Lifecycle hooks

A component instance has a lifecycle that starts when Angular instantiates the component class and renders the component view along with its child views. The lifecycle continues with change detection, as Angular checks to see when data-bound properties change, and updates both the view and the component instance as needed. The lifecycle ends when Angular destroys the component instance and removes its rendered template from the DOM. Directives have a similar lifecycle, as Angular creates, updates, and destroys instances in the course of execution.

Your application can use lifecycle hook methods to tap into key events in the lifecycle of a component or directive to initialize new instances, initiate change detection when needed, respond to updates during change detection, and clean up before deletion of instances.

Prerequisites

Before working with lifecycle hooks, you should have a basic understanding of the following:

- TypeScript programming.
- Angular app-design fundamentals, as described in Angular Concepts.

{@a hooks-overview}

Responding to lifecycle events

Respond to events in the lifecycle of a component or directive by implementing one or more of the *lifecycle hook* interfaces in the Angular `core` library. The hooks give you the opportunity to act on a component or directive instance at the appropriate moment, as Angular creates, updates, or destroys that instance.

Each interface defines the prototype for a single hook method, whose name is the interface name prefixed with `ng`. For example, the `OnInit` interface has a hook method named `ngOnInit()`. If you implement this method in your component or directive class, Angular calls it shortly after checking the input properties for that component or directive for the first time.

You don't have to implement all (or any) of the lifecycle hooks, just the ones you need.

{@a hooks-purpose-timing}

Lifecycle event sequence

After your application instantiates a component or directive by calling its constructor, Angular calls the hook methods you have implemented at the appropriate point in the lifecycle of that instance.

Angular executes hook methods in the following sequence. Use them to perform the following kinds of operations.

Hook method

Purpose

Timing

ngOnChanges()

```
  Respond when Angular sets or resets data-bound input properties.
  The method receives a `SimpleChanges` object of current and previous property values.

  Note that this happens very frequently, so any operation you perform here impacts performa
  See details in [Using change detection hooks](#onchanges) in this document.

</td>
<td>

  Called before `ngOnInit()` (if the component has bound inputs) and whenever one or more da

  Note that if your component has no inputs or you use it without providing any inputs, the

</td>
```

ngOnInit()

```
  Initialize the directive or component after Angular first displays the data-bound properti
  and sets the directive or component's input properties.
  See details in [Initializing a component or directive](#oninit) in this document.

</td>
<td>

  Called once, after the first `ngOnChanges()`. `ngOnInit()` is still called even when `ngOn

</td>
```

ngDoCheck()

```
  Detect and act upon changes that Angular can't or won't detect on its own.
  See details and example in [Defining custom change detection](#docheck) in this document.

</td>
<td>

Called immediately after `ngOnChanges()` on every change detection run, and immediately afte

</td>
```

ngAfterContentInit()

```
  Respond after Angular projects external content into the component's view, or into the vie

  See details and example in [Responding to changes in content](#aftercontent) in this docu


</td>
<td>

  Called _once_ after the first `ngDoCheck()`.

</td>
```

ngAfterContentChecked()

```
  Respond after Angular checks the content projected into the directive or component.

  See details and example in [Responding to projected content changes](#aftercontent) in thi

</td>

<td>

  Called after `ngAfterContentInit()` and every subsequent `ngDoCheck()`.

</td>
```

ngAfterViewInit()

```
  Respond after Angular initializes the component's views and child views, or the view that

  See details and example in [Responding to view changes](#afterview) in this document.

</td>

<td>

  Called _once_ after the first `ngAfterContentChecked()`.
</td>
```

ngAfterViewChecked()

```
  Respond after Angular checks the component's views and child views, or the view that conta

</td>

<td>
```

```
  Called after the `ngAfterViewInit()` and every subsequent `ngAfterContentChecked()`.

</td>
```

ngOnDestroy()

```
  Cleanup just before Angular destroys the directive or component.
  Unsubscribe Observables and detach event handlers to avoid memory leaks.
  See details in [Cleaning up on instance destruction](#ondestroy) in this document.

</td>

<td>

  Called immediately before Angular destroys the directive or component.

</td>
```

{@a the-sample}

Lifecycle example set

The demonstrates the use of lifecycle hooks through a series of exercises presented as components under the control of the root `AppComponent`. In each case a *parent* component serves as a test rig for a *child* component that illustrates one or more of the lifecycle hook methods.

The following table lists the exercises with brief descriptions. The sample code is also used to illustrate specific tasks in the following sections.

Component

Description

Peek-a-boo

```
  Demonstrates every lifecycle hook.
  Each hook method writes to the on-screen log.

</td>
```

Spy

```
  Shows how to use lifecycle hooks with a custom directive.
  The `SpyDirective` implements the `ngOnInit()` and `ngOnDestroy()` hooks,
  and uses them to watch and report when an element goes in or out of the current view.

</td>
```

OnChanges

```
  Demonstrates how Angular calls the `ngOnChanges()` hook
  every time one of the component input properties changes,
  and shows how to interpret the `changes` object passed to the hook method.

</td>
```

DoCheck

```
  Implements the `ngDoCheck()` method with custom change detection.
  Watch the hook post changes to a log to see how often Angular calls this hook.

</td>
```

AfterView

```
  Shows what Angular means by a [view](guide/glossary#view "Definition of view.").
  Demonstrates the `ngAfterViewInit()` and `ngAfterViewChecked()` hooks.

</td>
```

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.

</td>
```

Counter

```
  Demonstrates a combination of a component and a directive, each with its own hooks.

</td>
```

{@a oninit}

Initializing a component or directive

Use the `ngOnInit()` method to perform the following initialization tasks.

- Perform complex initializations outside of the constructor. Components should be cheap and safe to construct. You should not, for example, 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.

  An `ngOnInit()` is a good place for a component to fetch its initial data. For an example, see the Tour of Heroes tutorial.

- Set up the component after Angular sets the input properties. Constructors should do no more than set the initial local variables to simple values.

Keep in mind that a directive's data-bound input properties are not set until *after construction*. If you need to initialize the directive based on those properties, set them when `ngOnInit()` runs.

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

{@a ondestroy}

Cleaning up on instance destruction

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

This is the place to free resources that won't be garbage-collected automatically. You risk memory leaks if you neglect to do so.

- Unsubscribe from Observables and DOM events.
- Stop interval timers.
- Unregister all callbacks that the directive registered with global or application services.

The `ngOnDestroy()` method is also the time to notify another part of the application that the component is going away.

General examples

The following examples demonstrate the call sequence and relative frequency of the various lifecycle events, and how the hooks can be used separately or together for components and directives.

{@a peek-a-boo}

Sequence and frequency of all lifecycle events

To show how Angular calls the hooks in the expected order, the `PeekABooComponent` demonstrates all of the hooks in one component.

In practice you would rarely, if ever, implement all of the interfaces the way this demo does.

The following 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`.

Notice that the log confirms that input properties (the `name` property in this case) have no assigned values at construction. The input properties are available to the `onInit()` method for further initialization.

Had the user clicked the *Update Hero* button, the log would show another `OnChanges` and two more triplets of `DoCheck`, `AfterContentChecked` and `AfterViewChecked`. Notice that these three hooks fire *often*, so it is important to keep their logic as lean as possible.

{@a spy}

Use directives to watch the DOM

The `Spy` example demonstrates how to use the hook method for directives as well as components. The `SpyDirective` implements two hooks, `ngOnInit()` and `ngOnDestroy()`, to discover when a watched element is in the current view.

This template applies the `SpyDirective` to a `<div>` in the `ngFor` *hero* repeater managed by the parent `SpyComponent`.

The example does not perform any initialization or clean-up. It just tracks the appearance and disappearance of an element in the view by recording when the directive itself is instantiated and destroyed.

A spy directive like this can provide insight into a DOM object that you cannot change directly. You can't touch the implementation of a built-in `<div>`, or modify a third party component. You can, however watch these elements with a directive.

The directive defines `ngOnInit()` and `ngOnDestroy()` hooks that log messages to the parent using an injected `LoggerService`.

Apply the spy to any built-in or component element, and see that it is initialized and destroyed at the same time as that element. Here it is attached to the repeated hero `<p>`:

Each spy's creation and destruction marks the appearance and disappearance of the attached hero `<p>` with an entry in the *Hook Log*. Adding a hero results in a new hero `<p>`. The spy's `ngOnInit()` logs that event.

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

{@a counter}

Use component and directive hooks together

In this example, a `CounterComponent` uses the `ngOnChanges()` method to log a change every time the parent component increments its input `counter` property.

This example applies the `SpyDirective` from the previous example to the `CounterComponent` log, to watch the creation and destruction of log entries.

{@a onchanges}

Using change detection hooks

Angular calls the `ngOnChanges()` method of a component or directive whenever it detects changes to the ***input properties***. The *onChanges* example demonstrates this by monitoring 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 as follows.

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

The log entries appear as the string value of the *power* property changes. Notice, however, that the `ngOnChanges()` method does not catch changes to `hero.name`. This is because Angular calls the hook only when the value of the input property changes. In this case, `hero` is the input property, and the value of the `hero` property is the *reference to the hero object*. The object reference did not change when the value of its own `name` property changed.

{@a afterview}

Responding to view changes

As Angular traverses the view hierarchy during change detection, it needs to be sure that a change in a child does not attempt to cause a change in its own parent. Such a change would not be rendered properly, because of how unidirectional data flow works.

If you need to make a change that inverts the expected data flow, you must trigger a new change detection cycle to allow that change to be rendered. The examples illustrate how to make such changes safely.

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 using the property decorated with @ViewChild.

{@a wait-a-tick}

Wait before updating the view In this example, the `doSomething()` method updates the screen when the hero name exceeds 10 characters, but waits a tick before updating `comment`.

Both the `AfterViewInit()` and `AfterViewChecked()` hooks fire after the component's view is composed. If you modify the code so that the hook updates the component's data-bound `comment` property immediately, you can see that Angular throws an error.

The `LoggerService.tick_then()` statement postpones the log update for one turn of the browser's JavaScript cycle, which triggers a new change-detection cycle.

Write lean hook methods to avoid performance problems When you run the *AfterView* sample, notice how frequently Angular calls `AfterViewChecked()`-often when there are no changes of interest. Be very careful about how much logic or computation you put into one of these methods.

{@a aftercontent} {@a aftercontent-hooks} {@a content-projection}

Responding to projected content changes

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. Identify content projection in a template by looking for the following constructs.

- HTML between component element tags.
- The presence of `<ng-content>` tags in the component's template.

AngularJS developers know this technique as *transclusion*.

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

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. The following is the parent's template.

Notice that the `<app-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 `<app-child>` from the parent.

Using 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 using the property decorated with @ContentChild.

{@a no-unidirectional-flow-worries}

No need to wait for content updates

This component's `doSomething()` method updates the component's data-bound `comment` property immediately. There's no need to delay the update to ensure proper rendering.

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 that lets you modify the host view.

{@a docheck}

Defining custom change detection

To monitor changes that occur where `ngOnChanges()` won't catch them, implement your own change check, as shown in the *DoCheck* example. This example shows how to use the `ngDoCheck()` hook to detect and act upon changes that Angular doesn't catch on its own.

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 is very expensive. 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*. Just moving the cursor into another `<input>` triggers a call. Relatively few calls reveal actual changes to pertinent data. If you use this

hook, your implementation must be extremely lightweight or the user experience suffers.

@reviewed 2021-09-16