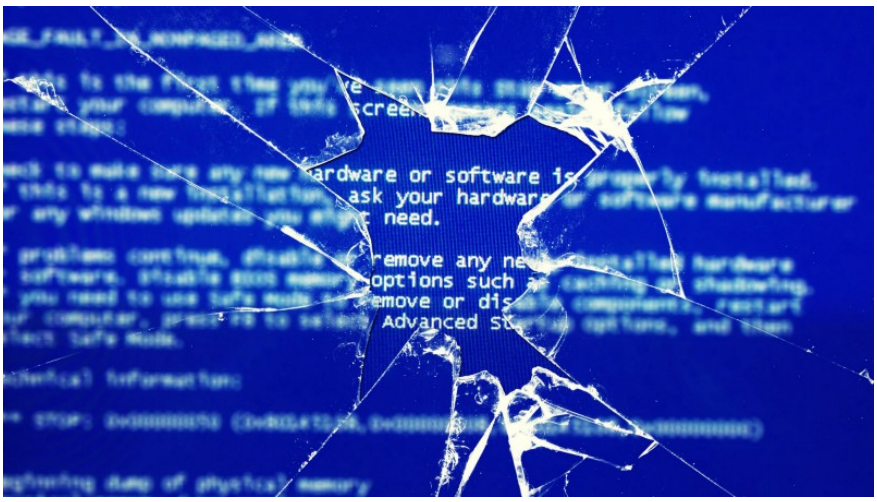
Everything you need to know about the `ExpressionChangedAfterItHasBeenCheckedError` error

Max Koretskyi aka Wizard Follow
Jul 2, 2017 · 10 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.

It seems that recently almost every day there's a question on stackoverflow regarding the `ExpressionChangedAfterItHasBeenCheckedError` error thrown by Angular. Usually these questions come up because Angular developers do not understand how change detection works and why the check that produces this error is required. Many developers even view it as a bug. But it's certainly not. This is a cautionary mechanism

put in place to prevent inconsistencies between model data and UI so that erroneous or old data are not shown to a user on the page.

This article explains the underlying causes of the error and the mechanism by which it's detected, provides a few common patterns that may lead to the error and suggests a few possible fixes. The last chapter provides an explanation of why this check is important.

It seems that the more links to the sources I put in the article the less likely people are to recommend it 😃 . That's why there will be no reference to the sources in this article.

> 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!**

• • •

Relevant change detection operations

A running Angular application is a tree of components. During change detection Angular performs checks for each component which consists of the following operations performed in the specified order:

- update bound properties for all child components/directives
- call `ngOnInit` , `OnChanges` and `ngDoCheck` lifecycle hooks on all child components/directives
- update DOM for the current component
- run change detection for a child component
- call `ngAfterViewInit` lifecycle hook for all child components/directives

There are other operations that are performed during change detection and I've presented them all in the Everything you need to know about change detection in Angular.

After each operation Angular remembers what values it used to perform an operation. They are stored in the `oldValues` property of

the component view. After the checks have been done for all components Angular then starts the next digest cycle but instead of performing the operations listed above it compares the current values with the ones it remembers from the previous digest cycle:

- check that values passed down to the child components are the same as the values that would be used to update properties of these components now
- check that values used to update the DOM elements are the same as the values that would be used to update these elements now
- perform the same checks for all child components

Please note that this additional check is only performed in the development mode. I'll explain why in the last section of the article.

Let's see an example. Suppose you have a parent component `A` and a child component `B`. The `A` component has a `name` and `text` properties. In its template it uses the expression that references `name` property:

```
template: '<span>{{name}}</span>'
```

And it also has `B` component in its template and passes the `text` property to this component through input property binding:

```
@Component({
    selector: 'a-comp',
    template: `
        <span>{{name}}</span>
        <b-comp [text]="text"></b-comp>
    `
})
export class AComponent {
    name = 'I am A component';
    text = 'A message for the child component`;
```

So here is what happens when Angular runs change detection. It starts by checking `A` component. The first operation in the list is to update bindings so it evaluates `text` expression to `A message for`

`the child component` and passes it down to the `B` component. It also stores this value on the view:

```
view.oldValues[0] = 'A message for the child component';
```

Then it calls the lifecycle hooks mentioned in the list.

Now, it performs the third operation and evaluates the expression `{{name}}` to the text `I am A component`. It updates the DOM with this value and puts the evaluated value to the `oldValues`:

```
view.oldValues[1] = 'I am A component';
```

Then Angular performs the next operation and runs the same check for the child `B` component. Once the `B` component is checked the current digest loop is finished.

If Angular is running in the development mode it then runs the second digest performing verification operations I listed above. Now imagine that somehow the property `text` was updated on the `A` component to the `updated text` after Angular passed the value `A message for the child component` to the `B` component and stored it. So it now runs the verification digest and the first operation is to check that the property `text` is not changed:

```
AComponentView.instance.text === view.oldValues[0]; // false
'A message for the child component' === 'updated text'; //
false
```

Yet it has and so Angular throws the error `ExpressionChangedAfterItHasBeenCheckedError`.

The same holds for the third operation. If the `name` property was updated after it was rendered in the DOM and stored we'll get the same error:

```
AComponentView.instance.name === view.oldValues[1]; // false
'I am A component' === 'updated name'; // false
```

You probably have a question in your mind now how is it possible that these values changed. Let's see that.

• • •

Causes of values change

The culprit is always the child component or a directive. Let's have a quick simple demonstration. I will use the simplest example as possible but I will then show real-world scenarios after that. You probably know that child components and directives can inject their parent components. So let's have our `B` component inject parent `A` component and update the bound property `text`. We will update the property in the `ngOnInit` lifecycle hook as it is triggered after the bindings have been processed which is shown here:

```
export class BComponent {
    @Input() text;

    constructor(private parent: AppComponent) {}

    ngOnInit() {
        this.parent.text = 'updated text';
    }
}
```

And as expected we get the error:

```
Error: ExpressionChangedAfterItHasBeenCheckedError:
Expression has changed after it was checked. Previous value:
'A message for the child component'. Current value: 'updated
text'.
```

Now, let's do the same for the property `name` that is used in the template expression of the parent `A` component:

```
ngOnInit() {
    this.parent.name = 'updated name';
}
```

And now everything works OK. How come?

Well if you take an attentive look at the operations order you will see that the `ngOnInit` lifecycle hook is triggered before the DOM update operation. That's why there's no error. We need a hook that is called after the DOM update operations and the `ngAfterViewInit` is a good candidate:

```
export class BComponent {
    @Input() text;

    constructor(private parent: AppComponent) {}

    ngAfterViewInit() {
        this.parent.name = 'updated name';
    }
}
```

And this time we get the expected error:

```
AppComponent.ngfactory.js:8 ERROR Error:
ExpressionChangedAfterItHasBeenCheckedError: Expression has
changed after it was checked. Previous value: 'I am A
component'. Current value: 'updated name'.
```

Of course, real world examples are much more intricate and complex. The parent component properties update or operations that cause DOM rendering are usually done indirectly by using services or observables. But the root cause is always the same.

Now let's see some real-world common patterns that lead to the error.

Shared service

This pattern is illustrated by this plunker. The application is designed to have a service shared between a parent and a child component. A child component sets a value to the service which in turn reacts by

updating the property on the parent component. I call this parent property update indirect because unlike in our example above it isn't evident right away that a child component updates a parent component property.

Synchronous event broadcasting

This pattern is illustrated by this plunker. The application is designed to have a child component emitting an event and a parent component listening to this event. The event causes some of the parent properties to be updated. And these properties are used as input binding for the child component. This is also an indirect parent property update.

Dynamic component instantiation

This pattern is different because unlike the previous ones where input bindings were affected this pattern causes DOM update operation to throw the error. This pattern is illustrated by this plunker. The application is designed to have a parent component that dynamically adds a child component in the `ngAfterViewInit`. Since adding a child component requires DOM modification and `ngAfterViewInit` lifecycle hook is triggered after the Angular updated DOM the error is thrown.

. . .

Possible fixes

If you take a look at the error description the last statement says the following:

> *Expression has changed after it was checked. Previous value:... Has it been created **in a change detection hook** ?*

Often, the fix is to use the right change detection hook to create a dynamic component. For example, the last example in the previous section with dynamic components can be fixed by moving the component creation to the `ngOnInit` hook. Although the documentation states the `ViewChild` is available only after `ngAfterViewInit`, it populates the children when creating a view and so they are available earlier.

If you google around you will probably find two most common fixes for this error—asynchronous property update and forcing additional change detection cycle. Although I show them here and explain why

they work I don't recommend using them but rather redesign your application. I explain why in the last chapter.

Asynchronous update

One thing to notice here is that both change detection and verification digests are performed synchronously. It means that if we update properties asynchronously the values will not be updated when the verification loop is running and we should get no error. Let's test it:

```
export class BComponent {
    name = 'I am B component';
    @Input() text;

    constructor(private parent: AppComponent) {}

    ngOnInit() {
        setTimeout(() => {
            this.parent.text = 'updated text';
        });
    }

    ngAfterViewInit() {
        setTimeout(() => {
            this.parent.name = 'updated name';
        });
    }
}
```

Indeed, no error is thrown. The `setTimeout` function schedules a macrotask then will be executed in the following VM turn. It is also possible to executed the update in the current VM turn but after the current synchronous code has finished executing by using `then` callback of a promise:

```
Promise.resolve(null).then(() => this.parent.name = 'updated
name');
```

Instead of a macrotask `Promise.then` creates a microtask. The microtask queue is processed after the current synchronous code has finished executing hence the update to the property will happen after the verification step. To learn more about micro and macro tasks in Angular you can read I reverse-engineered Zones (zone.js) and here is what I've found.

If you're using `EventEmitter` you can pass `true` option to make asynchronous:

```
new EventEmitter(true);
```

Forcing change detection

The other possible solution is to force another change detection cycle for the parent `A` component between the first one and the verification phase. And the best place to do it is inside the `ngAfterViewInit` lifecycle hook as it's triggered when change detection for all child components have been performed and so they all had possibility to update parent components property:

```
export class AppComponent {
    name = 'I am A component';
    text = 'A message for the child component';

    constructor(private cd: ChangeDetectorRef) {
    }

    ngAfterViewInit() {
        this.cd.detectChanges();
    }
```

Well, no error. So it seems to be working but there is a problem with this solution. When we trigger change detection for the parent `A` component, Angular will run change detection for all child components as well and so there's a possibility of parent property update.

. . .

Why do we need verification loop

Angular enforces so-called **unidirectional data flow from top to bottom**. No component lower in hierarchy is allowed to update properties of a parent component **after parent changes have been processed**. This ensures that after the first digest loop the entire tree of components is stable. A tree is unstable if there are changes in the properties that need to be synchronized with the consumers that depend on those properties. In our case a child `B`

component depends on the parent `text` property. Whenever these properties change the component tree becomes unstable until this change is delivered to the child `B` component. The same holds for the DOM. It is a consumer of some properties on the component and it renders them on UI. If some properties are not synchronized a user will see incorrect information on the page.

This data synchronization process is what happens during change detection—particularly those two operations I listed in the beginning. So what happens if you update the parent properties from the child component properties after the synchronization operation has been performed? Right, you're left with the unstable tree and the consequences of such state is not possible to predict. **Most of the times you will end up with an incorrect information shown on the page to the user.** And this will be very difficult to debug.

So why not run the change detection until the components tree stabilizes? The answer is simple—because it may never stabilize and run forever. If a child component updates a property on the parent component as a reaction to this property change you will get an infinite loop. Of course, as I said earlier it's trivial to spot such a pattern with the direct update or dependency but in the real application both update and dependency are usually indirect.

Interestingly, AngularJS didn't have a **unidirectional data flow** so it tried to stabilize the tree. But it often resulted in the infamous `10 $digest() iterations reached. Aborting!` error. Go ahead and google this error and you'll be surprised by the amount of questions that this error produced.

The last question you may have is why run this only during development mode? I guess this is because an unstable model is not as dramatic problem as a runtime error produced by the framework. After all it may stabilize in the next digest run. However, it's better to be notified of the possible error when developing an application than debug a running application on the client side.

. . .

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.