Exploring Angular DOM manipulation techniques using ViewContainerRef

The article explores common elements used for DOM manipulation in Angular with a particular focus on ViewContainerRef. Learn why they are needed, how they work and when you need to use which.

Angular | Dom-Manipulation | View-Container-Ref

27 November 2019 9 min read 2 comments

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.

@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 the form of `@ViewChild` and `@ViewChildren` decorators. They behave the same, only the former returns one reference, while

ABOUT THE AUTHOR

Max Koretskyi

Max is a self-taught software engineer that believes in fundamental knowledge and hardcore learning. He's the founder of inDepth.dev community and one of the top users on StackOverflow (70k rep).

the latter returns multiple references as a QueryList object. In the examples in this article I'll be using mostly the `ViewChild` decorator and will not be using the `@` symbol before it.

Usually, these decorators are paired 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 the `id` attribute of an `html` element. You mark a DOM element with a template reference and then query it inside a class using the `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 of the `ViewChild` decorator is:

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

In this example you can see that I specified `tref` as a template reference name in the `html` and receive the `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 the `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 its 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 the Angular team. Not only does it pose a security risk, but it also tightly couples 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 the usage of a specific DOM API like `textContent`. But as you'll see later, the DOM manipulation mental model implemented in Angular hardly ever requires such lower level access.

`ElementRef` can be returned for any DOM element using the `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 **Dependency**

Injection (DI):

```
@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 its host element through DI, the `ViewChild` decorator is used most often to get a reference to a DOM element in its view (template). But, it's reversed for directives — they have no views and they usually work directly with the element they are attached to.

TemplateRef

The notion of a **template** should be familiar for most web developers. It's a group of DOM elements that are reused in views across the application. Before the HTML5 standard introduced the template tag, most templates arrived to the browser wrapped in a `script` tag with some variation of the `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 the `template` tag a browser parses `html` and creates a `DOM` tree but doesn't render it. It can then be accessed through the `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 the `template` element from the DOM and inserts a comment in its place. This is how it looks when rendered:

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

By itself the `TemplateRef` class is a simple class. It holds a reference to its host element in the `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 the 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 the 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 an embedded view

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

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

Creating a 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 the 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

ViewContainerRef represents a container where one or more views can be attached

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 the `router-outlet` inserts components.

Usually, a good candidate to mark a place where a `ViewContainer` should be created is the `ng-container` element. It's rendered as a comment and so it doesn't introduce redundant HTML elements into the DOM. Here is the example of creating a `ViewContainer`at a specific place in a component 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 with other DOM abstractions, `ViewContainer` is bound to a particular DOM element accessed through the `element` property. In the example above it's bound to the `ng-container` element rendered as a comment, and so the output is `template bindings={}`.

Manipulating views

`ViewContainer` provides a convenient API for manipulating 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 the `insert` method. Here is the example of creating an embedded view from a template and inserting it into a specific place marked by an `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 the `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 an 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 mechanisms work, it's usually desirable to have some sort of a shortcut. This shortcut comes in the form of two directives: `ngTemplateOutlet` and `ngComponentOutlet`. At the time of this 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 without the need to explicitly do this in a component class. This means that the example above where we created a view and inserted it into a `#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 analogous to `ngTemplateOutlet` with the difference being that it creates a **host view** (instantiates a component), and not an embedded view. You can use it like this:

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

Wrapping up

I realize that all this information may be a lot to digest. But actually it's pretty coherent and lays out a clear mental model for manipulating the DOM via views. You get references to Angular DOM abstractions by using a `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).

Comments (2)

pardeepr08 12 June 2021

Awesome Maxim. I am really inspired after reading this article and a sense of inquisitiveness to know the angular internals has arisen.

Reply

ABSTRACT123 27 June 2021

hello,

Thanks for the article. I was looking for a way to dynamically insert html elements into an element.

for example I have

`<ul #somelist></ul>`

I would like to programmatically insert li elements into the #somelist elementRef.

There are indeed easy ways to do that, like with ngFor. But it is just for the sake of

learning/understanding Angular. I also could use JS to directly manipulate the document with createElement and appendChild... I also know how to dynamically insert components but was wondering if there was a way to just insert a template?

Thank you.

Reply

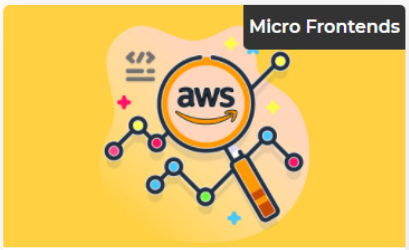
maxkoretskyi 29 June 2021

It should be possible by querying the `#somelist` element as `@ViewChild('somelist', {read: ViewContainerRef) container` and your template as `@ViewChild('mytpl')`, then you can call `createEmbeddedView` on the container and pass the reference to the mytpl.

More replies

JOIN THE DISCUSSION

Featured articles



How to Deploy a run-time Micro Frontend Application using AWS

Richard Bell

18 October 2021 14 min read

In this article we are going through deploying process your application to production using AWS and GitHub actions.

Read more →

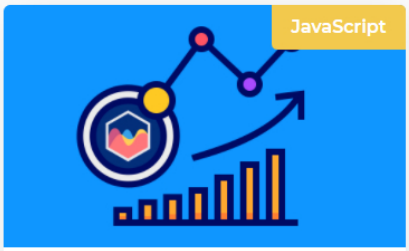
Exploring the difference between disabling a form...

Thabo Ambrose

17 October 2021 7 min read

Having parts of a form disabled is a common requirement for any large application. Sometimes users must be prevented from interacting with a form based on their role in an...

Read more →



An in-depth perspective on webpack's bundling...

Andrei Gatej

27 September 2021 30 min read

Webpack is a very powerful and interesting tool that can be considered a fundamental component in many of today's technologies that web developers...

Read more →

SIGN UP FOR OUR TOP-NOTCH NEWSLETTER

The Deep Dive

Get the latest coverage of **advanced** web development straight into your inbox. Twice a month.

See previous editions →

FOLLOW US

Name

Enter Your name

E-mail

Enter Your E-mail

What are you interested in?

E.g. JavaScript, Angular, React..

SUBSCRIBE