The mechanics of property bindings update in Angular

Max Koretskyi aka Wizard Follow
Jul 1, 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.

All modern frameworks are designed to help implement UI composition with components. Naturally you will have a parent-child hierarchy and the framework should provide a way to communicate between the child and parent. Angular provides two major ways of communication—through input/output bindings and services. I prefer using the first method for the stateless presentational components while I employ DI for the stateful container components.

This article is concerned with the first method. Particularly it explores the underlying mechanism of how Angular updates **child input propertie**s when parent bound values change. It continues a series of articles on change detection and builds upon my previous article The mechanics of DOM updates in Angular. Since we will be looking how Angular updates input properties for both DOM elements and components it is assumed that you know how Angular represents directives and components internally. If you don't then go ahead and get familiar with the topic by reading Here is why you will not find components inside Angular. Throughout this article I'll be using terms directive and component interchangeably because internally Angular represents components as directives.

> 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!**

. . .

Binding template syntax

As you probably know Angular provides template syntax for property bindings— `[]` . This syntax is generic so it can be applied for both child components and native DOM elements. So if you want to pass some data to the child `b-comp` and `span` from the parent `A` component you do it like this in the component's template:

```
import { Component } from '@angular/core';

@Component({
  moduleId: module.id,
  selector: 'a-comp',
  template: `
      <b-comp [textContent]="AText"></b-comp>
      <span [textContent]="AText"></span>
  `
})
export class AComponent {
  AText = 'some';
}
```

You don't have to do anything else for the `span`, but for the `b-comp` you need to indicate that it receives `textContent` property:

```
@Component({
    selector: 'b-comp',
    template: 'Comes from parent: {{textContent}}'
})
export class BComponent {
    @Input() textContent;
}
```

Now, whenever the `AText` property changes on the `A` component Angular will automatically update the property `textContent` on `B` component and `span`. It will also call `ngOnChanges` lifecycle hook for the child component.

You may be wondering how Angular knows that `BComponent` and `span` support `textContent` binding. For the simple DOM elements it's defined in the dom_element_schema_registry used by the compiler when parsing a template. For the components and directives it checks the metadata attached to the class and ensures that bound property is listed in the `input` decorator property. If the bound property not found the compiler throws an error:

> *Can't bind to 'text' since it isn't a known property of …*

This is a well document functionality and there should be no problem with its understanding. Now let's look at what happens internally.

. . .

Factory specific information

It's important to understand that although we specified input bindings on the `B` and `span` all the relevant information for input update is defined on the parent `A` factory. Let's take a look at the factory generated for the `A` component:

```
function View_AComponent_0(_l) {
  return jit_viewDef1(0, [
     jit_elementDef2(..., 'b-comp', ...),
     jit_directiveDef5(..., jit_BComponent6, [], {
         textContent: [0, 'textContent']
     }, ...),
     jit_elementDef2(..., 'span', [], [[8, 'textContent',
0]], ...)
  ], function (_ck, _v) {
     var _co = _v.component;
     var currVal_0 = _co.AText;
     var currVal_1 = 'd';
     _ck(_v, 1, 0, currVal_0, currVal_1);
  }, function (_ck, _v) {
     var _co = _v.component;
     var currVal_2 = _co.AText;
     _ck(_v, 2, 0, currVal_2);
```

```
    });
  }
```

If you’ve read the articles I mentioned in the beginning all the view nodes in the factory should be familiar to you already. The first two `jit_elementDef2` and `jit_directiveDef5` are an element and directive definition nodes that constitute our `B` component. The third is an element definition for the `span` .

Node definition bindings

One thing that distinguishes this factory from other you may have seen is the parameters that these node definitions take. Our `jit_directiveDef5` receives a one new parameter here:

```
jit_directiveDef5(..., jit_BComponent6, [], {
    textContent: [0, 'textContent']
}, ...),
```

This parameter is called **props** as you may see from the `directiveDef` function parameters list:

```
directiveDef(..., props?: {[name: string]: [number,
string]}, ...)
```

It is an object with keys where each key defines a bindings index and the property name to update. For our example there will be only one binding for the `textContent` property:

```
{textContent: [0, 'textContent']}
```

If our directive received several bindings, for example, like this:

```
<b-comp [textContent]="AText" [otherProp]="AProp">
```

The props parameter would contain two properties:

```
jit_directiveDef5(49152, null, 0, jit_BComponent6, [], {
    textContent: [0, 'textContent'],
    otherProp: [1, 'otherProp']
}, null),
```

When Angular creates a directive definition node it uses these values to generate binding for the view node. During change detection each binding determines the type of operation Angular should use to update the node and provides context information. The binding type is set using bindings flags. In case of property update for each binding the compiler sets the following flags:

```
export const enum BindingFlags {
    TypeProperty = 1 << 3,
```

And since we also have bindings on the `span` element, the compiler generates the props parameter supplied to the span element definition as well:

```
jit_elementDef2(..., 'span', [], [[8, 'textContent', 0]],
...)
```

For elements this parameter has a bit different structure—an array of props. The span has only one input binding so there is only one child array. The first number in the array specifies the type of operation to be used with the binding—which is property update:

```
export const enum BindingFlags {
    TypeProperty = 1 << 3, // 8
```

The other possible variants are the following and they are explained in the The mechanics of DOM updates in Angular:

```
TypeElementAttribute = 1 << 0,
TypeElementClass = 1 << 1,
TypeElementStyle = 1 << 2,
```

The compiler didn't provide operation type in the props for the directive definition since only properties can be updated for the directive so all bindings are set to `BindingFlags.TypeProperty`.

Update renderer and Update directives

The compiler also generated two functions in the factory:

```
function (_ck, _v) {
    var _co = _v.component;
    var currVal_0 = _co.AText;
    var currVal_1 = _co.AProp;
    _ck(_v, 1, 0, currVal_0, currVal_1);
},
function (_ck, _v) {
    var _co = _v.component;
    var currVal_2 = _co.AText;
    _ck(_v, 2, 0, currVal_2);
}
```

You should be familiar with the second updateRenderer function already from the article about DOM update. The first function is called `updateDirectives`. These both functions are defined by the ViewUpdateFn interface and both are attached to the view definition:

```
interface ViewDefinition {
  flags: ViewFlags;
  updateDirectives: ViewUpdateFn;
  updateRenderer: ViewUpdateFn;
```

What's interesting is that the body of these functions is almost the same. They both take two parameters `_ck` and `v` which reference the same entities in each case. So why two functions?

Well, it's because during change detection there are two distinct operations:

- updating input properties on the child components
- updating DOM elements of the current component

And these operations are performed at different stages during change detection cycle. So Angular has two functions each specializing on a particular node definition and calls them at different stages when checking a component:

- updateDirectives—performs updates for directives (`directiveDef`) and is called in the beginning of the check
- updateRenderer—performs updates for DOM elements (`elementDef`) and is called in the middle of the check

Both these functions are executed each time Angular performs change detection for a component and the parameters to the function are supplied by the change detection mechanism. Now, let's see what these functions do.

The `_ck` is short for `check` and references the function prodCheckAndUpdate. The other parameter is a component's view with nodes. The main task of these functions is to retrieve the current value of the bound property from the component instance and call the `_ck` function passing the view, node index and the retrieved value. The `nodeIndex` is the index of the view node for which the change detection should be performed. What's important to understand is that Angular performs DOM updates for each view node separately—that's why node index is required. If for example we had two spans and two directives:

```
<b-comp [textContent]="AText"></b-comp>
<b-comp [textContent]="AText"></b-comp>
<span [textContent]="AText"></span>
<span [textContent]="AText"></span>
```

the compiler would generate the following body for the `updateRenderer` and `updateDirectives` functions:

```
function(_ck, _v) {
    var _co = _v.component;
    var currVal_0 = _co.AText;

    // update first component
```

```
        _ck(_v, 1, 0, currVal_0);
        var currVal_1 = _co.AText;

        // update second component
        _ck(_v, 3, 0, currVal_1);
    },

    function(_ck, _v) {
        var _co = _v.component;
        var currVal_2 = _co.AText;

        // update first span
        _ck(_v, 4, 0, currVal_2);
        var currVal_3 = _co.AText;

        // update second span
        _ck(_v, 5, 0, currVal_3);
    }
```

There's really not much going on. The crux of the functionality is outside these two functions. Let's see what this functionality is.

• • •

Updating properties on DOM elements

We learnt above that `updateRenderer` function generated by the compiler as part of a components factory is used during change detection to update input properties on DOM elements. I mentioned that it is passed `_ck` function during change detection and this parameter references checkAndUpdate. This is a short generic function that makes a bunch of calls that eventually execute checkAndUpdateElement. The function basically checks whether the binding is of the angular special form `[attr.name, class.name, style.some]` or some node specific property:

```
case BindingFlags.TypeElementAttribute ->
setElementAttribute
case BindingFlags.TypeElementClass     -> setElementClass
case BindingFlags.TypeElementStyle     -> setElementStyle
case BindingFlags.TypeProperty         ->
setElementProperty;
```

Here is where the bindings we explored above are used. Since the bindings where set as `BindingFlags.TypeProperty` the function setElementProperty will be used. Inside it just calls setProperty method of a renderer to update the property on the DOM element.

. . .

Updating properties on directives

As the complement to the `updateRenderer` function covered in the previous section the compiler adds `updateDirective` function to the component factory that is used to update input properties on components. Just as with `updateRenderer` it is passed `_ck` function during change detection and this parameter references checkAndUpdate. Only this time the checkAndUpdateDirective function is used to perform update. This is because we're updating properties on the nodes marked as `NodeFlags.TypeDirective` . The function does the following things:

1. retrieves component/directive class instance from the view node
2. checks if the property changed
3. if value changed:
   a. updates relevant property on the class instance
   b. prepares SimpleChange data and updates oldValues
   c. sets state to `checksEnabled` if a component uses `OnPush` strategy
   d. call `ngOnChanges` lifecycle hook
4. calls `ngOnInit` lifecycle hook (if it's the first time the view is being checked)
5. call `ngDoCheck` lifecycle hook

Of course, all lifecycle hooks are called only if they are defined on the component/directive class. Angular uses `nodeDef` flags shown above for that. You can see it from this code:

```
if (... && (def.flags & NodeFlags.OnInit)) {
  directive.ngOnInit();
}
if (def.flags & NodeFlags.DoCheck) {
  directive.ngDoCheck();
}
```

Just as with DOM update all previous values are stored on view in oldValues property on the view. That’s it!

. . .

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