ANGULAR CHANGE DETECTION EXPLAINED

*by **Pascal Precht** on Feb 22, 2016, last updated on Dec 18, 2016*

NG-NL has happened and it was awesome! I had the honour of giving a talk about change detection in Angular and it seemed to be a huge success as attendees liked it a lot. With this article, we'd like to transform the talk into a written version, so everyone can read about how Angular's change detection works and how to make it faster for our use cases. If you're interested in the talk, you can view the slides here . Now let's take a look at this beast.

TABLE OF CONTENTS

What's Change Detection anyways?

The basic task of change detection is to take the internal state of a program and make it somehow visible to the user interface. This state can be any kind of objects, arrays, primitives, … just any kind of JavaScript data structures.

This state might end up as paragraphs, forms, links or buttons in the user interface and specifically on the web, it's the Document Object Model (DOM). So basically we take data structures as input and generate DOM output to display it to the user. We call this process rendering.

However, it gets trickier when a change happens at runtime. Some time later when the DOM has already been rendered. How do we figure out what has changed in our model, and where do we need to update the DOM? Accessing the DOM tree is always expensive, so not only do we need to find out where updates are needed, but we also want to keep that access as tiny as possible.

This can be tackled in many different ways. One way, for instance, is simply making a http request and re-rendering the whole page. Another approach is the concept of diffing the DOM of the new state with the previous state and only render the difference, which is what ReactJS is doing with **Virtual DOM**.

Tero has written an awesome article on Change and its detection in JavaScript frameworks, we recommend checking it out if you're more interested in how different frameworks solve this issue. In this article we're going to focus on Angular version >= 2.x.

So basically the goal of change detection is always projecting data and its change.

What causes change?

Now that we know what change detection is all about, we might wonder, when exactly can such a change happen? When does Angular know that it has to update the view? Well, let's take a look at the following code:

```
@Component({

  template: `

    <h1>{{firstname}} {{lastname}}</h1>

    <button (click)="changeName()">Change name</button>

  `

})

class MyApp {



  firstname:string = 'Pascal';

  lastname:string = 'Precht';



  changeName() {

    this.firstname = 'Brad';
```

```
    this.lastname = 'Green';

  }

}
```

If this is the first time you're seeing an Angular component, you might want to read our article on building a tabs component.

The component above simply displays two properties and provides a method to change them when the button in the template is clicked. The moment this particular button is clicked is the moment when application state has changed, because it changes the properties of the component. That's the moment we want to update the view.

Here's another one:

```
@Component()

class ContactsApp implements OnInit{



  contacts:Contact[] = [];



  constructor(private http: Http) {}
```

```
  ngOnInit() {

    this.http.get('/contacts')

      .map(res => res.json())

      .subscribe(contacts => this.contacts = contacts);

  }

}
```

This component holds a list of contacts and when it initializes, it performs a http request. Once this request comes back, the list gets updated. Again, at this point, our application state has changed so we will want to update the view.

Basically application state change can be caused by three things:

- **Events** - `click`, `submit`, ...
- **XHR** - Fetching data from a remote server
- **Timers** - `setTimeout()`, `setInterval()`

They are all asynchronous. Which brings us to the conclusion that, basically whenever some asynchronous operation has been performed, our application state might have changed. This is when someone needs to tell Angular to update the view.

Who notifies Angular?

Alright, we now know what causes application state change. But what is it that tells Angular, that at this particular moment, the view has to be updated?

Angular allows us to use native APIs directly. There are no interceptor methods we have to call so Angular gets notified to update the DOM. Is that pure magic?

If you've followed our latest articles, you know that Zones take care of this. In fact, Angular comes with its own zone called `NgZone`, which we've written about in our article Zones in Angular. You might want to read that, too.

The short version is, that somewhere in Angular's source code, there's this thing called `ApplicationRef`, which listens to `NgZones` `onTurnDone` event. Whenever this event is fired, it executes a `tick()` function which essentially performs change detection.

```
// very simplified version of actual source

class ApplicationRef {



  changeDetectorRefs:ChangeDetectorRef[] = [];



  constructor(private zone: NgZone) {

    this.zone.onTurnDone

      .subscribe(() => this.zone.run(() => this.tick());

  }
```

```
  tick() {
    this.changeDetectorRefs
      .forEach((ref) => ref.detectChanges());
  }
}
```

Change Detection

Okay cool, we now know when change detection is triggered, but how is it performed? Well, the first thing we need to notice is that, in Angular, **each component has its own change detector**.

This is a significant fact, since this allows us to control, for each component individually, how and when change detection is performed! More on that later.

Let’s assume that somewhere in our component tree an event is fired, maybe a button has been clicked. What happens next? We just learned that zones execute the given handler and notify Angular when the turn is done, which eventually causes Angular to perform change detection.

Since each component has its own change detector, and an Angular application consists of a component tree, the logical result is that we're having a **change detector tree** too. This tree can also be viewed as a directed graph where data always flows from top to bottom.

The reason why data flows from top to bottom, is because change detection is also always performed from top to bottom for every single component, every single time, starting from the root component. This is awesome, as unidirectional data flow is more predictable than cycles. We always know where the data we use in our views comes from, because it can only result from its component.

Another interesting observation is that change detection gets stable after a single pass. Meaning that, if one of our components causes any additional side effects after the first run during change detection, Angular will throw an error.

Performance

By default, even if we have to check every single component every single time an event happens, Angular is very fast. It can perform hundreds of thousands of checks within a couple of milliseconds. This is mainly due to the fact that **Angular generates VM friendly code**.

What does that mean? Well, when we said that each component has its own change detector, it's not like there's this single generic thing in Angular that takes care of change detection for each individual component.

The reason for that is, that it has to be written in a dynamic way, so it can check every component no matter what its model structure looks like. VMs don't like this sort of dynamic code, because they can't optimize it. It's considered **polymorphic** as the shape of the objects isn't always the same.

Angular creates change detector classes at runtime for each component, which are monomorphic, because they know exactly what the shape of the component's model is. VMs can perfectly optimize this code, which makes it very fast to execute. The

good thing is that we don't have to care about that too much, because Angular does it automatically.

Check out Victor Savkin's talk on Change Detection Reinvented for a deeper explanation on this.

Smarter Change Detection

Again, Angular has to check every component every single time an event happens because… well, maybe the application state has changed. But wouldn't it be great if we could tell Angular to **only** run change detection for the parts of the application that changed their state?

Yes it would, and in fact we can! It turns out there are data structures that give us some guarantees of when something has changed or not
- **Immutables** and **Observables**. If we happen to use these structures or types, and we tell Angular about it, change detection can be much much faster. Okay cool, but how so?

Understanding Mutability

In order to understand why and how e.g. immutable data structures can help, we need to understand what mutability means. Assume we have the following component:

```
@Component({

  template: '<v-card [vData]="vData"></v-card>'

})

class VCardApp {
```

```
  constructor() {
    this.vData = {
      name: 'Christoph Burgdorf',
      email: 'christoph@thoughtram.io'
    }
  }

  changeData() {
    this.vData.name = 'Pascal Precht';
  }
}
```

`VCardApp` uses `<v-card>` as a child component, which has an input property `vData`. We're passing data to that component with `VCardApp`'s own `vData` property. `vData` is an object with two properties. In addition, there's a method `changeData()`, which changes the name of `vData`. No magic going on here.

The important part is that `changeData()` mutates `vData` by changing its `name` property. Even though that property is going to be changed, the `vData` reference itself stays the same.

What happens when change detection is performed, assuming that some event causes `changeData()` to be executed? First, `vData.name` gets changed, and then it's passed to `<v-card>`. `<v-card>`'s change detector now checks if the given `vData` is still the same as before, and yes, it is. The reference hasn't changed. However, the `name` property has changed, so Angular will perform change detection for that object nonetheless.

Because objects are mutable by default in JavaScript (except for primitives), Angular has to be conservative and run change detection every single time for every component when an event happens.

This is where immutable data structures come into play.

Immutable Objects

Immutable objects give us the guarantee that objects can't change. Meaning that, if we use immutable objects and we want to make a change on such an object, we'll always get a new reference with that change, as the original object is immutable.

This pseudo code demonstrates it:

```
var vData = someAPIForImmutables.create({

              name: 'Pascal Precht'

            });
```

```
var vData2 = vData.set('name', 'Christoph Burgdorf');



vData === vData2 // false
```

`someAPIForImmutables` can be any API we want to use for immutable data structures. However, as we can see, we can't simply change the `name` property. We'll get a new object with that particular change and this object has a new reference. Or in short: **If there's a change, we get a new reference**.

Reducing the number of checks

Angular can skip entire change detection subtrees when input properties don't change. We just learned that a "change" means "new reference". If we use immutable objects in our Angular app, all we need to do is tell Angular that a component can skip change detection, if its input hasn't changed.

Let's see how that works by taking a look at `<v-card>`:

```
@Component({

  template: `

    <h2>{{vData.name}}</h2>

    <span>{{vData.email}}</span>

  `

})
```

```
class VCardCmp {

  @Input() vData;

}
```

As we can see, `VCardCmp` only depends on its input properties. Great. We can tell Angular to skip change detection for this component's subtree if none of its inputs changed by setting the change detection strategy to `OnPush` like this:

```
@Component({

  template: `

    <h2>{{vData.name}}</h2>

    <span>{{vData.email}}</span>

  `,

  changeDetection: ChangeDetectionStrategy.OnPush

})

class VCardCmp {

  @Input() vData;

}
```

That's it! Now imagine a bigger component tree. We can skip entire subtrees when immutable objects are used and Angular is informed accordingly.

Jurgen Van De Moere has written an in-depth article on how he made a minesweeper game built with Angular and Immutable.js blazingly fast. Make sure to check that one out.

Observables

As mentioned earlier, Observables also give us some certain guarantees of when a change has happened. Unlike immutable objects, they don't give us new references when a change is made. Instead, they fire events we can subscribe to in order to react to them.

So, if we use Observables and we want to use `OnPush` to skip change detector subtrees, but the reference of these objects will never change, how do we deal with that? It turns out Angular has a **very smart** way to enable paths in the component tree to be checked for certain events, which is exactly what we need in that case.

To understand what that means, let's take a look at this component:

```
@Component({

  template: '{{counter}}',

  changeDetection: ChangeDetectionStrategy.OnPush

})
```

```
class CartBadgeCmp {



  @Input() addItemStream:Observable<any>;

  counter = 0;



  ngOnInit() {

    this.addItemStream.subscribe(() => {

      this.counter++; // application state changed

    })

  }

}
```

Let’s say we build an e-commerce application with a shopping cart. Whenever a user puts a product into the shopping cart, we want a little counter to show up in our UI, so the user can see the amount of products in the cart.

`CartBadgeCmp` does exactly that. It has a `counter` and an input property `addItemStream`, which is a stream of events that gets fired, whenever a product is added to the shopping cart.

We won't go into much detail on how observables work in this article. If you want to learn more about observables, make sure to read our article on taking advantage of Observables in Angular.

In addition, we set the change detection strategy to `OnPush`, so change detection isn't performed all the time, only when the component's input properties change.

However, as mentioned earlier, the reference of `addItemStream` will never change, so change detection is never performed for this component's subtree. This is a problem because the component subscribes to that stream in its `ngOnInit` life cycle hook and increments the counter. This is application state change and we want to have this reflected in our view right?

Here's what our change detector tree might look like (we've set **all** to `OnPush`). No change detection will be performed when an event happens:

How can we inform Angular about this change? How can we tell Angular that change detection **needs** to be performed for this component, even though the entire tree is set to `OnPush`?

No worries, Angular has us covered. As we've learned earlier, change detection is **always**performed from top to bottom. So what we need is a way to detect changes for the entire path of the tree to the component where the change happened. Angular can't know which path it is, but we do.

We can access a component's `ChangeDetectorRef` via dependency injection, which comes with an API called `markForCheck()`. This method does exactly what we need! It marks the path from our component until root to be checked for the next change detection run.

Let's inject it into our component:

```
constructor(private cd: ChangeDetectorRef) {}
```

Then, tell Angular to mark the path from this component until root to be checked:

```
ngOnInit() {

    this.addItemStream.subscribe(() => {

      this.counter++; // application state changed

      this.cd.markForCheck(); // marks path

    })

  }

}
```

Boom, that’s it! Here’s what it looks like after the observable event is fired but before change detection kicked in:

Now, when change detection is performed, it’ll simply go from top to bottom.

How cool is that? Once the change detection run is over, it'll restore the `OnPush` state for the entire tree.

There's more…

In fact there are some more APIs which we haven't covered in this article, but feel free to check out the slides and/or the recording of the talk.

There are also some demos to play with in this repository, which you can run on your local machine.

Hopefully this made it a little bit more clear how using immutable data structures or Observables can make our Angular application even faster.

Credits

I'd like to thank Jurgen Van De Moere for being a **huge** help and support when I was preparing this talk. He spent a lot of time with me discussing my understandings and raised a lot of good questions that helped me put this content together. He also made sure that the demos look as nice as they do. His CSS skills are amazing - Jurgen, thank you so so much for being such a supportive and nice person.

I'd also like to thank Victor Savkin for answering a lot of my questions regarding change detection in Angular, plus all the very informative articles that he's written - Thanks Victor!