A gentle introduction into change detection in Angular

A high-level overview of the change detection mechanism, zones and the ExpressionChangedAfterItHasBeenCheckedError error

Max Koretskyi aka Wizard Follow
Dec 4, 2018 · 13 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.

Modern web applications are interactive. The state of an application can change anytime as a result of a button click or request coming back from a server. And as the state changes the code needs to detect that and reflect the change in the User Interface. That's the main job of the change detection mechanism.

> *If you prefer watching over reading, check out this talk I gave at AngularConnect.*

Over the last year I've written a lot of in-depth articles on the mechanics of **change detection** in Angular. They provide elaborate explanations and cover a lot of the internal details. But, they also require a lot of time to read thoroughly. For those of you who don't have the time but are, nevertheless, curious: this article provides a "lighter" explanation of the change detection mechanism. It'll give you a high-level overview of its constituent parts and mechanics: internal data structures used to represent a component, the role of bindings and the operations performed as part of the process. I'll also touch on zones and show you exactly how this functionality enables automatic change detection in Angular.

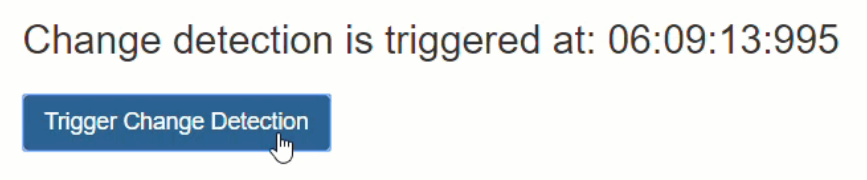
When things go awry, a knowledge of change detection internals will help you debug errors, like `ExpressionChangedAfterItHasBeenCheckedError`, more efficiently and avoid some common confusions. In this article I'll demonstrate a few setups that cause the error and use them to explain the internals of change detection.

> *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!***

. . .

First encounter

Let's start with this simple Angular component. It renders the time at the moment that change detection happens in the application. The timestamp has millisecond precision. Clicking on the button triggers change detection:



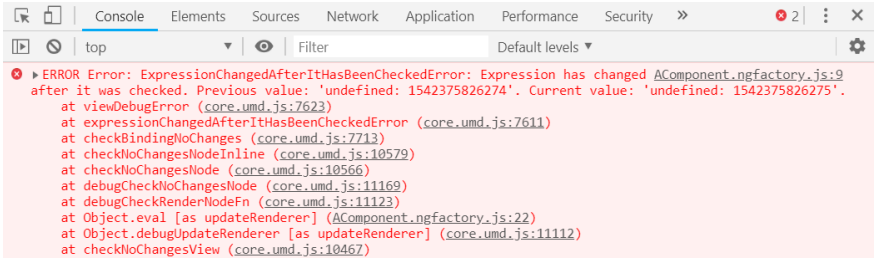
Here's the implementation:

```
@Component({
    selector: 'my-app',
    template: `
        <h3>
            Change detection is triggered at:
            <span [textContent]="time | date:'hh:mm:ss
        </h3>
        <button (click)="0">Trigger Change Detection</
    `
})
export class AppComponent {
```

As you can see it's rather basic. There's a getter named `time` that returns the current timestamp. And, I'm binding it to the `span` element in HTML.

> *Angular doesn't allow empty expressions, so I've put* `0` *as the click callback.*

You can play with it here. When Angular runs change detection, it takes the value of the `time` property, passes it through the `date` pipe and uses the result to update the DOM. Everything works as expected. However, when I check the console I see the `ExpressionChangedAfterItHasBeenCheckedError` error:



That's actually quite surprising. Usually that error comes up in a lot more sophisticated implementations. So how is it possible that we get it with such simple functionality? Don't worry, we're going to investigate it now.

Let's start with the error message:

> *Expression has changed after it was checked. Previous value: "textContent: 1542375826274". Current value: "textContent: 1542375826275".*

It tells us that the values produced by expressions for the `textContent` bindings are different. Yes, the milliseconds are indeed different. So Angular evaluated the expression `time | date:'hh:mm:ss:SSS'` twice and compared the results. It detected the difference and that is what caused the error.

But why does Angular perform that comparison?
Or when exactly does it do it?

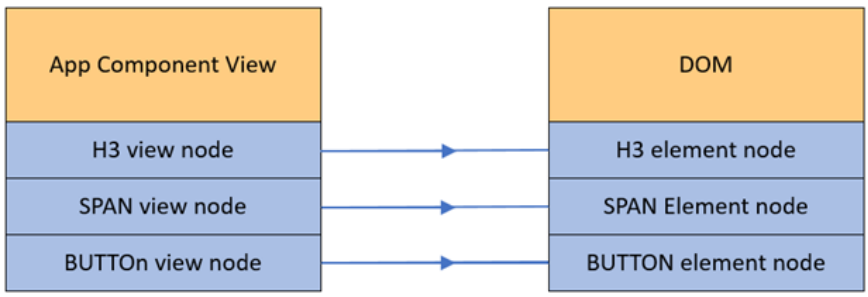
These were the questions that sparked my curiosity and eventually led me down into the internals of change detection. Because, to find out answers to these questions I had to start debugging. And I was debugging and debugging and, well, I think it lasted for about... a few months 😅 . Let's start with the second question of when the the error is thrown. But first, I need to share with you some of my findings that will help us understand the behavior we've just observed above.

Component views and bindings

There are two main building blocks of change detection in Angular:

- a component view
- the associated bindings

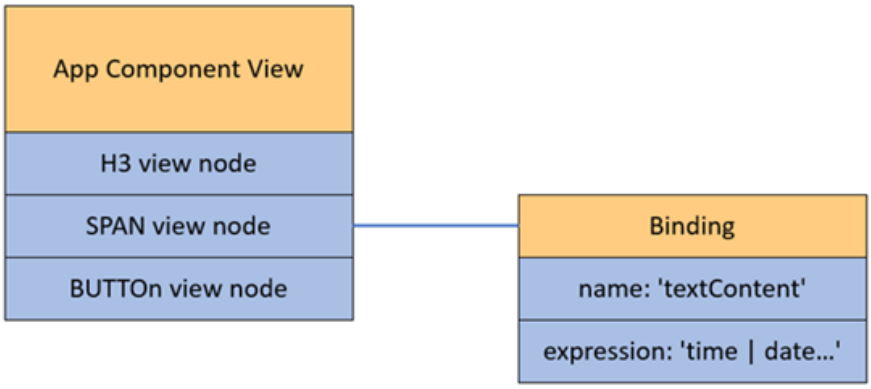
Every component in Angular has a template with HTML elements. When Angular creates the DOM nodes to render the contents of the template on the screen, it needs a place to store the references to those DOM nodes. For that purpose, internally there's a data structure known as View. It's also used to store the reference to the component instance and the previous values of binding expressions. There's a one to one relationship between a component and a view. Here's the diagram that demonstrates the view:



As the compiler analyzes the template, it identifies properties of the DOM elements that may need to be updated during change detection. For each such property, the compiler creates a **binding**. The binding

defines the property name to update and the expression that Angular uses to obtain a new value.

In our case, the property `time` is used in the expression for the property `textContent`. So Angular creates a binding and associates it with the `span` element:



> *In the actual implementation the binding is not a single object with all required information. A* `viewDefinition` *defines actual bindings for template elements and the properties to update. The expression used for a binding is placed in the* `updateRenderer` *function.*

Checking a component view

As you know, in Angular, change detection is performed for each component. Now that we know that components internally are represented as views, we can say that change detection is performed for each view.

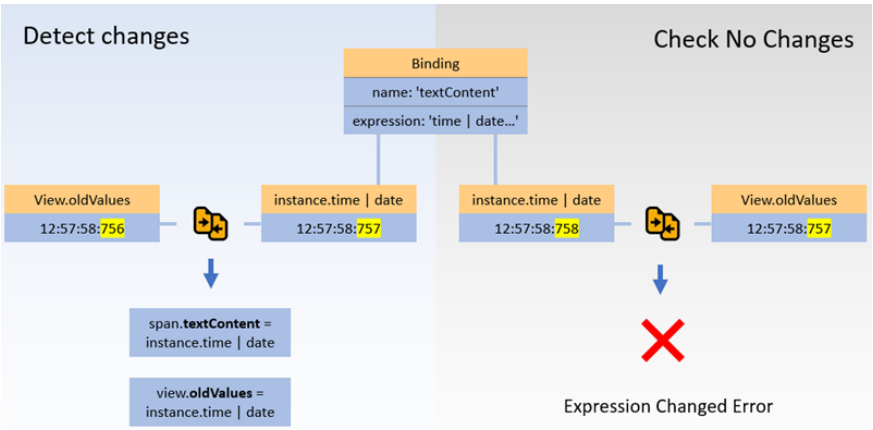
When Angular checks a view, it simply runs over all bindings generated for a view by the compiler. It evaluates expressions and compares their result to the values stored in the `oldValues` array on the view. That's where the name dirty checking comes from. If it detects the difference, it updates the DOM property relevant to the binding. And it also needs to put the new value into the `oldValues` array on the view. And that's it. You now have an updated UI. Once Angular is done checking the current component, it repeats exactly the same steps for child components.

In our application, there's only one binding to the property `textContent` of the `span` element in the `App` component. So during change detection Angular reads the value of the component's property `time`, applies the `date` pipe, and compares it to the previous value stored on the view. If it detects a difference, Angular

updates the `textContent` property of the span and the `oldValues` array.

But where does the error come from?

After each change detection cycle, in the development mode, Angular **synchronously** runs another check to ensure that expressions produce the same values as during the preceding change detection run. This check is not part of the original change detection cycle. It runs **after** the check is finished for the entire tree of components and performs exactly the same steps. However, this time, as Angular detects the difference, it doesn't update the DOM. Instead, it throws the `ExpressionChangedAfterItHasBeenCheckedError`.



The why

So now we know when the error is thrown. **But why does Angular need this check?** Well, imagine that some properties of components have been updated during the change detection run. As a result, expressions produce new values that are inconsistent with what's rendered in the user interface. So, what does Angular do? It certainly could run another change detection cycle to synchronize the application state with the user interface. But what if during that process some properties are updated again? See the pattern? Angular could actually end up in an infinite loop of change detection runs. And atually, that happened quite often in AngularJS.

To avoid that situation, Angular imposed the so-called Unidirectional Data Flow. And this check that runs after change detection and the resulting `ExpressionChangedAfterItHasBeenCheckedError` error is the enforcement mechanism. Once Angular has processed bindings for the current component, you can no longer update the properties of the component that are used in expressions for bindings.

Fixing the error

To prevent the error, we need to ensure that the values returned by expressions during the change detection run and the following check are the same. In our case, we can do that by moving the evaluation part out of the `time` getter like this:

```
export class AppComponent {
    _time;
    get time() {  return this._time; }

    constructor() {
        this._time = Date.now();
```

However, with this implementation the value for the getter `time` will always be the same. We still need to update the value. We learned earlier that the check that produces the error runs **synchronously** right after the change detection cycle. So if we update it **asynchronously**, we will avoid the error. So to update the value every millisecond we can use the `setInterval` function with `1` millisecond delay like this:

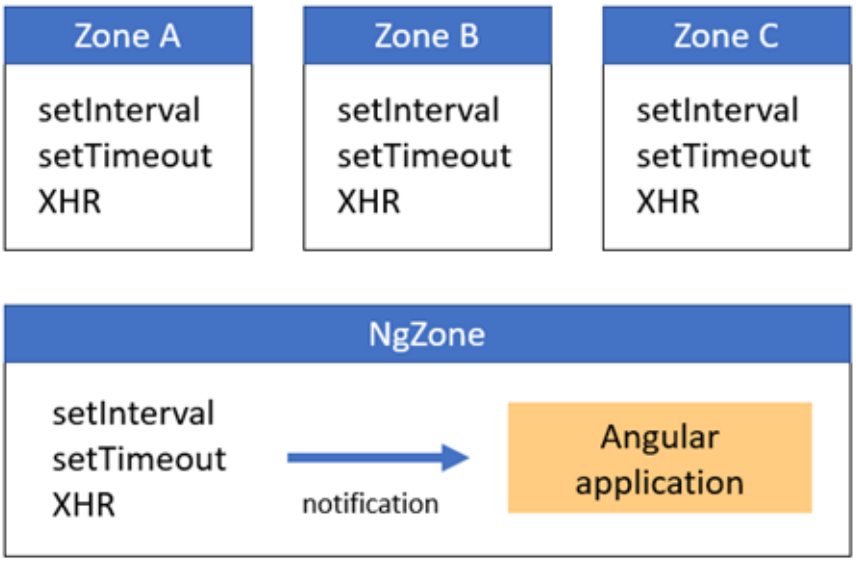
```
export class AppComponent {
    _time;
    get time() {  return this._time; }

    constructor() {
        this._time = Date.now();

        setInterval(() => {
            this._time = Date.now();
```

This implementation solves our original problem. But, unfortunately, it introduces a new one. All timing events, like `setInterval`, trigger change detection in Angular. That means that with this implementation we would end up in an infinite loop of change detection cycles. **To avoid that, we need a way to run `setInterval` and not trigger change detection.** Luckily for us, there's a way to do that. To learn how to do that, we need to understand why `setInterval` triggers change detection in Angular in the first place.

Automatic change detection with zones

As opposed to React, change detection in Angular can be triggered completely automatically as a result of any async event in a browser. This is made possible by using the library called `zone.js` that introduces a concept of zones. Contrary to popular belief, zones are not part of the change detection mechanism in Angular. In fact, Angular can work without them. The library simply provides a way to intercept async events, like `setInterval`, and notify Angular about them. Based on that notification, Angular runs change detection.

What's interesting is that you can have many different zones on a web page. One of them is going to be `NgZone`. It's created when Angular bootstraps. This is the zone that the Angular application runs in. And Angular **only** gets notifications about events that occur inside this zone.



But, `zone.js` also provides an API to run some code in a zone other than the Angular zone. Angular is not notified about async events happening in other zones. And no notification means no change detection. The method name to do this is called `runOutsideAngular` and it's implemented by the `NgZone` service.

Here's the implementation which injects `NgZone` and runs `setInterval` outside of the Angular zone:

```
export class AppComponent {
    _time;
    get time() {
        return this._time;
    }

    constructor(zone: NgZone) {
        this._time = Date.now();

        zone.runOutsideAngular(() => {
            setInterval(() => {
```

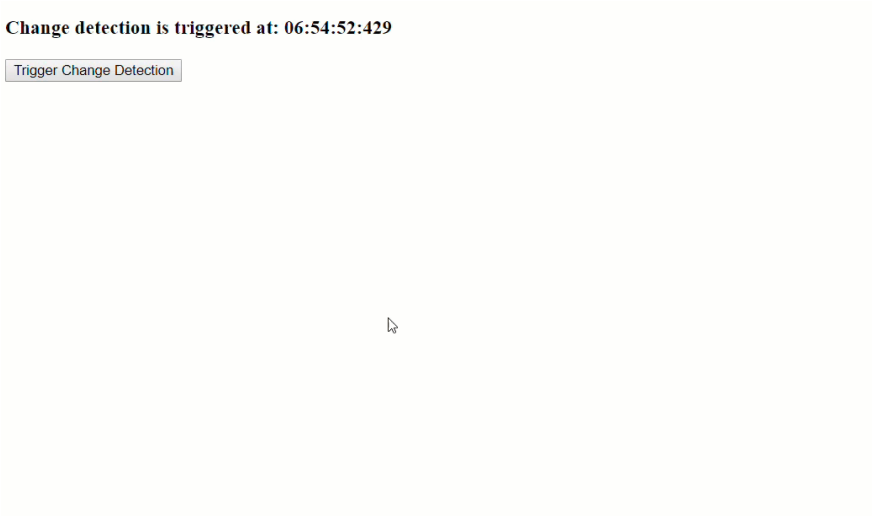
Now we're constantly updating the time, **but we're doing so asynchronously and outside of the Angular zone. This guarantees that during change detection and the following check the getter `time` returns the same value. And when Angular reads the `time` value during the next change detection cycle, the value will be updated and the changes will be reflected on the screen.**

> *Using NgZone to run some code outside of Angular to avoid triggering change detection is a common optimization technique.*

Debugging

You might be wondering if there's any way to see this view and the bindings inside Angular. In fact, there is. There's a function named `checkAndUpdateView` inside the `@angular/core` module. It runs over each view (component) in a tree of components and performs the check for each view. This is the function I always start debugging when I have problems with change detection.

Try debugging it for yourself. Go to this stackblitz demo app and open the console. Find the function and put a breakpoint there. Click on the button to trigger change detection. Inspect the `view` variable. Here's a recording of me doing this:



The first `view` is going to be the host view. It's sort of a root component created by Angular to host our app component. We need to resume execution to get to its child view which is going to be the view created for our `AppComponent`. **Explore it**. The `component` property holds a reference to the `App` component instance. The `nodes` property holds a reference to the DOM nodes created for the elements inside the `App` component's template. The `oldValues` array stores the results of the binding expressions.

• • •

Order of operations

We've just learned that because of the unidirectional data flow restriction you can't change some properties of a component during change detection after this component has been checked. Most often, this update happens through a shared service or synchronous event broadcasting when Angular runs change detection for child components. But it's also possible to directly inject a parent component into a child component and update the parent state in a lifecycle hook. Here is some code that demonstrates this:

```
@Component({
    selector: 'my-app',
    template: `
        <div [textContent]="text"></div>
        <child-comp></child-comp>
    `
})
export class AppComponent {
    text = 'Original text in parent component';
}

@Component({
    selector: 'child-comp',
    template: `<span>I am child component</span>`
})
```

You can play with it here. Basically, we're defining a simple hierarchy of two components. The parent component declares the `text` property that is used in the binding. The child component injects the parent component into the constructor and updates its property in the `ngAfterViewChecked` lifecycle hook. Can you guess what we're going to see in the console? 😃

Right, the familiar `ExpressionChangedAfterItWasChecked` error. And that's because when Angular calls the `ngAfterViewChecked` lifecycle hook for the child component, it already checked the binding for the parent `App` component. But we're updating the parent's property `text` used in the binding after the check.

But here's the interesting part. What if I now change the hook? Say, to `ngOnInit`. Do you think we're still going to see the error?

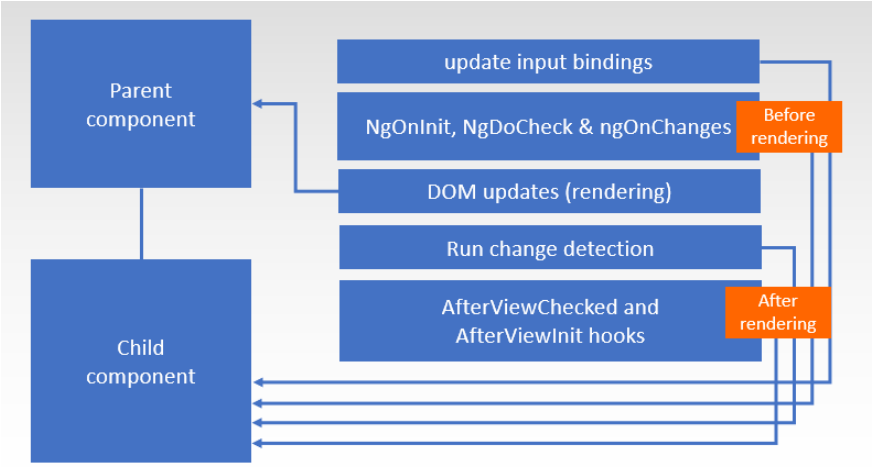
```
export class ChildComponent {
    constructor(private parent: AppComponent) {}

    ngOnInit() {
        this.parent.text = 'Updated text in parent comp
    }
```

Well, this time it's not there. Check the demo. In fact, we can put the code in any other hook (except for `AfterViewInit` and `AfterViewChecked`) and we won't see the error in the console. So what's going on here? Why is the `ngAfterViewChecked` hook special?

To understand this behavior, we need to know what operations Angular performs during change detection and their order. And, we already know where we can find them: the `checkAndUpdateView` function that I showed you previously. Here's part of the code you can find in the function body:

```
function checkAndUpdateView(view, ...) {
    ...
    // update input bindings on child views (component
    // call NgOnInit, NgDoCheck and ngOnChanges hooks
    Services.updateDirectives(view, CheckType.CheckAnd

    // DOM updates, perform rendering for the current
    Services.updateRenderer(view, CheckType.CheckAndUp

    // run change detection on child views (components
    execComponentViewsAction(view, ViewAction.CheckAnd
```

As you can see, Angular also triggers lifecycle hooks as part of change detection. **What's interesting is that some hooks are called before the rendering part when Angular processes bindings and some are called after that.** Here's a diagram that demonstrates what happens when Angular runs change detection for the parent component:



Let's go through it step by step. First, it updates the input bindings for the **child** component. Then it calls the `OnInit`, `DoCheck`, and `OnChanges` hooks, again, on the **child** component. It makes sense because it just updated the input bindings and Angular needs to notify the child components that the input bindings have been

initialized. **Then Angular performs rendering for the current component.** And after that, it runs change detection for the child component. This means that it'll basically repeat these operations on the child view. And finally, it calls the `AfterViewChecked` and `AfterViewInit` hooks on the **child** component to let it know that it's been checked.

What we can notice here is that Angular **calls the** `AfterViewChecked` lifecycle hook for the child component **after** it's processed the bindings of the parent component. On the other hand, the `OnInit` **hook is called before** the bindings are processed. So even if there's a change on the `text` value in the `OnInit`, it's still going to be the same during the following check. And that explains the seemingly weird behavior of not having the error with the `ngOnInit` hook. Mystery solved 🤓.

Summary

Okay, let's now summarize what we've just learned. All components in Angular internally are represented in a data structure known as the view. Angular's compiler parses a template and creates the bindings. Each binding defines a property of a DOM element to update and the expression used to obtain the value. The previous values used for comparison during change detection are stored on a view in the `oldValues` property. During change detection Angular runs over the bindings, evaluates expressions, compares them to the previous values and updates the DOM if necessary. After each change detection cycle, Angular runs a check to ensure the component state is in sync with user interface. This check is performed synchronously and may throw the `ExpressionChangedAfterItWasChecked` error.

We've learned a lot!

I hope that this knowledge will awaken your curiosity. I want to inspire you to learn more about the web platform, architecture, and programming. I want you to become an extraordinary engineer.

Where do you go from here?

These 5 articles will make you an Angular Change Detection expert

If you're looking for a more in-depth explanation of change detection in Angular, this article is a good starting point. It's a collection of in-depth articles about change detection topics like zones, the mechanics of DOM updates, unidirectional data flow and the `ExpressionChangedAfterItWasChecked` error.

Level Up Your Reverse Engineering Skills

Most of the stuff I shared with you today I discovered through reverse-engineering. To me, it's the most rewarding way learn, yet it's probably the most challenging. This article summarizes my reverse-engineering experience and outlines some guidelines and principles that will help you start exploring sources on your own.

Stay tuned for more insights and follow me on Twitter and on Medium.

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.