Exploring Angular DOM manipulation techniques using ViewContainerRef

Max Koretskyi aka Wizard Follow
Mar 5, 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.

Whenever I read about working with DOM in Angular I always see one or few of these classes mentioned: `ElementRef`, `TemplateRef`, `ViewContainerRef` and others. Unfortunately, although some of them are covered in Angular docs or related articles, I've yet to found the description of the overall mental model and examples of how these work together. This article aims to describe such model.

> *If you're looking for more **in-depth** information on DOM manipulation in Angular using Renderer and View Containers check out my talk at NgVikings. Or read an in-depth article on dynamic DOM manipulation Working with DOM in Angular: unexpected consequences and optimization techniques*

If you come from `angular.js` world, you know that it was pretty easy to manipulate the DOM. Angular injected DOM `element` into `link` function and you could query any node within component's template, add or remove child nodes, modify styles etc. However, this approach had one major shortcoming—it was tightly bound to a browser platform.

The new Angular version runs on different platforms—in a browser, on a mobile platform or inside a web worker. So a level of abstraction is required to stand between platform specific API and the framework

interfaces. In angular these abstractions come in a form of the following reference types: `ElementRef`, `TemplateRef`, `ViewRef`, `ComponentRef` and `ViewContainerRef`. In this article we'll take a look at each reference type in detail and show how they can be used to manipulate DOM.

> 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!**

@ViewChild

Before we explore the DOM abstractions, let's understand how we access these abstractions inside a component/directive class. Angular provides a mechanism called DOM queries. It comes in a form of `@ViewChild` and `@ViewChildren` decorators. They behave the same, only the former returns one reference, while the latter returns multiple references as a QueryList object. In this article in examples I'll be using mostly `ViewChild` decorator and will not be using `@` symbol before it.

Usually, these decorators work in pair with template reference variables. A **template reference variable** is simply a named reference to a DOM element within a template. You can view it as something similar to `id` attribute of an `html` element. You mark a DOM element with a template reference and then query it inside a class using `ViewChild` decorator. Here is the basic example:

```
@Component({
    selector: 'sample',
    template: `
        <span #tref>I am span</span>
    `
})
export class SampleComponent implements AfterViewInit {
    @ViewChild("tref", {read: ElementRef}) tref: ElementRef;

    ngAfterViewInit(): void {
        // outputs `I am span`
        console.log(this.tref.nativeElement.textContent);
    }
}
```

The basic syntax to `ViewChild` decorator is the following:

```
@ViewChild([reference from template], {read: [reference
type]});
```

In this example you can see that I specified `tref` as a template reference name in `html` and receive `ElementRef` associated with this element. The second parameter `read` is not always required, since angular can infer the reference type by the type of the DOM element. For example, if it's a simple html element like `span`, angular returns `ElementRef.` If it's a `template` element, it returns `TemplateRef`. Some references, like `ViewContainerRef` cannot be inferred and have to be asked for specifically in `read` parameter. Others, like `ViewRef` cannot be returned from the DOM and have to be constructed manually.

Okay, now that we know how to query for the references, let's start exploring them.

ElementRef

This is the most basic abstraction. If you observe it's class structure, you'll see that it only holds the native element it's associated with. It's useful for accessing native DOM element as we can see here:

```
// outputs `I am span`
console.log(this.tref.nativeElement.textContent);
```

However, such usage is discouraged by Angular team. Not only it poses security risk, but it also creates tight coupling between your application and rendering layers which makes is difficult to run an app on multiple platforms. I believe that it's not the access to `nativeElement` that breaks the abstraction, but rather usage of specific DOM API like `textContent`. But as you'll see later the DOM manipulation mental model implemented in Angular hardly ever requires such a lower level access.

`ElementRef` can be returned for any DOM element using `ViewChild` decorator. But since all components are hosted inside a custom DOM element and all directives are applied to DOM elements, component and directive classes can obtain an instance of `ElementRef` associated with their host element through DI mechanism:

```
@Component({
    selector: 'sample',
    ...

export class SampleComponent{
    constructor(private hostElement: ElementRef) {
        //outputs <sample>...</sample>

console.log(this.hostElement.nativeElement.outerHTML);
    }
```

So while a component can get access to it's host element through DI, the `ViewChild` decorator is used most often to get a reference to a DOM element in their view (template). And it's vice verse with directives—they have no views and they usually work directly with the element they are attached to.

TemplateRef

The notion of template should be familiar for most web developers. It's a group of DOM elements that are reused in views across the app. Before HTML5 standard introduced template tag, most templates arrived to a browser wrapped in a script tag with some variations of `type` attribute:

```
<script id="tpl" type="text/template">
  <span>I am span in template</span>
</script>
```

This approach certainly had many drawbacks like the semantics and the necessity to manually create DOM models. With `template` tag a browser parses `html` and creates `DOM` tree but not renders it. It then can be accessed through `content` property:

```
<script>
    let tpl = document.querySelector('#tpl');
    let container = document.querySelector('.insert-after-
me');
    insertAfter(container, tpl.content);
</script>
<div class="insert-after-me"></div>
<ng-template id="tpl">
    <span>I am span in template</span>
</ng-template>
```

Angular embraces this approach and implements `TemplateRef` class to work with a template. Here is how it can be used:

```
@Component({
    selector: 'sample',
    template: `
        <ng-template #tpl>
            <span>I am span in template</span>
        </ng-template>
    `
})
export class SampleComponent implements AfterViewInit {
    @ViewChild("tpl") tpl: TemplateRef<any>;

    ngAfterViewInit() {
        let elementRef = this.tpl.elementRef;
        // outputs `template bindings={}`
        console.log(elementRef.nativeElement.textContent);
    }
}
```

The framework removes `template` element from the DOM and inserts a comment in its place. This is how it looks like when rendered:

```
<sample>
    <!--template bindings={}-->
</sample>
```

By itself the `TemplateRef` class is a simple class. It holds a reference to its host element in `elementRef` property and has one method `createEmbeddedView`. However, this method is very useful since it allows us to create a view and return a reference to it as `ViewRef`.

ViewRef

This type of abstraction represents an angular View. In angular world a View is a fundamental building block of the application UI. It is the smallest grouping of elements which are created and destroyed together. Angular philosophy encourages developers to see UI as a composition of Views, not as a tree of standalone html tags.

Angular supports two types of views:

- *Embedded Views* which are linked to a *Template*

- *Host Views* which are linked to a *Component*

Creating embedded view

A template simply holds a blueprint for a view. A view can be instantiated from the template using aforementioned `createEmbeddedView` method like this:

```
ngAfterViewInit() {
    let view = this.tpl.createEmbeddedView(null);
}
```

Creating host view

Host views are created when a component is dynamically instantiated. A component can be created dynamically using `ComponentFactoryResolver` :

```
constructor(private injector: Injector,
            private r: ComponentFactoryResolver) {
    let factory =
this.r.resolveComponentFactory(ColorComponent);
    let componentRef = factory.create(injector);
    let view = componentRef.hostView;
}
```

In Angular, each component is bound to a particular instance of an injector, so we're passing the current injector instance when creating the component. Also, don't forget that components that are instantiated dynamically must be added to EntryComponents of a module or hosting component.

So, we've seen how both embedded and host views can be created. Once a view is created it can be inserted into the DOM using `ViewContainer` . The next section explores its functionality.

ViewContainerRef

Represents a container where one or more views can be attached.

The first thing to mention here is that any DOM element can be used as a view container. What's interesting is that Angular doesn't insert views inside the element, but appends them after the element bound

to `ViewContainer`. This is similar to how `router-outlet` inserts components.

Usually, a good candidate to mark a place where a `ViewContainer` should be created is `ng-container` element. It's rendered as a comment and so it doesn't introduce redundant html elements into DOM. Here is the example of creating a `ViewContainer` at the specific place in a components template:

```
@Component({
    selector: 'sample',
    template: `
        <span>I am first span</span>
        <ng-container #vc></ng-container>
        <span>I am last span</span>
    `
})
export class SampleComponent implements AfterViewInit {
    @ViewChild("vc", {read: ViewContainerRef}) vc:
ViewContainerRef;

    ngAfterViewInit(): void {
        // outputs `template bindings={}`

console.log(this.vc.element.nativeElement.textContent);
    }
}
```

Just as other DOM abstractions, `ViewContainer` is bound to a particular DOM element accessed through `element` property. In the example about it's bound to `ng-container` element rendered as a comment, and so the output is `template bindings={}`.

Manipulating views

`ViewContainer` provides a convenient API for manipulating the views:

```
class ViewContainerRef {
    ...
    clear() : void
    insert(viewRef: ViewRef, index?: number) : ViewRef
    get(index: number) : ViewRef
    indexOf(viewRef: ViewRef) : number
    detach(index?: number) : ViewRef
    move(viewRef: ViewRef, currentIndex: number) : ViewRef
}
```

We've seen earlier how two types of views can be manually created from a template and a component. Once we have a view, we can insert it into a DOM using `insert` method. So, here is the example of creating an embedded view from a template and inserting it in a particular place marked by `ng-container` element:

```
@Component({
    selector: 'sample',
    template: `
        <span>I am first span</span>
        <ng-container #vc></ng-container>
        <span>I am last span</span>
        <ng-template #tpl>
            <span>I am span in template</span>
        </ng-template>
    `
})
export class SampleComponent implements AfterViewInit {
    @ViewChild("vc", {read: ViewContainerRef}) vc:
ViewContainerRef;
    @ViewChild("tpl") tpl: TemplateRef<any>;

    ngAfterViewInit() {
        let view = this.tpl.createEmbeddedView(null);
        this.vc.insert(view);
    }
}
```

With this implementation, the resulting `html` looks like this:

```
<sample>
    <span>I am first span</span>
    <!--template bindings={}-->
    <span>I am span in template</span>

    <span>I am last span</span>
    <!--template bindings={}-->
</sample>
```

To remove a view from the DOM, we can use `detach` method. All other methods are self explanatory and can be used to get a reference to a view by the index, move the view to another location or remove all views from the container.

Creating Views

`ViewContainer` also provides API to create a view automatically:

```
class ViewContainerRef {
    element: ElementRef
    length: number

    createComponent(componentFactory...): ComponentRef<C>
    createEmbeddedView(templateRef...): EmbeddedViewRef<C>
    ...
}
```

These are simply convenient wrappers to what we've done manually above. They create a view from a template or component and insert it at the specified location.

ngTemplateOutlet and ngComponentOutlet

While it's always good to know how the underlying mechanism works, it's usually desirable to have some sort of a shortcut. This shortcut comes in a form of two directives: `ngTemplateOutlet` and `ngComponentOutlet`. At the time of the writing both are experimental and `ngComponentOutlet` will be available as of version 4. But if you've read everything above it'll be very easy to understand what they do.

ngTemplateOutlet

This one marks a DOM element as a `ViewContainer` and inserts an embedded view created by a template in it without the need to explicitly doing this in component class. This means that the example above where we created a view and inserted it into `#vc` DOM element can be rewritten like this:

```
@Component({
    selector: 'sample',
    template: `
        <span>I am first span</span>
        <ng-container [ngTemplateOutlet]="tpl"></ng-
container>
        <span>I am last span</span>
        <ng-template #tpl>
            <span>I am span in template</span>
        </ng-template>
    `
})
export class SampleComponent {}
```

As you can see we don't use any view instantiating code in the component class. Very handy.

ngComponentOutlet

This directive is analogues to `ngTemplateOutlet` with the difference that it creates a **host view** (instantiates a component), not an embedded view. And you can use it like this:

```
<ng-container *ngComponentOutlet="ColorComponent"></ng-
container>
```

Wrapping up

Now all this information may seem a lot to digest, but actually it's pretty coherent and lays out into a clear mental model for manipulating DOM via views. You get a reference to Angular DOM abstractions by using `ViewChild` query along with template variable references. The simplest wrapper around a DOM element is `ElementRef`. For templates you have `TemplateRef` that allows you to create an embedded view. Host views can be accessed on `componentRef` created using `ComponentFactoryResolver`. The views can be manipulated with `ViewContainerRef`. There are two directives that make the manual process automatic: `ngTemplateOutlet` —for embedded views and `ngComponentOutlet` for host views (dynamic components).

. . .

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.