Angular's $digest is reborn in the newer version of Angular

Max Koretskyi aka Wizard Follow
May 8, 2017 · 9 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.

> *Angular's $digest is gone. Long live the digest!*

I've worked with Angular.js for a few years and despite the widespread criticism I think this is a fantastic framework. I've started with a great book Building your own Angular.js and read most of the framework's source code. So I want to believe I have a solid knowledge of the Angular.js inner workings and a good grasp of the ideas used to built the framework. Now, I'm trying to get to the same level of understanding with the newer Angular and map the ideas between the versions. What I've found is that contrary to what is

usually claimed on the internet Angular borrows many ideas from its predecessor.

One of such ideas is the infamous digest loop:

> *The problem with this is that it is tremendously expensive. Changing anything in the application becomes an operation that triggers hundreds or thousands of functions looking for changes.* ***This is a fundamental part of what Angular is, and it puts a hard limit on the size of the UI you can build in Angular while remaining performant.***

Although with a good understanding of the way angular digest was implemented it was possible to design your application to be quite performant. For example, selectively using `$scope.$digest()` instead of `$scope.$apply` everywhere and embracing immutable objects. But the fact that knowing under the hood implementation is required to be able to design a performant application is probably a show-stopper for many.

So it's no wonder most tutorials on Angular claim there is no more $digest cycle in the framework. This view largely depends on what exactly is meant by the digest, but I think that given its purpose it's a misleading claim. It's still there. Yes, we don't have explicit scopes and watchers and don't call `$scope.$digest()` anymore, but the mechanism to detect changes that traverses the tree of components, calls implicit watchers and updates the DOM has been a given second life in Angular. Completely rewritten. Much enhanced.

This article explores the differences in the implementation of digest between Angular.js and Angular. It will be very beneficial for Angular.js developers migrating to newer Angular as well as for existing Angular developers.

> 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!**

• • •

The need for digest

Before we begin, let's remember why the digest appeared in the angular.js in the first place. Every framework solves the problem of synchronization between a data model (JavaScript objects) and UI (Browser DOM). The biggest challenge in the implementation is to know when a data model changed. The process of checking for the changes is called change detection. And it's implementation is the biggest differentiator between most popular frameworks nowadays. I'm planning to write an in-depth article on change detection mechanisms comparison between existing frameworks. If you'd like to get notified, do follow me :).

There are two principle ways to detect changes—ask user to notify a framework or detect changes automatically by comparison. Suppose we have the following object:

```
let person = {name: 'Angular'};
```

and we updated the `name` property. How does a framework know it's been updated? One way is to ask a user to notify a framework:

```
constructor() {
    let person = {name: 'Angular'};
    this.state = person;
}
...
// explicitly notifying React about the changes
// and specifying what is about to change
this.setState({name: 'Changed'});
```

or force him to use a wrapper on properties so the framework can add setters:

```
let app = new Vue({
    data: {
        name: 'Hello Vue!'
    }
});
// the setter is triggered so Vue knows what changed
app.name = 'Changed';
```

The other way is to save the previous value for the `name` property and compare it to the current:

```
if (previousValue !== person.name) // change detected,
update DOM
```

But when should the comparison be done? We should run the check every time the code runs. And since we know that code runs as a result of an asynchronous event—so called Virtual Machine (VM) turn/tick, we can run this check in the end of the turn. And this is what Angular.js use digest for. So we can define digest as

> *a change detection mechanism that walks the tree of components, checks each component for changes and updates DOM when a component property is changed*

If we use this definition of digest, I assert that the primary mechanism hasn't changed in the newer Angular. What changed is the implementation of digest.

Angular.js

Angular.js uses a concept of a watcher and a listener. A watcher is a function that returns a value being watched. Most often these values are the properties on a data model. But it doesn't always have to be model properties—we can track component state on the scope, computed values, third-party components etc. If the returned value is different from the previous returned value, angular calls a listener. This listener is usually used to update the UI.

This is reflected in the parameters list of the `$watch` function:

```
$watch(watcher, listener);
```

So, if we have an object `person` with the property `name` used in html as `<span>{{name}}</span>`, we can track this property and update the DOM using the following:

```
$watch(() => {
    return person.name
}, (value) => {
    span.textContent = value
});
```

This is essentially what interpolation and directives like `ng-bind` do. Angular.js uses directives to reflect data model in the DOM. The newer Angular doesn't do that anymore. It uses property mappings to connect data model and DOM. The previous example is now implemented like this:

```
<span [textContent]="person.name"></span>
```

Since we have many components that make up a tree and each component has different data model, we have a hierarchy of watchers that closely resembles the components tree. Watchers are grouped using `$scope`, but it is not really relevant here.

Now, during digest `angular.js` walks this tree of watchers and updates the DOM. Usually this digest cycle is triggered on every asynchronous event if you use existing mechanisms `$timeout`, `$http` or on demand by the means of `$scope.$apply` and `$scope.$digest`.

Watchers are triggered in the strict order—first for parent components and then for child components. This makes sense, but it has some unwelcome implications. A watcher listener can have various side effects, including updating properties on a parent component. If parent listeners have already been processed, and a child listener updates parent properties, the changes will not be detected. That's why the digest loop has to run multiple times to get stable—to ensure that there are no more changes. And the number of such runs are limited to 10. This design decision is now considered flawed and Angular doesn't allow that.

Angular

Angular doesn't have a concept of a watcher similar to Angular.js. But the functions that track model properties do exist. These update functions are now generated by the framework compiler and cannot

be accessed. Also they are now strongly connected to the underlying DOM. These functions are stored in updateRenderer property name on a view.

They are also very specific—they track only model changes instead of a tracking anything in Angular.js. Each component gets one watcher which tracks all component properties used in a template. Instead of returning a value it calls checkAndUpdateTextInline function for each property being tracked. This function compares previous value to the current and updates DOM if changed.

For example, for the `AppComponent` with the following template:

```
<h1>Hello {{model.name}}</h1>
```

The compiler will generate the following code:

```
function View_AppComponent_0(l) {
    // jit_viewDef2 is `viewDef` constructor
    return jit_viewDef2(0,

        // array of nodes generated from the template
        // first node for `h1` element
        // second node is textNode for `Hello
{{model.name}}`
        [
            jit_elementDef3(...),
            jit_textDef4(...)
        ],
        ...

        // updateRenderer function similar to a watcher
        function (ck, v) {
            var co = v.component;

            // gets current value for the component `name`
property
            var currVal_0 = co.model.name;

            // calls CheckAndUpdateNode function passing
            // currentView and node index (1) which uses
            // interpolated `currVal_0` value
            ck(v, 1, 0, currVal_0);
        });
}
```

So even if a watcher is now implemented differently, the digest loop is still there. It altered its name to change detection cycle:

> *In development mode,* `tick()` *also performs a second* ***change detection cycle*** *to ensure that no further changes are detected.*

I mentioned earlier that during digest `angular.js` walks the tree of watchers and updates DOM. The very same thing happens with Angular. During change detection cycle angular walks a tree of components and calls renderer update functions. It's done as part of a checking and updating view process and I've written quite at length about it in Everything you need to know about change detection in Angular.

Just as in Angular.js in the newer Angular this change detection cycle is triggered on every asynchronous event. But since Angular uses zone to patch all asynchronous events, no manual triggering of change detection is required for most of the events. The framework subscribes to onMicrotaskEmpty event and gets notified when an async event is completed. This event is fired when there is no more microtasks enqueued in the current VM Turn. Yet, the change detection can be triggered manually as well using view.detectChanges or ApplicationRef.tick methods .

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**. If a component updates parent model properties in the DoCheck hook, it's fine since this lifecycle hook is called before detecting properties changes. But if the property is updated in some other way, for example, from the AfterViewChecked hook which is called after processing changes, you'll get an error in the development mode:

> *Expression has changed after it was checked*

You can read more about this error in Everything you need to know about the `ExpressionChangedAfterItHasBeenCheckedError` error.

In production mode there will be no error but Angular will not detect these changes until the next change detection cycle.

Using life-cycle hooks to track changes

In angular.js each component defines a set of watchers to track the following:

- parent component bindings
- self component properties
- computed values
- third-party widgets outside Angular ecosystem

Here is how these functions can be implemented in Angular. To track parent component bound properties we can now use OnChanges life cycle hook.

We can use DoCheck life cycle hook to track self-component properties and calculate computed properties. Since this hook is triggered before Angular process properties changes on the current component, we can do whatever we need to get correctly reflected changes in UI.

We can use OnInit hook to watch third-party widgets outside Angular ecosystem and run change detection manually.

For example, we have a component that displays current time. The time is supplied by the `Time` service. Here is how it would have been implemented in Angular.js:

```
function link(scope, element) {
    scope.$watch(() => {
        return Time.getCurrentTime();
    }, (value) => {
        $scope.time = value;
    })
}
```

Here is how you should implement it in Angular:

```
class TimeComponent {
    ngDoCheck()
    {
        this.time = Time.getCurrentTime();
    }
}
```

Another example is if we had a third-part `slider` component not integrated into Angular ecosystem, but we needed to show the current slide, we would simply wrap this component into angular component, track slider's `changed` event and triggered digest manually to reflect changes in UI:

```
function link(scope, element) {
    slider.on('changed', (slide) => {
        scope.slide = slide;

        // detect changes on the current component
        $scope.$digest();

        // or run change detection for the all app
        $rootScope.$digest();
    })
}
```

The idea is the same in Angular. Here is how it can be done:

```
class SliderComponent {
    ngOnInit() {
        slider.on('changed', (slide) => {
            this.slide = slide

            // detect changes on the current component
            // this.cd is an injected ChangeDetector
instance
            this.cd.detectChanges();

            // or run change detection for the all app
            // this.appRef is an ApplicationRef instance
            this.appRef.tick();
        })
    }
}
```

That's all folks!

. . .

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.


3 reasons why you should follow
Angular-In-Depth publication