Do you really know what unidirectional data flow means in Angular

Max Koretskyi aka Wizard Follow
Nov 9, 2017 · 7 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.

Most architectural patterns are not easy to grasp especially when the information that describes them is scarce. One of such patterns in Angular is unidirectional data flow. There's no clear explanation of what that means in the official documentation and it's only briefly mentioned in the expression guidelines and template statements sections. I also haven't found any good detailed explanation on the web. So this article is intended to provide one.

> 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!**

. . .

Two-way data-binding VS unidirectional data flow

Unidirectional data flow as a pattern is usually mentioned when talking about performance differences between AngularJS and Angular. This is what makes Angular much faster than its predecessor. Let's see where this pattern comes into play.

Both frameworks have similar mechanism of communication between components through bindings. So if you have a parent component `A` defined like this in AngularJS:

```
app.component('aComponent', {
  controller: class ParentComponent() {
    this.value = {name: 'initial'};
  },
  template: `
    <b-component obj="$ctrl.value"></b-component>
  `
});


----------------
```

```
app.component('bComponent', {
    bindings: {
        obj: '='
    },
```

You can see that it has a child `B` component and the parent `A` component passes the `value` to the child `B` component through `obj` input binding:

```
<b-component obj="$ctrl.value"></b-component>
```

This is quite similar to what we would have in Angular:

```
@Component({
    template: `
        <b-component [obj]="value"></b-component>
    ...
export class AppComponent {
    value = {name: 'initial'};
}
```

```
----------------
```

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

The first important thing to understand is that **both Angular and AngularJS update bindings during change detection**. So when the framework runs change detection for the parent component `A` it will update the `obj` property on the child component `B`:

```
bComponentInstance.obj = aComponentInstance.value;
```

This process demonstrates one-way data binding or unidirectional data flow from top to bottom. But where AngularJS differ is that it can also **update the parent** component's bound `value` property **from the child**:

```
app.component('parentComponent', {
  controller: function ParentComponent($timeout) {
    $timeout(()=>{
      console.log(this.value); // logs {name: 'updated'}
    }, 3000)
  }
```

```
----------------

app.component('childComponent', {
    controller: function ChildComponent($timeout) {
      $timeout(()=>{
        this.obj = { name: 'updated' };
      }, 2000)
```

In the code snippet above you can see two timeouts with callbacks—first updates child component property while the second that is executed 1 second later checks if the parent component property has been updated. If you run the following code in AngularJS you will see

that that parent component property is updated. Here is how that happens.

Once the first timeout callback is executed and the `obj` property of the child `B` component is updated to `{name: 'updated'}` AngularJS runs change detection. During this process AngularJS detects the change in the bound child property and updates the parent's `value` property. **This is a built-in feature of the change detection mechanism in AngularJS**. If I do the same in Angular it will simply update the property on the child component but the change won't be propagated to the parent component. That's a very important distinction in the implementation of change detection mechanism, however, there was one thing that bothered me for quite some time.

In Angular we still have a mechanism to update parent component from the child component and this mechanism is output bindings. I can use it like this:

```
@Component({
    template: `
        <h1>Hello {{value.name}}</h1>
        <a-comp (updateObj)="value = $event"></a-comp>
    ...
export class AppComponent {
    value = {name: 'initial'};

    constructor() {
        setTimeout(() => {
            console.log(this.value); // logs {name:
'updated'}
        }, 3000);


_________________
```

```
@Component({...})
export class AComponent {
    @Output() updateObj = new EventEmitter();

    constructor() {
        setTimeout(() => {
            this.updateObj.emit({name: 'updated'});
        }, 2000);
```

I admit this is not the same as updating the input directly from the child component, but it's a parent update nevertheless. So for a long time I couldn't find the answer to the question why this isn't

considered a two-way data binding? After all, the communication goes in both directions.

But then one evening I read an article Two Phases of Angular Applications by Victor Savkin where he explains that:

> ***Angular 2 separates updating the application model and reflecting the state of the model in the view into two distinct phases.*** *The developer is responsible for updating the application model. Angular, by means of change detection, is responsible for reflecting the state of the model in the view.*

It took me a few days to realize that the parent property update using output binding mechanism:

```
<a-comp (updateObj)="value = $event"></a-comp>
```

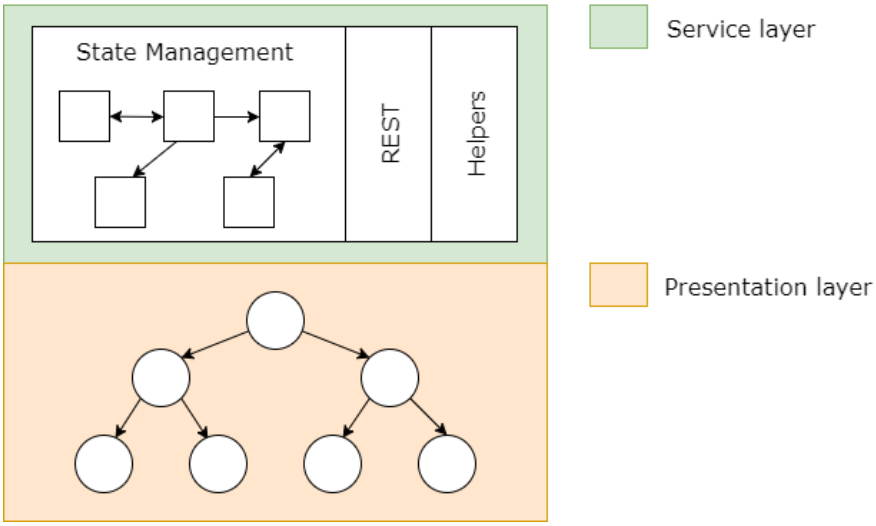
is not performed as part of change detection. It's executed during the first phase of updating the application model before the change detection starts. So the **unidirectional data flow** defines the architecture of **bindings updates that are processed during change detection**. Unlike AngularJS there is no code in change detection mechanism in Angular that propagates a child property updates to its parent. The **output bindings processing** is performed **outside of change detection** and hence doesn't transform unidirectional data flow into two-way data binding.

On a side note, although there's no built-in mechanism in Angular that can cause parent component model update during change detection it's still possible to do that through a shared service or synchronous event broadcasting. Since the framework enforces **unidirectional data flow** doing so will result in the often misunderstood error `ExpressionChangedAfterItHasBeenCheckedError`. To learn more about it, it's causes and possible fixes read Everything you need to know about the `ExpressionChangedAfterItHasBeenCheckedError` error.

• • •

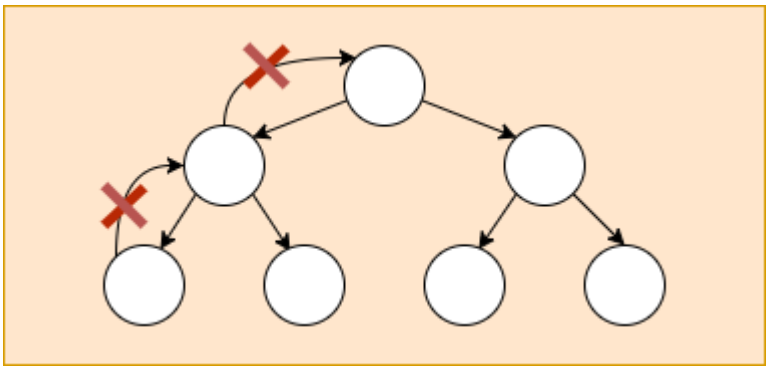
Unidirectional data flow in view and service layers

As you probably know, most web applications are designed to have two major architectural layers—view/presentation and service layers.



In a web environment the presentation layer contains functionality to display application relevant data to a user through DOM. In Angular this layer is implemented using components. The service layer implements functionality to process and store business relevant data. As shown in the picture above it can be split into state management and infrastructure parts like various REST services or reusable utility services (helpers).

The unidirectional data flow explained in the first chapter and which Angular mentions in the docs is relevant for the presentation layer of an application since it's the layer that relies on components.

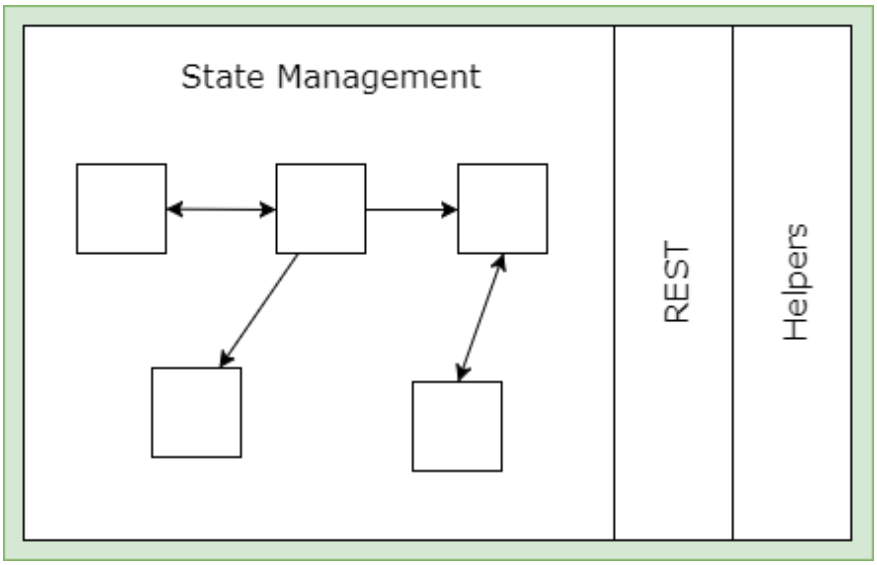


However, with the introduction of ngrx which implements redux like state management pattern another confusion was introduced. The

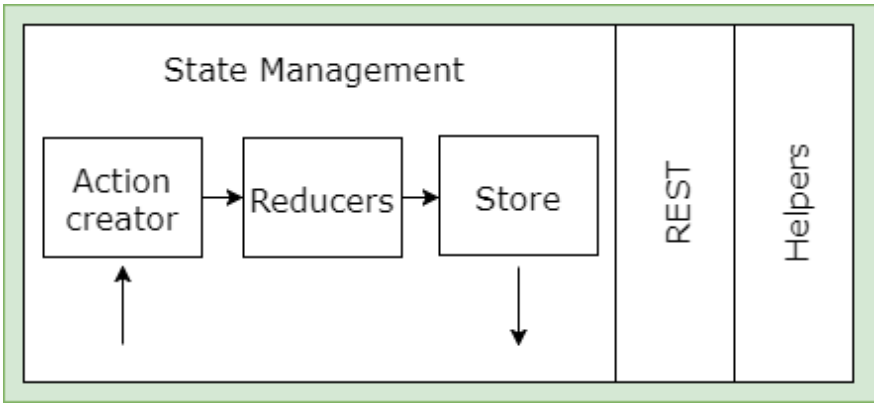
docs for the redux state the following:

> *Redux architecture revolves around a **strict unidirectional data flow**.*
> *This means that all data in an application follows the same lifecycle pattern, making the logic of your app more predictable and easier to understand…*

And this unidirectional data flow is related to the service layer, not presentation. But I sometimes come across unidirectional data flow explanations that mix up two layers and associate the pattern in redux architecture with that of Angular. It's important to avoid that confusion. The **unidirectional data flow that redux talks about is not relevant to the presentation layer.** It's related to the service layer, specifically state management module, and transforms the architecture we've seen above



into the following:



. . .

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.