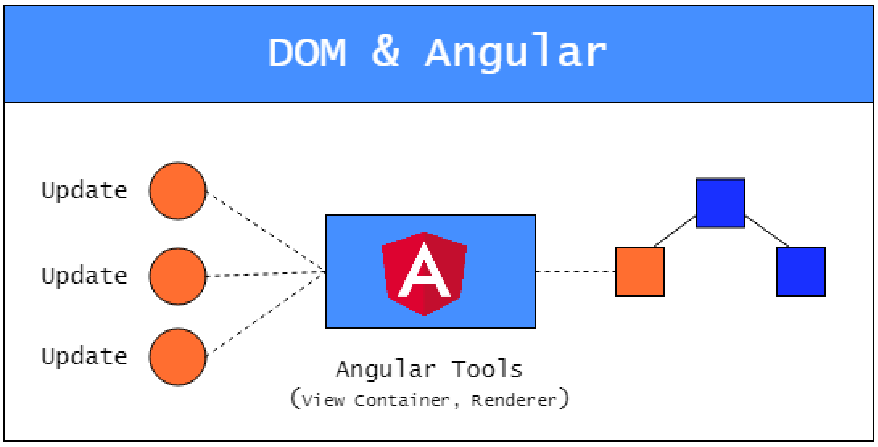
Working with DOM in Angular: unexpected consequences and optimization techniques

Max Koretskyi aka Wizard Follow
May 3, 2018 · 10 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.

I recently gave a talk on advanced DOM manipulations in Angular in a form of a workshop at NgConf. I went from the basics like using template references and DOM queries to access DOM elements to using a view container to render templates and components dynamically. If you haven't seen the talk already, I encourage you to do so. By going through a bunch of practical exercises you'll be able to learn and reinforce new knowledge much quicker. There's also a shorter talk on that subject I gave at NgViking.

However, if you want a TL;DR version or simply like reading more than listening I've summarized the key concepts in this article. I'll first explain the tools and approaches to working with DOM in Angular and then move on to a more advanced optimization techniques I didn't get to during the workshop.

You can find the examples I used in the talk in this github repository.

> 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!**

• • •

A peek into the View Engine

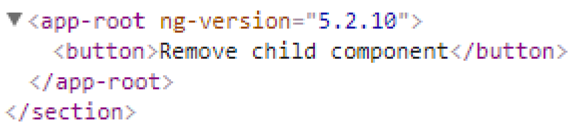
Suppose you have a task to remove a child component from the DOM. Here is a parent component's template with a child `A` component that needs to be removed:

```
@Component({
  ...
  template: `
    <button (click)="remove()">Remove child
component</button>
    <a-comp></a-comp>
  `
})
export class AppComponent {}
```

An **incorrect** approach to solving the task would be use either Renderer or native DOM API to remove the `<a-comp>` DOM element directly:

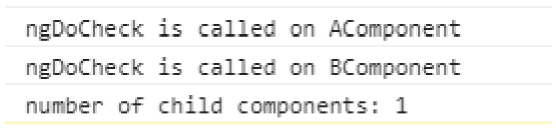
```
@Component({...})
export class AppComponent {
  ...
  remove() {
    this.renderer.removeChild(
       this.hostElement.nativeElement,      // parent App
comp node
       this.childComps.first.nativeElement  // child A comp
node
     );
  }
}
```

You can see the full solution here. If you inspect the resulting HTML in the `Elements` tab after removing the node, you will see that the child `A` component is no longer present in the DOM:



```
▼<app-root ng-version="5.2.10">
    <button>Remove child component</button>
  </app-root>
</section>
```

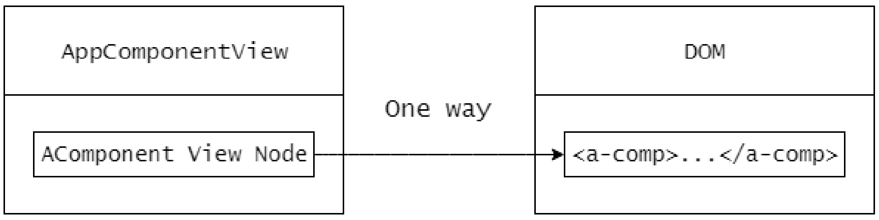
However, if you then check the console, Angular still reports the number of child components as `1` instead of `0`. And what's worse the change detection **is still run** for the child `A` component and its children. Here's the logs from the console:



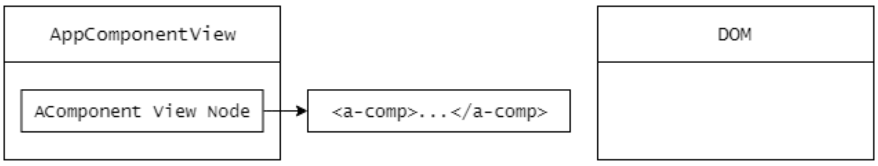
```
ngDoCheck is called on AComponent
ngDoCheck is called on BComponent
number of child components: 1
```

Why?

This happens because Angular internally represents a component using a data structure commonly referred to as a **View** or a **Component View**. Here is a diagram that represents a relationship between a view and DOM:



Each view consists of view nodes that hold references to corresponding DOM elements. So when we change the DOM directly, the view node that sits inside the view and holds a reference to that DOM element is not affected. Here is a diagram that shows the state of the view and DOM after we remove the `A` component element from the DOM:



And since all change detection operations, including ViewChildren run on a View, **not the DOM**, Angular detects one view corresponding to `A` component and reports the number `1`, instead of `0` as expected. Moreover, since the view corresponding to `A` component is there, it also runs change detection for the `A` component and all its children.

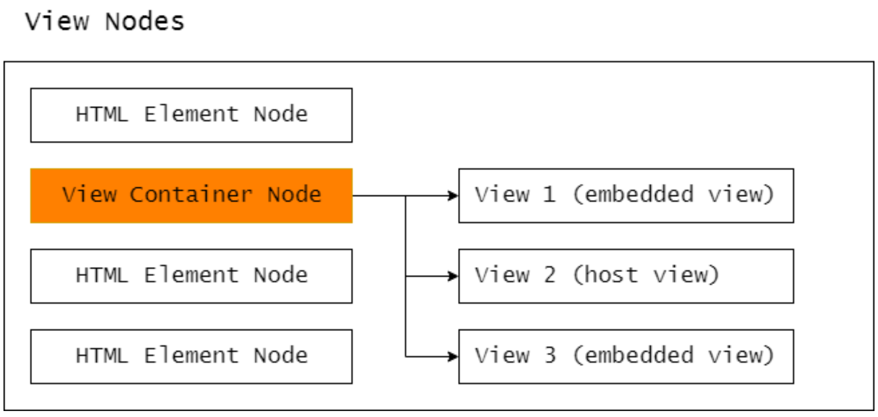
> *What this shows is that you can't simply remove child components directly from the DOM. As a matter of fact, you should avoid removing any HTML element created by the framework and only remove the elements Angular doesn't know about. These could be elements created by your code or by some 3 party plugin.*

To solve this task **correctly**, we need a tool that works directly with views and such tool in Angular is View Container.

. . .

View Container

A view container makes changes to DOM hierarchy safe and is used by all built-in structural directives in Angular. It is a special kind of a View Node that sits inside a View and acts as container for other views:

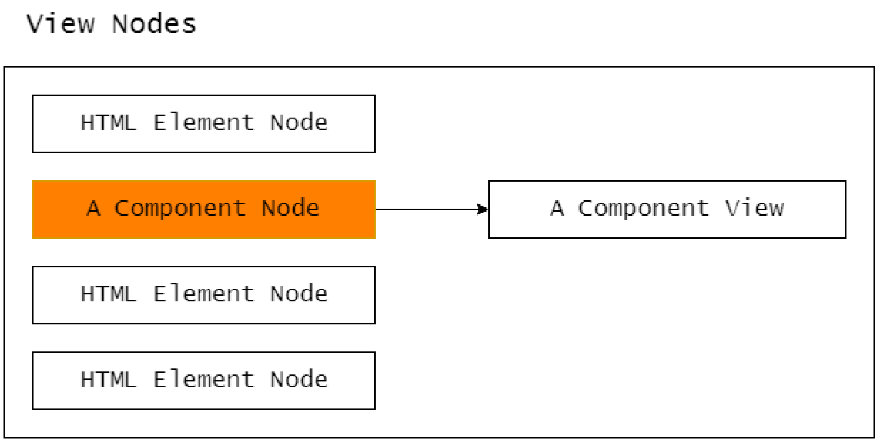


As you can see, it can hold two types of views: embedded and host views.

These are the only types of views that exist in Angular and they mainly differ depending on what input data is used to create them. Also, embedded views can only be attached to view containers, while host views can also be attached to any DOM element (usually referred to as host elements).

Embedded views are created from templates using TemplateRef, while host views are created using a view (component) factory. For example, the main component that is used to bootstrap an application (`AppComponent`) is represented internally as a host view attached to the component's host element (`<app-comp>`).

View Container provides API to create, manipulate and remove dynamic views. I call them dynamic views as opposed to static views created by the framework for static components found in templates. Angular doesn't use a View Container for static views and instead holds a reference to child views inside the node specific to the child component. Here is a diagram that illustrates that idea:



As you can see, there's no view container node here and the reference to the child view is attached directly to the `A` component view node.

• • •

Manipulating dynamic views

Before you can start creating and attaching views to a view container, you need to introduce that container into a component's template and initialize it. Any element inside a template can act as a view container, but the most common candidate for that role is `<ng-container>` because it's rendered as a comment node and hence doesn't introduce redundant elements to the DOM.

To turn any element into a view container we use `{read: ViewContainerRef}` option to a view query:

```
@Component({
 …
 template: `<ng-container #vc></ng-container>`
})
export class AppComponent implements AfterViewChecked {
  @ViewChild('vc', {read: ViewContainerRef}) viewContainer:
ViewContainerRef;
}
```

Once Angular evaluates the view query and assigns the reference to a view container to a class property, you can use the reference to create a dynamic view.

Creating an embedded view

To create an **embedded** view you need a **template**. In Angular, we use `<ng-template>` element to wrap around any DOM elements and to define the structure of a template. Then we can simply use a view query with `{read: TemplateRef}` parameter to get a reference to the template:

```
@Component({
  ...
  template: `
    <ng-template #tpl>
        <!-- any HTML elements can go here -->
    </ng-template>
  `
})
export class AppComponent implements AfterViewChecked {
    @ViewChild('tpl', {read: TemplateRef}) tpl:
TemplateRef<null>;
}
```

Once Angular evaluates this query and assigns the reference to the template to a class property, we can use the reference to create and attach an embedded view to a view container using `createEmbeddedView` method:

```
@Component({ ... })
export class AppComponent implements AfterViewInit {
    ...
    ngAfterViewInit() {
        this.viewContainer.createEmbeddedView(this.tpl);
    }
}
```

You should implement your logic inside `ngAfterViewInit` lifecycle hook because that's when view queries are initialized. Also, for embedded views, you can define a context object with values used for bindings inside a template. Check API docs for more details.

You can find a full example of creating an embedded view here.

Creating a host view

To create a **host** view, you need a component **factory**. To learn more about factories and dynamic components check Here is what you need to know about dynamic components in Angular.

In Angular, we use the `componentFactoryResolver` service to obtain a reference to a component factory:

```
@Component({ ... })
export class AppComponent implements AfterViewChecked {
  ...
  constructor(private r: ComponentFactoryResolver) {}
  ngAfterViewInit() {
    const factory =
this.r.resolveComponentFactory(ComponentClass);
  }
 }
}
```

Once we get the factory for a component, we can use it to initialize the component, create the host view and attach this view to a view container. To do that we simply call `createComponent` method and pass in a component factory:

```
@Component({ ... })
export class AppComponent implements AfterViewChecked {
    ...
    ngAfterViewInit() {
        this.viewContainer.createComponent(this.factory);
    }
}
```

You can find a full example of creating a host view here.

Removing a view

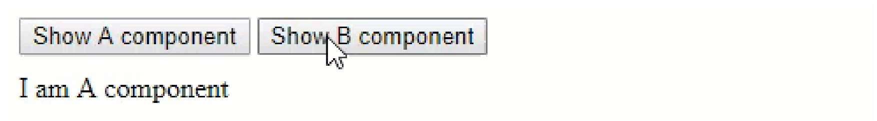
Any view attached to a view container can be removed using either `remove` or `detach` methods. Both method remove a view from a view container and the DOM. But while the `remove` method destroys the view so it can't be re-attached later, the `detach` method preserves it to be re-used in the future which is important for optimization techniques I'll show next.

So to correctly solve the task of removing a child component or any DOM element it is necessary to first create either an embedded or a host view and attach it to a view container. And after doing that you will be able to use any of the available API methods to remove it from a view container and the DOM.

· · ·

Optimization techniques

Sometimes you may need to repeatedly render and hide the same component or HTML defined by a template. In the example below, by clicking on different buttons we're toggling the component to show:



I am A component

If we simply use the approach we learnt above and put the knowledge into the following code to achieve that:

```
@Component({...})
export class AppComponent {
  show(type) {
    ...
    // a view is destroyed
    this.viewContainer.clear();

    // a view is created and attached to a view container
    this.viewContainer.createComponent(factory);
  }
}
```

we'll end up with an undesirable consequence of destroying and re-creating views each time a button is clicked and the `show` method is executed.

In this particular example it's the host view that is destroyed and re-created since we're using a component factory and `createComponent` method. If instead we used the `createEmbeddedView` method and a `TemplateRef`, an embedded view would be destroyed and re-created:

```
show(type) {
    ...
    // a view is destroyed
    this.viewContainer.clear();


    // a view is created and attached to a view container
    this.viewContainer.createEmbeddedView(this.tpl);
}
```

Ideally, we need to create a view once and then reuse it later when needed. And a view container API provides a way to attach an existing view to a view container and remove it later without destroying it.

ViewRef

Both `ComponentFactory` and `TemplateRef` implement view creation methods that can be used to create a view. In fact, a view container uses these methods under the hood when you call its `createEmbeddedView` or `createComponent` methods and pass in an input data. The good news is that we can call these methods ourselves to create an embedded or a host view and obtain a reference to the view. In Angular views are referenced using ViewRef type and its subtypes.

Creating a host view

So this is how you use a component factory to create a host view and get a reference to it:

```
aComponentFactory =
resolver.resolveComponentFactory(AComponent);
aComponentRef = aComponentFactory.create(this.injector);
view: ViewRef = aComponentRef.hostView;
```

In the case of a **host** view, the view associated with a component can be retrieved from ComponentRef returned by `create` method. It is exposed through similarly named property `hostView`.

Once we've got the view, it then can be attached to a view container using `insert` method. The other view you no longer want to show can be removed and preserved using `detach` method. So the **optimized solution** for the task with toggled components should be implemented like this:

```
showView2() {
    ...
    // Existing view 1 is removed from a view container and
the DOM
    this.viewContainer.detach();

    // Existing view 2 is attached to a view container and
the DOM
    this.viewContainer.insert(view);
}
```

Notice again that we're using `detach` method instead of `clear` or `remove` to preserve the view for later reuse. You can find the full implementation here.

Creating an embedded view

In the case of an **embedded** view created based on a template, the view is returned directly by `createEmbeddedView` method:

```
view1: ViewRef;
view2: ViewRef;

ngAfterViewInit() {
    this.view1 = this.t1.createEmbeddedView(null);
    this.view2 = this.t2.createEmbeddedView(null);
}
```

Then similarly to the previous example one view can be removed from a view container and the other re-attached. Again you can find the full implementation here.

Interestingly, both view creation methods `createEmbeddedView` and `createComponent` of a view container also return a reference to the created view.

. . .

New Ivy View Engine… Interested to know in-depth details?

Check out Ivy engine in Angular: first in-depth look at compilation, runtime and change detection.

. . .

My new course on advanced DOM manipulation

I’m also starting working on a DOM manipulation course now. There I’ll touch upon every single DOM related abstraction in Angular like ElementRef, ViewRef, ViewContaierRef and others. To provide a comprehensive explanations I’ll dive into the architecture of a View Layer. We’ll see if that will be current implementation or Ivy. I plan to include in the course the solutions to some common problems I often encounter on StackOverflow.

If you’re interested, here is a sign-up form:

Sign up for the course on DOM manipulation

Email

Sign up

☐ I agree to leave Blog.angularindepth.com and submit this information, which will be collected and used according to Upscribe's privacy policy.

. . .

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