Everything you need to know about change detection in Angular

Exploring the underlying implementation and use cases

Max Koretskyi aka Wizard Follow
Apr 4, 2017 · 11 min read

> **We're organizing the first Angular conference in Kiev, Ukraine as part of Angular In Depth community. Read more about it here.** I'll be happy to see you at the conference. We'll have experts from Angular team and the community and we made tickets affordable for most developers (around $150), even if you pay out of your own pocket.

If like me and want to have a comprehensive understanding of the change detection mechanism in Angular you'll have to explore the sources since there is not much information available on the web. Most articles mention that each component has its own change detector which is responsible for checking the component, but they

don’t go beyond that and mostly focus on use cases for immutables and change detection strategy. This article provides you with the information required to understand *why* use cases with immutables work and *how* change detection strategy affects the check. Also, what you will learn from this article will enable you to come up with various scenarios for performance optimization on your own.

> *This article is quite in-depth. For an introduction into change detection, check out A gentle introduction into change detection in Angular*

This article consists of two parts. The first part is pretty technical and contains a lot of links to the sources. It explains in detail how the change detection mechanism works under the hood. Its content is based on the newest Angular version—4.0.1. The way how change detection mechanism is implemented under the hood in this version is different from the earlier 2.4.1. If interested you can read a little about how it worked in this stackoverflow answer.

The second part shows how change detection can be used in the application and its content is applicable for both earlier 2.4.1 and the newest 4.0.1 versions of Angular since public API has not changed.

> I work as a developer advocate at **ag-Grid**. If you’re curious to learn about data grids or looking for the ultimate Angular data grid solution, give it a try with the guide “**Get started with Angular grid in 5 minutes**”. I’m happy to answer any questions you may have. **And follow me to stay tuned!**

• • •

View as a core concept

It’s been mentioned across the tutorials that an Angular application is a tree of components. However, under the hood angular uses a low-level abstraction called view. There is a direct relationship between a view and a component—one view is associated with one component and vice verse. A view holds a reference to the associated component class instance in the `component` property. All operations like property checks and DOM updates are performed on views, hence it’s more technically correct to state that angular is a tree of views, while a component can be described as a higher level concept of a view. Here is what you can read about the view in the sources:

> *A View is a fundamental building block of the application UI. It is the smallest grouping of Elements which are created and destroyed together.*

> *Properties of elements in a View can change, but the structure (number and order) of elements in a View cannot. Changing the structure of Elements can only be done by inserting, moving or removing nested Views via a ViewContainerRef. Each View can contain many View Containers.*

In this article I will be using notions of component view and component interchangeably.

> *It's important to note here that all articles on the web and answers on StackOverflow regarding change detection refer to the View I'm describing here as Change Detector Object or ChangeDetectorRef. In reality, there's no separate object for change detection and View is what change detection runs on.*

Each view has a link to its child views through nodes property and hence can perform actions on child views.

View state

Each view has a state, which plays very important role because based on its value Angular decides whether to run change detection for the view and **all its children** or skip it. There are many possible states but the following ones are relevant in the context of this article:

1. FirstCheck
2. ChecksEnabled
3. Errored
4. Destroyed

Change detection is skipped for the view and its child views if `ChecksEnabled` is `false` or view is in the `Errored` or `Destroyed`

state. By default, all views are initialized with `ChecksEnabled` unless `ChangeDetectionStrategy.OnPush` is used. More on that later. The states can be combined, for example, a view can have both `FirstCheck` and `ChecksEnabled` flags set.

Angular has a bunch of high-level concepts to manipulate the views. I've written about some of them here. One such concept is ViewRef. It encapsulates the underlying component view and has an aptly named method detectChanges. When an asynchronous event takes place, Angular triggers change detection on its top-most ViewRef, which after running change detection for itself **runs change detection for its child views**.

This `viewRef` is what you can inject into a component constructor using `ChangeDetectorRef` token:

```
export class AppComponent {
    constructor(cd: ChangeDetectorRef) { ... }
```

As can be seen from this classes definition:

```
export declare abstract class ChangeDetectorRef {
    abstract checkNoChanges(): void;
    abstract detach(): void;
    abstract detectChanges(): void;
    abstract markForCheck(): void;
    abstract reattach(): void;
}
```

```
export abstract class ViewRef extends ChangeDetectorRef {
   ...
}
```

• • •

Change detection operations

The main logic responsible for running change detection for a view resides in checkAndUpdateView function. Most of its functionality performs operations on **child** component views. This function **is called recursively** for each component starting from the host

component. It means that a child component becomes parent component on the next call as a recursive tree unfolds.

When this function triggered for a particular view it does the following operations in the specified order:

1. sets `ViewState.firstCheck` to `true` if a view is checked for the first time and to `false` if it was already checked before
2. checks and updates input properties on a child component/directive instance
3. updates child view change detection state (part of change detection strategy implementation)
4. runs change detection for the embedded views (repeats the steps in the list)
5. calls `OnChanges` lifecycle hook on a child component if bindings changed
6. calls `OnInit` and `ngDoCheck` on a child component (`OnInit` is called only during first check)
7. updates `ContentChildren` query list on a child view component instance
8. calls `AfterContentInit` and `AfterContentChecked` lifecycle hooks on child component instance (`AfterContentInit` is called only during first check)
9. updates DOM interpolations for the **current view** if properties on **current view** component instance changed
10. runs change detection for a child view (repeats the steps in this list)
11. updates `ViewChildren` query list on the current view component instance
12. calls `AfterViewInit` and `AfterViewChecked` lifecycle hooks on child component instance (`AfterViewInit` is called only during first check)
13. disables checks for the current view (part of change detection strategy implementation)

There are few things to highlight based on the operations listed above.

The first thing is that `onChanges` lifecycle hook is triggered on a child component before the child view is checked and it will be triggered even if changed detection for the child view will be skipped. This is important information and we will see how we can leverage this knowledge in the second part of the article.

The second thing is that DOM for a view is updated as part of a change detection mechanism while the view being checked. This means that if a component is not checked, the DOM is not updated even if component properties used in a template change. The templates are rendered before the first check. What I refer to as DOM update is actually interpolation update. So if you have `<span>some {{name}}</span>`, the DOM element `span` will be rendered before the first check. During the check only `{{name}}` part will be rendered.

Another interesting observation is that state of a child component view can be changed during change detection. I mentioned earlier that all component views are initialized with `ChecksEnabled` by default, but for all components that use `OnPush` strategy change detection is disabled after the first check (operation 9 in the list):

```
if (view.def.flags & ViewFlags.OnPush) {
  view.state &= ~ViewState.ChecksEnabled;
}
```

It means that during the following change detection run the check will be skipped for this component view and all its children. The documentation about the `OnPush` strategy states that a component will be checked only if its bindings have changed. So to do that the checks have to be enabled by setting `ChecksEnabled` bit. And this is what the following code does (operation 2):

```
if (compView.def.flags & ViewFlags.OnPush) {
  compView.state |= ViewState.ChecksEnabled;
}
```

The state is updated only if parent view bindings changed and child component view was initialized with `ChangeDetectionStrategy.OnPush`.

Finally, change detection for the current view is responsible for starting change detection for child views (operation 8). This is the place where state of the child component view is checked and if it's `ChecksEnabled`, then for this view the change detection is performed. Here is the relevant code:

```
viewState = view.state;
...
case ViewAction.CheckAndUpdate:
  if ((viewState & ViewState.ChecksEnabled) &&
    (viewState & (ViewState.Errored | ViewState.Destroyed))
=== 0) {
    checkAndUpdateView(view);
  }
}
```

Now you know that view state controls whether change detection is performed for this view and its children or not. So the question begs —can we control that state? It turns out we can and this is what the second part of this article is about.

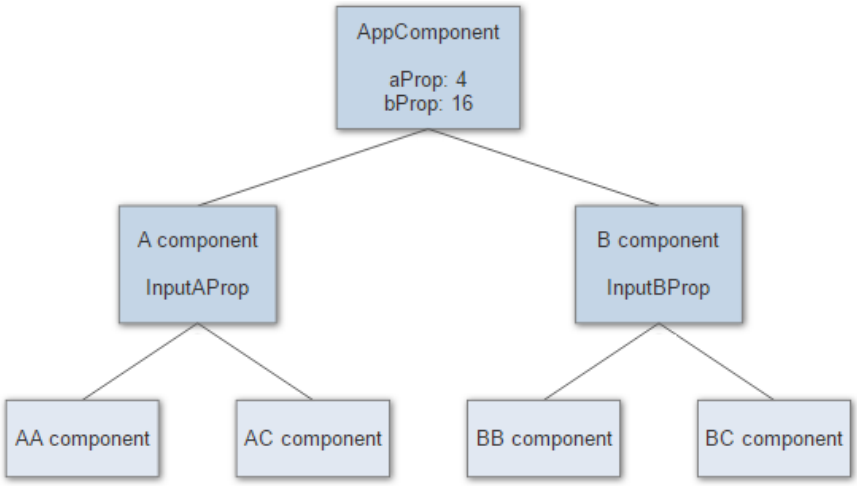
Some lifecycle hooks are called before the DOM update (3,4,5) and some after (9). So if you have the following components hierarchy: `A -> B -> C`, here is the order of hooks calls and bindings updates:

```
A: AfterContentInit
A: AfterContentChecked
A: Update bindings
    B: AfterContentInit
    B: AfterContentChecked
    B: Update bindings
        C: AfterContentInit
        C: AfterContentChecked
        C: Update bindings
        C: AfterViewInit
        C: AfterViewChecked
    B: AfterViewInit
    B: AfterViewChecked
A: AfterViewInit
A: AfterViewChecked
```

• • •

Exploring the implications

Let's assume that we have the following components tree:



A tree of components

As we learnt above, each component is associated with a component view. Each view is initialized with the `ViewState.ChecksEnabled` which means when angular runs change detection every component in the tree will be checked.

Suppose we want to disable change detection for the `AComponent` and its children. That's easy to do—we just need to set `ViewState.ChecksEnabled` to `false`. Changing state is a low-level operation, so Angular provides for us a bunch of public methods available on the view. Every component can get a hold of its associated view through `ChangeDetectorRef` token. For this class Angular docs define the following public interface:

```
class ChangeDetectorRef {
  markForCheck() : void
  detach() : void
  reattach() : void

  detectChanges() : void
  checkNoChanges() : void
}
```

Let's see what we can wrangle it to our benefit.

detach

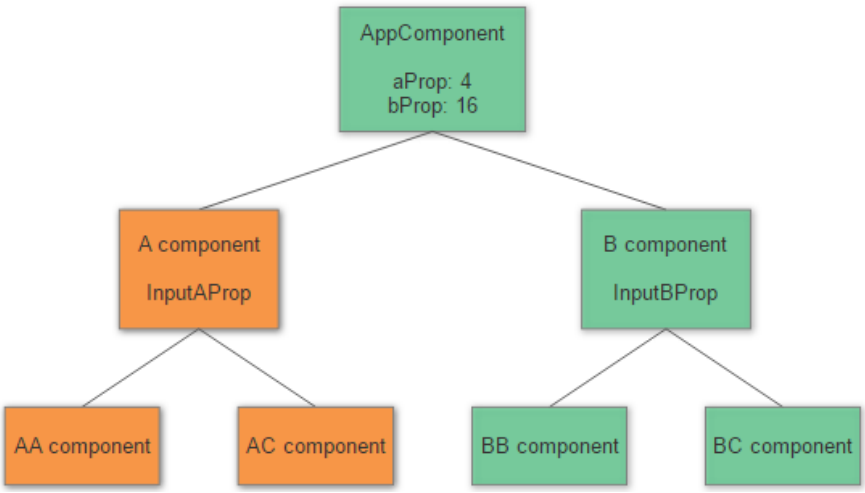
The first method that allows us manipulating the state is `detach` which simply disables checks for the current view:

```
detach(): void { this._view.state &=
~ViewState.ChecksEnabled; }
```

Let's see how it can be used in the code:

```
export class AComponent {
  constructor(public cd: ChangeDetectorRef) {
    this.cd.detach();
  }
```

This ensures that during the following change detection runs the left branch starting with `AComponent` will be skipped (orange components will not be checked):



There are two things to note here—first is that even though we changed state for `AComponent`, all its child components will not be checked as well. Second is that since no change detection will be performed for the left branch components, DOM in their templates will not be updated as well. Here is the small example to demonstrate it:

```
@Component({
  selector: 'a-comp',
  template: `<span>See if I change: {{changed}}</span>`
})
export class AComponent {
  constructor(public cd: ChangeDetectorRef) {
    this.changed = 'false';
```

```
      setTimeout(() => {
        this.cd.detach();
        this.changed = 'true';
      }, 2000);
    }
```

The first time the component is checked the span will be rendered with the text `See if I change: false`. And within two seconds when `changed` property is updated to `true` the text in the span will not be changed. However, if we remove this line `this.cd.detach()`, everything will work as expected.

reattach

As shown in the first part of the article `OnChanges` lifecycle hook will still be triggered for `AComponent` if input binding `aProp` changes on the `AppComponent`. This means that once we are notified that input properties change, we can activate change detector for the current component to run change detection and detach it on the next tick. Here is the snippet demonstrating that:

```
export class AComponent {
  @Input() inputAProp;

  constructor(public cd: ChangeDetectorRef) {
    this.cd.detach();
  }

  ngOnChanges(values) {
    this.cd.reattach();
    setTimeout(() => {
      this.cd.detach();
    })
  }
```

Since `reattach` simply sets `ViewState.ChecksEnabled` bit:

```
reattach(): void { this._view.state |=
ViewState.ChecksEnabled; }
```

this is almost equivalent to what is done when `ChangeDetectionStrategy` is set to `OnPush`: disable check after the first change detection run, enable it when parent component bound property changes and disable after the run.

Please note that `OnChanges` hook is only triggered for the top-most component in the disabled branch, not for every component in the disabled branch.

markForCheck

The `reattach` method enables checks for the current component only, but if changed detection is not enabled for its parent component, it will have no effect. It means that `reattach` method is only useful for top-most component in the disabled branch.

We need a way to enable check for all parent components up to root component. And there is a method for it `markForCheck`:

```
let currView: ViewData|null = view;
while (currView) {
  if (currView.def.flags & ViewFlags.OnPush) {
    currView.state |= ViewState.ChecksEnabled;
  }
  currView = currView.viewContainerParent ||
currView.parent;
}
```

As you can see from the implementation, it simply iterates upwards and enables checks for every parent component up to the root.

When is this useful? Just as with `ngOnChanges` the `ngDoCheck` lifecycle hook is triggered even if the component uses `OnPush` strategy. Again, it's only triggered for the top-most component in the disabled branch, not for every component in the disabled branch. But we can use this hook to perform custom logic and mark our component eligible for one change detection cycle run. Since Angular only checks object references, we may implement the dirty checking of some object property:

```
Component({
   ...,
   changeDetection: ChangeDetectionStrategy.OnPush
})
MyComponent {
   @Input() items;
   prevLength;
   constructor(cd: ChangeDetectorRef) {}

   ngOnInit() {
      this.prevLength = this.items.length;
   }
```

```
    ngDoCheck() {
       if (this.items.length !== this.prevLength) {
          this.cd.markForCheck();
          this.prevLenght = this.items.length;
       }
    }
```

detectChanges

There is a way to run change detection **once** for the current component and all its children. This is done using `detectChanges` method. This method runs change detection for the current component view regardless of its state, which means that checks may remain disabled for the current view and component will not be checked during following regular change detection runs. Here is an example:

```
export class AComponent {
  @Input() inputAProp;

  constructor(public cd: ChangeDetectorRef) {
    this.cd.detach();
  }

  ngOnChanges(values) {
    this.cd.detectChanges();
  }
```

DOM is updated when input property changes even though change detector reference remains detached.

checkNoChanges

This last method available on the change detector ensures that there will be no changes done on the current run of change detection. Basically, it performs 1,7,8 operations from the list in the first article and throws an exception if it finds a changed binding or determines that DOM should be updated.

• • •

Have a question?

If you have a clarifying question regarding this article contents, I suggest you create a question on stackoverflow and post a link here in

the comments. In this way the entire community benefits. Thanks

. . .

Thanks for reading! If you liked this article, hit that clap button below 👏. It means a lot to me and it helps other people see the story. For more insights follow me on Twitter and on Medium.

. . .