Change Detection and Zones

CS569 – Web Application Development II

Maharishi International University

Department of Computer Science

Associate Professor Asaad Saad

# Maharishi International University - Fairfield, Iowa

JS References

```
let a = { n: 1 }
let b = a;

a.n = 2;
console.log(b === a);  // true

a = { n: 2 };
console.log(b === a);  // false
```

Data Binding

The basic task of data binding 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.
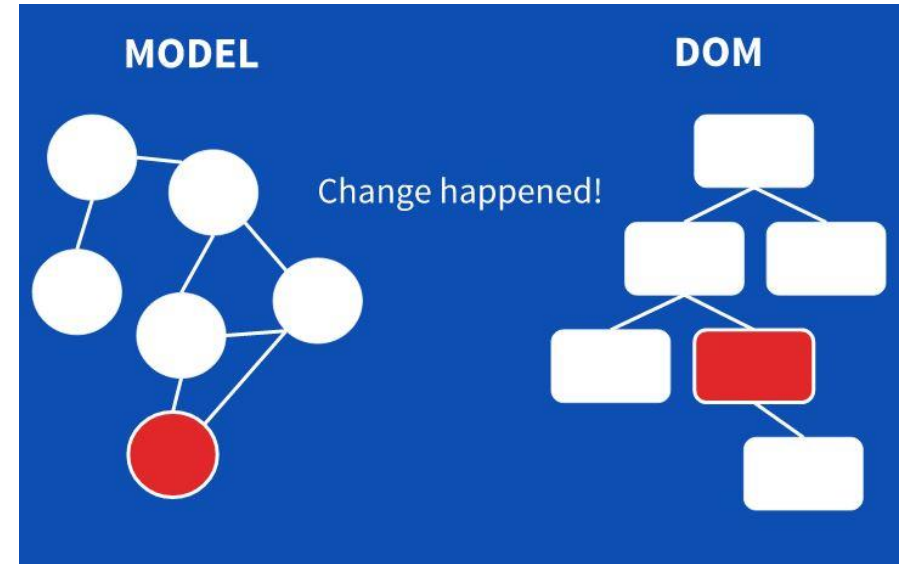
This state might end up as paragraphs, forms, links or buttons in the user interface (DOM). So basically we take data structures as input and generate DOM output to display it to the user. We call this process rendering.

Bindings

Property bindings, which can be added using the [] syntax, it reflects the state of the model in the view.

Event bindings, which can be added using the () syntax, it captures a browser event or component output to execute some function on a component or a directive.

Template bindings, which can be added using the {{}} syntax, it reflects the state of the model in the view.

One-way vs Two-ways Data Binding

In **AngularJS**, the default data binding method was **two-way data binding**. The problem with two-way data binding is that it often causes cascading effects throughout your application and makes it really difficult to trace data flow as your project grows.

One way data binding: we run CD one time only then it gets stable
Two-ways data binding: we must run CD many times UNTIL it gets stable

Change Detection

Let's define the application model that will store the state of our application.

Imagine an event changing the model: changing the teacher or number of students.

```
{
  "course": {"name": "MWA"},
  "details":
         { "id":"CS572",
           "teacher": "Asaad Saad",
           "block": "July",
           "students": 25
         }
}
```

At this point nothing has changed in the DOM. Only the Model has been updated. At the end of the VM turn, magically, the **change detection** kicks in to propagate changes in the DOM.

Change detection goes through every component in the component tree to check if the model it depends on changed. Because from JS point of view , if this object was used as an input in another component, then there is **no change has happened to the object**! (same reference).

One-way Data Binding in Two Phases

An Angular application consists of nested components, so it will always have a component tree.

Angular separates updating the application model and reflecting the state of the model in the view into two distinct phases.

1. The developer is responsible for updating the application model.
2. Angular via bindings (observables), by means of **change detection**, is responsible for reflecting the state of the model in the DOM.

Why Using Two Phases in One-way Data Binding?

Using change detection only for updating the view state limits the number of places where the application model can be changed.

The major benefit of the separation is that it allows us to limit the options of who's contributing in updating the view state propagation process. This makes the system **more predictable**, and it also makes it a lot more performant.

The fact that the change detection graph in Angular can be modeled as a tree allowed us to get rid of digest loop (multiple digest runs until no changes occur). Now the system should get stable after a single pass (from top to bottom).

What has changed? Where?

It gets trickier when a change happens at runtime 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 very 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.

One way to solve this, is simply making http request and **re-rendering the whole page**.

React.js Virtual DOM

The Virtual DOM is a collection of modules designed to provide a declarative way of representing the DOM for your app. So instead of updating the DOM when your application state changes, you simply create a virtual tree, which looks like the DOM state that you want.

The Virtual DOM allows you to update a view **whenever state changes** by **creating a full Virtual Tree of the view** and then **patching the DOM** efficiently to look exactly as you described it. This results in keeping manual DOM manipulation and previous state tracking out of your application code, promoting clean and maintainable rendering logic for web applications.

Angular Differs

In order to evaluate what changed, Angular provides *differs*. Differs will evaluate a given property of your directive to determine what changed.

There are two types of built-in differs:

Iterable differs

Iterable differs is used when we have a **list-like structure** and we're only interested on knowing things that were added or removed from that list.

Key-value differs

Key-value differs is used for **dictionary-like structures**, and it works at the key level. This differ will identify changes when a new key is added, removed and changed.

Who Triggers a Change Detection Cycle

We want to display the value we enter on the **input** element in real-time.

We added an event handler to be triggered on every **keyup**. When any event happens in any component, Angular is going to check all bounded variables (**box**) and start a **Detection Change Cycle** for the component.

```
<input #box (keyup)='0' /> {{ box.value }}
```

What Causes the Change?

Basically application state change can be caused by three things:

    Events - click, submit, …

    XHR - Fetching data from a remote server

    Timers - setTimeout(), setInterval()

Notice how they are all **asynchronous**. Which brings us to the conclusion that whenever an asynchronous operation has been performed, our application state will be changed. **This is when someone needs to tell Angular to update the view (Zones).**

All async operations will run in an execution context that refers to the global scope, so how to know in which component the change has happened?

NgZone

Your code runs in Zone, which is simply an execution context for **async operations**. Angular **Monkey-Patch** all async code and add hooks around it.

```
a();
setTimeout(b, 0);
c();
```

➡
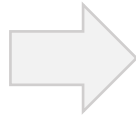
```
a
c
b // async
```

What if we wanted to time how long this runs? This won't work!

```
start()
a();
setTimeout(b, 0);
c();
stop();
```

➡

```
start
a
c
stop
b // missed
```

How Zone works?

All async tasks are **Monkey-Patched** and they run in the **same Zone**.

```
Zone.run(function(){
    a();
    setTimeout(b, 0);
    c();
});
```

Monkey-Patch →

```
var start, time;
function onZoneEnter(){
    start = Date.now();
}
function onZoneLeave(){
    time += (Date.now() - start);
}
```

```
Function a(){
    onZoneEnter();
    //code for function a
    onZoneLeave();
};
```

```
Function b(){
    onZoneEnter();
    //code for function b
    onZoneLeave();
};
```

```
Function c(){
    onZoneEnter();
    //code for function c
    onZoneLeave();
};
```

Turn Zones Off

We can disable ng-Zone for the entire application:

```
platformBrowserDynamic().bootstrapModule(AppModule, {
  ngZone: 'noop'
});
```

Trigger Change Detection Manually:

```
constructor(private cd: ChangeDetectorRef) {}
triggerChangeDetection() {
      this.cd.detectChanges();
}
```

Zones notifies Angular about Changes

Let's assume that somewhere in our component tree an event is fired, maybe a button has been clicked. **Zones** execute the given handler and knows to which component it belongs because it's monkey-patched and notify Angular when the turn is done, which eventually causes Angular to perform a **change detection cycle**.

In React, we call setstate() to inform React that a change has happened. In angular, Zones notifies Angular about a change + where the change has happened.

Change Detection Cycle Algorithm

When one of the components change, no matter where in the tree it is, a **change detection pass is triggered for the whole tree**.

Angular scans for changes from the top component node, all the way to the bottom leaves of the tree.

Each component has its own change detector.

TOP TO BOTTOM
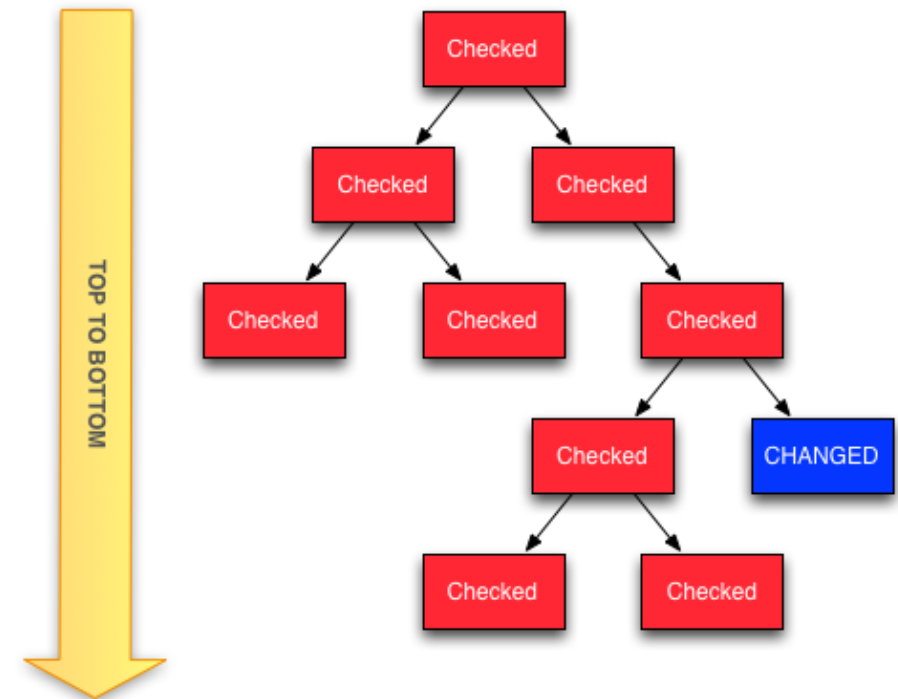
Zones inform Angular that change has happened in this component

Example

```
import { Component, Input, OnChanges } from '@angular/core';
@Component({
    selector: 'app-child',
    template:`<p>Child Component: {{person.name}} lives at {{person.zipcode}}</p>`,
})
export class ChildComponent implements OnChanges {
    @Input() person: { name: string, zipcode: string };

    ngOnChanges(change) {
        console.log(`Change detected`)
    }
}
```

What happens when the App component (parent) changes **zipcode** by mutation vs without mutation?

Angular Change Detection Facts

- Change Detection Graph is a **directed tree.** It's performed in the same order starting from Root component.
- Way more predictable (Data flows from top to bottom)
- **Gets stable after a single pass**

Angular can perform hundreds of thousands of checks in milliseconds. Because **Angular generates VM friendly Monomorphic code**.

(component objects are **monomorphic**: Every components is a class with same properties type)

Monomorphic Type

Monomorphic use of operations is preferred over polymorphic operations. Operations are monomorphic if the hidden classes of inputs are always of the same type- otherwise they are **Polymorphic**

```
function add(x, y) {
        return x + y;
}
add(1, 2); // + in add is monomorphic
add("a", "b"); // + in add becomes polymorphic
```

The compiler will run Monomorphic functions much faster than polymorphic code.

Heuristic Detection

VMs do **heuristic detection** of "**hot functions**" (code that is executed hundreds or even thousands of times). If a function's execution count exceeds a predetermined limit, the VMs optimizer will pick up that bit of code and attempt to compile an optimized version based on the arguments passed to the function. In this case, it presumes your function will always be called with the **same *type* of arguments** (not necessarily the *same* objects).

On the other hand, if your code has to be written in a dynamic way (**polymorphic**: the shape of the objects isn't always the same), VMs will check every component no matter what its model structure looks like.

VMs don't like this sort of dynamic code, because they cannot optimize it.

Angular and Change Detection

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 because Angular does it automatically.

Smarter Change Detection?

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? Rather than running for the whole CD tree. There are data structures that give us some guarantees of when something has changed or not: **Immutables** and **Observables**.

OnPush Strategy

When a component depends only on its input and this input was an immutable object, all we need to do is tell Angular that this component may **skip change detection if its input hasn't changed**.

```
@Component({
        changeDetection: ChangeDetectionStrategy.OnPush
})
```

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.

You must use **Immutables** or **Observables** to use it.

We can skip entire components subtrees when immutable objects are used and Angular is informed accordingly.

Better Solution

```
import { Component, Input, ChangeDetectionStrategy, OnChanges } from '@angular/core';
@Component({
    selector: 'app-child',
    template: ` <p>Child Component: {{person.name}} lives in {{address}} </p> `,
    changeDetection: ChangeDetectionStrategy.OnPush
})
export class ChildComponent implements OnChanges {
    @Input() person: { name: string, zipcode: number };
    address: String;
    constructor() { console.info({ 'Constructor Input Value': this.person }) }
    ngOnInit() {
        console.info({ 'ngOnInit Input Value': this.person })
        this.address = this.locateAddress(this.person.zipcode);
    }

    ngOnChanges(change) {
        console.log(`Change detected, `)
        this.address = this.locateAddress(this.person.zipcode);
    }
}
```

What if...

- Change Detection starts
- Check **A component**
- Evaluate the expression **{{name}}** to the text **Asaad Saad**
- Update the DOM with this value
- Save the evaluated value in **oldValues:** view.oldValues[1] = 'Asaad Saad';
- Evaluate **address** expression to **Fairfield** and pass it down to the **B component**
- Save this value in **oldValues:** view.oldValues[0] = 'Fairfield';
- Run the same check for **B component** and call its lifecycle hooks
- Once the **B component** is checked, the current digest loop is finished.

```
@Component({
    selector: 'a-comp',
    template: `
        <span>{{name}}</span>
        <b-comp [address]="address"></b-comp>`
})
export class AComponent {
    name = 'Asaad Saad';
    address = 'Fairfield';
}
```

```
@Component({
    selector: 'b-comp',
    template: `
        <span>{{address}}</span> `
})
export class BComponent {
    @Input() address;
    ngAfterViewChecked() {
        this.address = 'Burlington';
    }
}
```

What if...

If Angular is running in the **development mode** it then runs a second digest loop performing verification phase.

Angular runs a verification digest to check that properties value have not changed:

```
ACompView.instance.text === view.oldValues[0]; // false
            'Burlington' === 'Fairfield'
```

```
ERROR Error:                          BcomponentComponent.html:2
  ExpressionChangedAfterItHasBeenCheckedError: Expression has
  changed after it was checked. Previous value: 'null:
  Fairfield'. Current value: 'null: Burlington'.
```


ExpressionChangedAfterItHasBeenCheckedError

Why Do We Need Verification Phase

Angular enforces 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.

So why not run the change detection until the components tree stabilizes? 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.
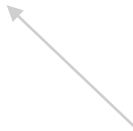
Possible Fixes

Asynchronous property update (both change detection and verification digests are performed synchronously)

```
ngAfterViewChecked() {
        setTimeout(() => { this.parent.name = 'Burlington'; });
}
```

Forcing additional change detection cycle

```
constructor(private cd: ChangeDetectorRef) {}
ngAfterViewChecked() {
        this.cd.detectChanges();
}
```

The setTimeout function schedules a callback that will be executed in the next VM turn.

Main Points

When somewhere in our component tree an event is fired. Zones execute the given handler and notify Angular, which eventually causes Angular to perform change detection.

Each component has its own change detector, and an Angular application consists of a component tree, so we have change detector tree too. This tree can also be viewed as a directed graph where data always flows from top to bottom.

Unidirectional data flow is more predictable than cycles. We always know where the data we use in our views comes from.

Change detection gets stable after a single pass. If one of our components causes any additional change after the first run and during change detection, Angular will throw an error.

Main Points

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.

Angular has to check every component every single time an event happens because maybe the application state has changed.

When using Immutable objects or Observables we can optimize the Change Detection Algorithm.

Main Points

Angular separates updating the application model and updating the view.

Event bindings are used to update the application model.

Change detection uses property bindings to update the view. Updating the view is unidirectional and top-down. This makes the system more predictable and performant.

We make the system more efficient by using the OnPush change detection strategy for the components that depend on immutable input and only have local mutable state.