The difference between NgDoCheck and AsyncPipe in OnPush components

Max Koretskyi aka Wizard Follow
Oct 25, 2018 · 7 min read

An in-depth guide into manual control of change detection in Angular

> **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.

This post comes as a response to this tweet by Shai. He asks whether it makes sense to use `NgDoCheck` lifecycle hook to manually compare values instead of using the recommend approach with the `async` pipe. That's a very good question that requires a lot of understanding of how things work under the hood: change detection, pipes and lifecycle hooks. That's where I come in 😎 .

In this article I'm going to show you how to manually work with change detection. These techniques give you a finer control over the comparisons performed automatically by Angular for input bindings

and async values checks. Once we have this knowledge, I'll share with you my thoughts on the performance impact of these solutions.

> 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!**

Let's get started!

OnPush components

In Angular, we have a very common optimization technique that requires adding the `ChangeDetectionStrategy.OnPush` to a component's decorator. Suppose we have a simple hierarchy of two components like this:

```
@Component({
    selector: 'a-comp',
    template: `
        <span>I am A component</span>
        <b-comp></b-comp>
    `
})
export class AComponent {}

@Component({
```

With this setup, Angular runs change detection always for both `A` and `B` components every single time. If we now add the `OnPush` strategy for the `B` component:

```
@Component({
    selector: 'b-comp',
    template: `<span>I am B component</span>`,
    changeDetection: ChangeDetectionStrategy.OnPush
})
```

Angular will run change detection for the `B` component **only if its input bindings have changed**. Since at this point it doesn't have any bindings, the component will ever only be checked once during the bootstrap.

Triggering change detection manually

Is there a way to force change detection on the component `B` ? Yes, we can inject `changeDetectorRef` and use its method `markForCheck` to indicate for Angular that this component needs to be checked. And since the NgDoCheck hook will still be triggered for B component, that's where we should call the method:

```
@Component({
    selector: 'b-comp',
    template: `<span>I am B component</span>`,
    changeDetection: ChangeDetectionStrategy.OnPush
})
export class BComponent {
    constructor(private cd: ChangeDetectorRef) {}

    ngDoCheck() {
```

Now, the component `B` will always be checked when Angular checks the parent `A` component. Let's now see where we can use it.

Input bindings

I told you that Angular only runs change detection for `OnPush` components when bindings change. So let's see the example with input bindings. Suppose we have an object that is passed down from the parent component through the inputs:

```
@Component({
    selector: 'b-comp',
    template: `
        <span>I am B component</span>
        <span>User name: {{user.name}}</span>
    `,
    changeDetection: ChangeDetectionStrategy.OnPush
})
```

In the parent component `A` we define the object and also implement the `changeName` method that updates the name of the object when a button is clicked:

```
@Component({
    selector: 'a-comp',
    template: `
        <span>I am A component</span>
        <button (click)="changeName()">Trigger change
        <b-comp [user]="user"></b-comp>
    `
})
export class AComponent {
    user = {name: 'A'};
```

If you now run this example, after the first change detection you're going to see the user's name printed:

```
User name: A
```

But when we click on the button and change the name in the callback:

```
changeName() {
    this.user.name = 'B';
}
```

the name is **not updated** on the screen. And we know why, that's because Angular performs shallow comparison for the input parameters and the reference to the `user` object hasn't changed. So how can we fix this?

Well, we can manually check the name and trigger change detection when we detect the difference:

```
@Component({
    selector: 'b-comp',
    template: `
        <span>I am B component</span>
        <span>User name: {{user.name}}</span>
    `,
    changeDetection: ChangeDetectionStrategy.OnPush
})
export class BComponent {
    @Input() user;
    previousName = '';

    constructor(private cd: ChangeDetectorRef) {}

```

If you now run this code, you're going to see the name updated on the screen.

Asynchronous updates

Now, let's make our example a bit more complex. We're going to introduce an RxJs based service that emits updates asynchronously. This is similar to what you have in NgRx based architectures. I'm going to use a `BehaviorSubject` as a source of values because I need to start the stream with an initial value:

```
@Component({
    selector: 'a-comp',
    template: `
        <span>I am A component</span>
        <button (click)="changeName()">Trigger change
        <b-comp [user]="user"></b-comp>
    `
})
export class AComponent {
    stream = new BehaviorSubject({name: 'A'});
    user = this.stream.asObservable();
```

So we receive this stream of `user` objects in the child component. We need to subscribe to the stream and check if the values are updated. And the common approach to doing that is to use Async pipe.

Async pipe

So here's the implementation of the child `B` component:

```
@Component({
    selector: 'b-comp',
    template: `
        <span>I am B component</span>
        <span>User name: {{(user | async).name}}</span
    `,
    changeDetection: ChangeDetectionStrategy.OnPush
})
```

Here's the demo. But is there another way that doesn't use the pipe?

Manual check and change detection

Yes, we can check the value manually and trigger change detection if needed. Just as with the examples in the beginning, we can use `NgDoCheck` lifecycle hook for that:

```
@Component({
    selector: 'b-comp',
    template: `
        <span>I am B component</span>
        <span>User name: {{user.name}}</span>
    `,
    changeDetection: ChangeDetectionStrategy.OnPush
})
export class BComponent {
    @Input('user') user$;
    user;
    previousName = '';

    constructor(private cd: ChangeDetectorRef) {}

    ngOnInit() {
        this.user$.subscribe((user) => {
            this.user = user;
        })
```

You can play with it here.

Ideally, though, we would want to move our comparison and update logic from `NgDoCheck` and put it into the subscription callback, because that's when the new value will be available:

```
export class BComponent {
    @Input('user') user$;
    user = {name: null};

    constructor(private cd: ChangeDetectorRef) {}

    ngOnInit() {
        this.user$.subscribe((user) => {
            if (this.user.name !== user.name) {
                this.cd.markForCheck();
                this.user = user;
```

Play with it here.

What's interesting is that it's exactly what the Async pipe is doing under the hood:

```
@Pipe({name: 'async', pure: false})
export class AsyncPipe implements OnDestroy, PipeTrans
  constructor(private _ref: ChangeDetectorRef) {}

  transform(obj: ...): any {
    ...
    this._subscribe(obj);

    ...
    if (this._latestValue === this._latestReturnedValu
      return this._latestReturnedValue;
    }

    this._latestReturnedValue = this._latestValue;
    return WrappedValue.wrap(this._latestValue);
  }

  private _subscribe(obj): void {
    ...
    this._strategy.createSubscription(
```

• • •

So which solution is faster?

So now that we know how we can use manual change detection instead of the async pipe, let's answer the question we started with. Who's faster?

Well, it depends on how you compare them, but with everything else being equal, manual approach is going to be faster. I don't think though that the difference will be tangible. Here are just a few examples why manual approach can be faster.

In terms of memory you don't need to create an instance of a Pipe class. In terms of compilation time the compiler doesn't have to spend time parsing pipe specific syntax and generating pipe specific output. In terms of runtime, you save yourself a couple of function calls for each change detection run on the component with async pipe. Here's for example the code for the updateRenderer function generated for the code with pipe:

```
function (_ck, _v) {
    var _co = _v.component;
    var currVal_0 = jit_unwrapValue_7(_v, 3, 0, asyncpi
    _ck(_v, 3, 0, currVal_0);
```

As you can see, the code for the async pipe calls the `transform` method on the pipe instance to get the new value. The pipe is going to return the latest value it received from the subscription.

Compare it to the plain code generated for the manual approach:

```
function(_ck,_v) {
    var _co = _v.component;
    var currVal_0 = _co.user.name;
    _ck(_v,3,0,currVal_0);
```

These are the functions executed by Angular when checking `B` component.

A few more interesting things

Unlike input bindings that perform shallow comparison, **the async pipe** implementation **doesn't perform comparison at all** (kudos

to Olena Horal for noticing that). It treats every new emission as an update even if it matches the previously emitted value. Here's the implementation of the parent component `A` that emits the same object. Despite this fact, Angular still runs change detection for the component `B`:

```
export class AComponent {
    o = {name: 'A'};
    user = new BehaviorSubject(this.o);

    changeName() {
        this.user.next(this.o);
```

It means that the component with the async pipe will be **marked for check every time a new value is emitted**. And Angular will check the component next time it runs change detection even if the value hasn't changed.

Where is this relevant? Well, in our case we're only interested in the property `name` from the `user` object because we use it in the template. We don't really care about the whole object and the fact that the reference to the object may change. If the name is the same we don't need to re-render the component. But you can't avoid that with the async pipe.

`NgDoCheck` is not without the problems on its own :) As the hook is only triggered if the parent component is checked, it won't be triggered if one of its parent components uses `OnPush` strategy and is not checked during change detection. So you can't rely on it to trigger change detection when you receive a new value through a service. In this case, the solution I showed with putting `markForCheck` in the subscription callback is the way to go.

Conclusion

Basically, manual comparison gives you more control over the check. You can define when the component needs to be checked. And this is the same as with many other tools—manual control gives you more flexibility, but you have to know what you're doing. And to acquire this knowledge, I encourage you to invest time and effort in learning and reading sources.

If you're concerned with how often `NgDoCheck` lifecycle hook is called or that it's going to be called more often than the pipe's `transform` — don't. First, I showed the solution above where you don't use the hook in the manual approach with asynchronous stream. Second, the hook will only be called when the parent component is checked. If the parent component is not checked, the hook is not called. And with regards to the pipe, because of the shallow check and changing references in the stream, you're going to have the same number of calls or even more with the `transform` method of the pipe.

Want to learn more about change detection in Angular?

Start with These 5 articles will make you an Angular Change Detection expert. This series is a must-read if you want to have a solid grasp of the change detection mechanism in Angular. Each article builds upon the information explained in the preceding one and goes from high-level overview down to implementation details with references to the sources.

For more insights follow me on Twitter and on Medium.

. . .