: all hooks

The `PeekABooComponent` demonstrates all of the hooks in one component.

You would rarely, if ever, implement all of the interfaces like this. The peek-a-boo exists to show how Angular calls the hooks in the expected order.

This snapshot reflects the state of the log after the user clicked the *Create...* button and then the *Destroy...*

button.

The sequence of log messages follows the prescribed hook calling order: `OnChanges`, `OnInit`, `DoCheck` (3x), `AfterContentInit`, `AfterContentChecked` (3x), `AfterViewInit`, `AfterViewChecked` (3x), and `OnDestroy`.

The constructor isn't an Angular hook *per se*. The log confirms that input properties (the `name` property in this case) have no assigned values at construction.

Had the user clicked the *Update Hero* button, the log would show another `OnChanges` and two more triplets of `DoCheck`, `AfterContentChecked` and `AfterViewChecked`. Clearly these three hooks fire *often*. Keep the logic in these hooks as lean as possible!

The next examples focus on hook details.

{@a spy}

Spying *OnInit* and *OnDestroy*

Go undercover with these two spy hooks to discover when an element is initialized or destroyed.

This is the perfect infiltration job for a directive. The heroes will never know they're being watched.

Kidding aside, pay attention to two key points: 1. Angular calls hook methods for *directives* as well as components.

2. A spy directive can provide insight into a DOM object that you cannot change directly. Obviously you can't touch the implementation of a native `
`. You can't modify a third party component either. But you can watch both with a directive.

The sneaky spy directive is simple, consisting almost entirely of `ngOnInit()` and `ngOnDestroy()` hooks that log messages to the parent via an injected `LoggerService`.

You can apply the spy to any native or component element and it'll be initialized and destroyed at the same time as that element. Here it is attached to the repeated hero `<div>`:

Each spy's birth and death marks the birth and death of the attached hero `<div>` with an entry in the *Hook Log* as seen here:

Adding a hero results in a new hero `<div>`. The spy's `ngOnInit()` logs that event.

The *Reset* button clears the `heroes` list. Angular removes all hero `<div>` elements from the DOM and destroys their spy directives at the same time. The spy's `ngOnDestroy()` method reports its last moments.

The `ngOnInit()` and `ngOnDestroy()` methods have more vital roles to play in real applications.

{@a oninit}

OnInit()

Use `ngOnInit()` for two main reasons:

1. To perform complex initializations shortly after construction.
2. To set up the component after Angular sets the input properties.

Experienced developers agree that components should be cheap and safe to construct.

Misko Hevery, Angular team lead, [explains why](http://misko.hevery.com/code-reviewers-guide/flaw-constructor-does-real-work/) you should avoid complex constructor logic.

Don't fetch data in a component constructor. You shouldn't worry that a new component will try to contact a remote server when created under test or before you decide to display it. Constructors should do no more than set the initial local variables to simple values.

An `ngOnInit()` is a good place for a component to fetch its initial data. The Tour of Heroes Tutorial guide shows how.

Remember also that a directive's data-bound input properties are not set until *after construction*. That's a problem if you need to initialize the directive based on those properties. They'll have been set when `ngOnInit()` runs.

The `ngOnChanges()` method is your first opportunity to access those properties. Angular calls `ngOnChanges()` before `ngOnInit()` and many times after that. It only calls `ngOnInit()` once.

You can count on Angular to call the `ngOnInit()` method *soon* after creating the component. That's where the heavy initialization logic belongs.

{@a ondestroy}

OnDestroy()

Put cleanup logic in `ngOnDestroy()`, the logic that *must* run before Angular destroys the directive.

This is the time to notify another part of the application that the component is going away.

This is the place to free resources that won't be garbage collected automatically. Unsubscribe from Observables and DOM events. Stop interval timers. Unregister all callbacks that this directive registered with global or application services. You risk memory leaks if you neglect to do so.

{@a onchanges}

OnChanges()

Angular calls its `ngOnChanges()` method whenever it detects changes to ***input properties*** of the component (or directive). This example monitors the `OnChanges` hook.

The `ngOnChanges()` method takes an object that maps each changed property name to a SimpleChange object holding the current and previous property values. This hook iterates over the changed properties and logs them.

The example component, `OnChangesComponent`, has two input properties: `hero` and `power`.

The host `OnChangesParentComponent` binds to them like this:

Here's the sample in action as the user makes changes.

The log entries appear as the string value of the *power* property changes. But the `ngOnChanges` does not catch changes to `hero.name` That's surprising at first.

Angular only calls the hook when the value of the input property changes. The value of the `hero` property is the *reference to the hero object*. Angular doesn't care that the hero's own `name` property changed. The hero object *reference* didn't change so, from Angular's perspective, there is no change to report!

{@a docheck}

DoCheck()

Use the `DoCheck` hook to detect and act upon changes that Angular doesn't catch on its own.

Use this method to detect a change that Angular overlooked.

The *DoCheck* sample extends the *OnChanges* sample with the following `ngDoCheck()` hook:

This code inspects certain *values of interest*, capturing and comparing their current state against previous values. It writes a special message to the log when there are no substantive changes to the `hero` or the `power` so you can see how often `DoCheck` is called. The results are illuminating:

While the `ngDoCheck()` hook can detect when the hero's `name` has changed, it has a frightful cost. This hook is called with enormous frequency—after *every* change detection cycle no matter where the change occurred. It's called over twenty times in this example before the user can do anything.

Most of these initial checks are triggered by Angular's first rendering of *unrelated data elsewhere on the page*. Mere mousing into another `<input>` triggers a call. Relatively few calls reveal actual changes to pertinent data. Clearly our implementation must be very lightweight or the user experience suffers.

{@a afterview}

AfterView

The *AfterView* sample explores the `AfterViewInit()` and `AfterViewChecked()` hooks that Angular calls *after* it creates a component's child views.

Here's a child view that displays a hero's name in an `<input>`:

The `AfterViewComponent` displays this child view *within its template*:

The following hooks take action based on changing values *within the child view*, which can only be reached by querying for the child view via the property decorated with @ViewChild.

{@a wait-a-tick}

Abide by the unidirectional data flow rule

The `doSomething()` method updates the screen when the hero name exceeds 10 characters.

Why does the `doSomething()` method wait a tick before updating `comment`?

Angular's unidirectional data flow rule forbids updates to the view *after* it has been composed. Both of these hooks fire *after* the component's view has been composed.

Angular throws an error if the hook updates the component's data-bound `comment` property immediately (try it!). The `LoggerService.tick_then()` postpones the log update for one turn of the browser's JavaScript cycle and that's just long enough.

Here's *AfterView* in action:

Notice that Angular frequently calls `AfterViewChecked()`, often when there are no changes of interest. Write lean hook methods to avoid performance problems.

{@a aftercontent}

AfterContent

The *AfterContent* sample explores the `AfterContentInit()` and `AfterContentChecked()` hooks that Angular calls *after* Angular projects external content into the component.

{@a content-projection}

Content projection

Content projection is a way to import HTML content from outside the component and insert that content into the component's template in a designated spot.

AngularJS developers know this technique as *transclusion*.

Consider this variation on the previous *AfterView* example. This time, instead of including the child view within the template, it imports the content from the `AfterContentComponent`'s parent. Here's the parent's template:

Notice that the `<my-child>` tag is tucked between the `<after-content>` tags. Never put content between a component's element tags *unless you intend to project that content into the component*.

Now look at the component's template:

The `<ng-content>` tag is a *placeholder* for the external content. It tells Angular where to insert that content. In this case, the projected content is the `<my-child>` from the parent.

The telltale signs of *content projection* are twofold: * HTML between component element tags. * The presence of `` tags in the component's template.

{@a aftercontent-hooks}

AfterContent hooks

AfterContent hooks are similar to the *AfterView* hooks. The key difference is in the child component.

- The *AfterView* hooks concern `ViewChildren`, the child components whose element tags appear *within* the component's template.
- The *AfterContent* hooks concern `ContentChildren`, the child components that Angular projected into the component.

The following *AfterContent* hooks take action based on changing values in a *content child*, which can only be reached by querying for them via the property decorated with @ContentChild.

{@a no-unidirectional-flow-worries}

No unidirectional flow worries with *AfterContent*

This component's `doSomething()` method update's the component's data-bound `comment` property immediately. There's no need to wait.

Recall that Angular calls both *AfterContent* hooks before calling either of the *AfterView* hooks. Angular completes composition of the projected content *before* finishing the composition of this component's view. There is a small window between the `AfterContent...` and `AfterView...` hooks to modify the host view.