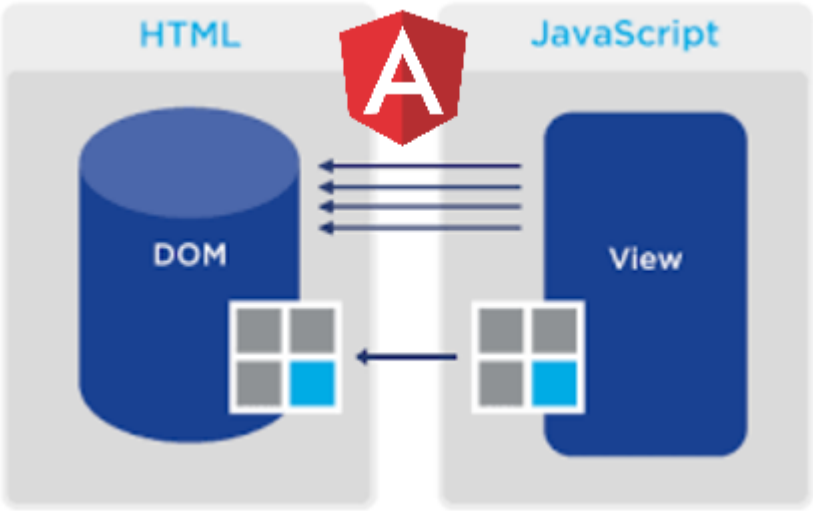
The mechanics of DOM updates in Angular

Max Koretskyi aka Wizard Follow
Jun 20, 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.

DOM updates that are triggered by the model change is the key feature of all modern front-end frameworks and the Angular is no exception. We just specify the expression like this:

```
<span>Hello {{name}}</span>
```

or a binding like this:

```
<span [textContent]="'Hello ' + name"></span>
```

and Angular magically updates the DOM whenever the `name` property changes. It's seems so easy on the outside but it's actually a pretty complicated process on the inside. DOM updates is part of Angular's change detection mechanism which mostly consists of three major operations:

- DOM updates
- child components `Input` bindings updates
- query list updates

This article explores how the rendering part of change detection works. If you ever wondered how that is accomplished read on and achieve enlightenment. When referencing sources I assume that the application is running in production mode. Let's start.

> 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!**

• • •

Application internal representation

Before we can begin exploring angular DOM updating functionality we need to first understand how Angular application is represented under the hood. Let's briefly review that.

View

As you may know from my other articles for each component that is used in the application Angular compiler generates a **factory**. When Angular creates a component from a factory, for example, like this:

```
const factory = r.resolveComponentFactory(AComponent);
factory.create(injector);
```

Angular uses this factory to instantiate View Definition which in turn is used to create component View. **Under the hood Angular represents an application as a tree of views**. There is only one instance of a view definition per component type which acts as a template for all views. But for each component instance Angular creates a separate view.

Factory

A component factory mostly consists of the view nodes generated by the compiler as a result of template parsing. Suppose you define a component's template like this:

```
<span>I am {{name}}</span>
```

Using this data the compiler generates the following component factory:

```
function View_AComponent_0(l) {
    return jit_viewDef1(0,
        [
          jit_elementDef2(0,null,null,1,'span',...),
          jit_textDef3(null,['I am ',...])
        ],
        null,
        function(_ck,_v) {
            var _co = _v.component;
            var currVal_0 = _co.name;
            _ck(_v,1,0,currVal_0);
```

It describes the structure of a component view and is used when instantiating the component. The `jit_viewDef1` is the reference to the viewDef function that creates a view definition.

View definition receives view definition nodes as parameters which resemble the structure of the html but also contain many angular specific details. In the example above the first node `jit_elementDef2` is element definition and the second `jit_textDef3` is a text definition. Angular compiler generates many different node definitions and the type associated with the node is set in the

NodeFlags. We will see later how Angular uses the information about the node type to decide how to handle the update.

For the purposes of this article we are only interested in element and text nodes:

```
export const enum NodeFlags {
    TypeElement = 1 << 0,
    TypeText = 1 << 1
```

Let's review them briefly.

Element definition

Element definition is a node that Angular generates for every html element. This kind of element is also generated for components. Element nodes can contain other element nodes and text definition nodes as children which is reflected in the `childCount` property.

All element definitions are generated by the elementDef function so `jit_elementDef2` used in the factory references this function. The element definition takes some generic parameters:

```
+------------------+-----------------------------------+
|       Name       |            Description            |
+------------------+-----------------------------------+
| childCount       | specifies how many children       |
|                  | the current element have          |
| namespaceAndName | the name of the html element      |
| fixedAttrs       | attributes defined on the element |
+------------------+-----------------------------------+
```

And others that are specific to the particular Angular functionality:

```
+----------------------+-------------------------------------
------+
|         Name         |                Description
|
+----------------------+-------------------------------------
------+
| matchedQueriesDsl    | used when querying child nodes
|
| ngContentIndex       | used for node projection
|
| bindings             | used for dom and bound properties
```

```
update |
| outputs, handleEvent | used for event propagation
|
+----------------------+-----------------------------------
------+
```

For the purposes of this article we're only interested in the bindings parameters.

Text definition

Text definition is a node definition that Angular compiler generates for every text node. Usually these are the child nodes of the element definition nodes as is the case in our example. This is a very simple node definition that is generated by the textDef function. It receives parsed expressions in the form of constants as a second parameter. For example, the following text:

```
<h1>Hello {{name}} and another {{prop}}</h1>
```

will be parsed as an array:

```
["Hello ", " and another ", ""]
```

which is then used to generate correct bindings:

```
{
  text: 'Hello',
  bindings: [
    {
      name: 'name',
      suffix: ' and another '
    },
    {
      name: 'prop',
      suffix: ''
    }
  ]
}
```

and evaluated like this during dirty checking:

```
text
+ context[bindings[0][property]] + context[bindings[0]
[suffix]]
+ context[bindings[1][property]] + context[bindings[1]
[suffix]]
```

Node definition bindings

Angular uses bindings to define dependencies of each node on the component class properties. During change detection each binding determines the type of operation Angular should use to update the node and provides context information. The type of operation is determined by binding flags and for the DOM specific operations it constitutes the following list:

```
+----------------------+--------------------------+
|         Name         | Construction in template |
+----------------------+--------------------------+
| TypeElementAttribute | attr.name                |
| TypeElementClass     | class.name               |
| TypeElementStyle     | style.name               |
+----------------------+--------------------------+
```

Element and text definitions create these bindings internally based on the bindings flags identified by the compiler. Each node type has different logic specific to bindings generation.

• • •

Update renderer

What interests us the most is the function listed at the end of the factory `View_AComponent_0` generated by the compiler:

```
function(_ck,_v) {
    var _co = _v.component;
    var currVal_0 = _co.name;
    _ck(_v,1,0,currVal_0);
```

This function is called `updateRenderer`. It takes two parameters `_ck` and `v`. The `_ck` is short for `check` and references the function

prodCheckAndUpdate. The other parameter is a component's view with nodes. The `updateRenderer` function is executed each time when Angular performs change detection for a component and the parameters to the function are supplied by the change detection mechanism.

The main task of the `updateRenderer` function is to retrieve the current value of the bound property from component instance and call the `_ck` function passing the view, node index and the retrieved value. What's important to understand is that Angular performs DOM updates for each view node separately—that's why node index is required. You can clearly see it when checking the parameters list of the function referenced by `_ck` :

```
function prodCheckAndUpdateNode(
    view: ViewData,
    nodeIndex: number,
    argStyle: ArgumentType,
    v0?: any,
    v1?: any,
    v2?: any,
```

The `nodeIndex` is the index of the view node for which the change detection should be performed. If you have multiple expressions in your template:

```
<h1>Hello {{name}}</h1>
<h1>Hello {{age}}</h1>
```

the compiler will generate the following body for the `updateRenderer` function:

```
var _co = _v.component;

// here node index is 1 and property is `name`
var currVal_0 = _co.name;
_ck(_v,1,0,currVal_0);

// here node index is 4 and bound property is `age`
var currVal_1 = _co.age;
_ck(_v,4,0,currVal_1);
```

• • •

Updating the DOM

Now that we know all the specific objects that Angular compiler generates we can explore how the actual DOM update is performed using these objects.

We learnt above that `updateRenderer` function is passed `_ck` function during change detection and this parameter references prodCheckAndUpdate. This is a short generic function that makes a bunch of calls that eventually execute checkAndUpdateNodeInline function. There's a variation of that function for the cases when expressions count exceed 10.

The `checkAndUpdateNode` function is just a router that differentiates between the following types of view nodes and delegates check and update to the respective functions:

```
case NodeFlags.TypeElement   -> checkAndUpdateElementInline
case NodeFlags.TypeText      -> checkAndUpdateTextInline
case NodeFlags.TypeDirective ->
checkAndUpdateDirectiveInline
```

Now let's see what those functions do. For the `NodeFlags.TypeDirective` see The mechanics of property bindings update in Angular.

Type Element

It uses the function 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;
```

The respective function simply uses the corresponding method of the renderer to perform the required action on the node.

Type Text

It uses the function CheckAndUpdateText in both variations. Here is the gist of the function:

```
if (checkAndUpdateBinding(view, nodeDef, bindingIndex,
newValue)) {
    value = text + _addInterpolationPart(...);
    view.renderer.setValue(DOMNode, value);
}
```

It basically takes the current value passed from the `updateRenderer` function and compares it to the value from the previous change detection. A View holds old values in the oldValues property. If the value changed Angular uses the updated value to compose a string and update the DOM using a renderer.

. . .

Conclusion

I'm sure that's an insurmountable amount of information to take in. But by understanding it you'll be better off when designing an application or debugging DOM updates related issues. I suggest you also use a debugger and follow the executing logic explained in the article.

. . .

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.