Understanding ViewChild, ViewChildren & Querylist in Angular

3 Comments / 10 minutes of reading / March 9, 2023

← HostBinding & HostListener

Angular Tutorial

Elementref →

The ViewChild or ViewChildren decorators are used to Query and get the reference of the DOM element in the Component. ViewChild returns the first matching element and ViewChildren returns all the matching elements as a QueryList of items. We can use these references to manipulate element properties in the component.

To Query a DOM element(s), we must supply the query selector, which can be a string or a type as the first argument to the ViewChild or ViewChildren. The argument static determines whether the query is performed, before or after the change detection. The read option allows us to query a different token rather than the default and is useful when the element is associated with multiple types. We will learn all these in this tutorial.

Table of Contents

ViewChild

The `ViewChild` query returns the first matching element from the DOM and updates the component variable on which we apply it.

Syntax

The Syntax of the `viewChild` is as shown below.

```
1
2 ViewChild(selector: string | Function | Type<any>, opts: { read?: any; static: boolean; }):
3
```

We apply the `viewChild` decorator on a **Component Property**. It takes two arguments. A `selector` and `opts`.

selector: can be a string, a type or a function that returns a string or type. The change detector looks for the first element that matches the selector and updates the component property with the reference to the element. If the DOM changes and a new element matches the selector, the change detector updates the component property.

opts: has two options.

static Determines when the query is resolved. `True` is when the view is initialized (before the first change detection) for the first time. `False` if you want it to be resolved after every change detection

read: Use it to read the different token from the queried elements.

ViewChild Examples

Now, let us learn `ViewChild` using few examples.

Injecting Component or Directive Reference

One of the use cases of `ViewChild` is to get the reference of the Child Component in the Parent Component and manipulate its properties. This is one of the ways by which the Parent can communicate with the child components.

For Example consider the following `ChildComponent`. It has two methods. `Increment` & `Decrement`.

```
1
2 import { Component } from '@angular/core';
3
4 @Component({
5   selector: 'child-component',
6   template: `<h2>Child Component</h2>
```

```
        current count is {{ count }}
    `
})
export class ChildComponent {
  count = 0;

  increment() {
    this.count++;
  }
  decrement() {
    this.count--;
  }
}

```

We can use the ViewChild in the parent component to get the reference to the ChildComponent

```

@ViewChild(ChildComponent, {static:true}) child: ChildComponent;

```

In the code above, the ViewChild looks for the first occurrence of the ChildComponent in the view and updates the child variable. Now we can invoke the methods Increment & Decrement of the ChildComponent from the Parent.

The complete code is as shown below.

```
import { Component, ViewChild } from '@angular/core';
import { ChildComponent } from './child.component';

@Component({
  selector: 'app-root',
  template: `
       <h1>{{title}}</h1>
       <p> current count is {{child.count}} </p>
       <button (click)="increment()">Increment</button>
       <button (click)="decrement()">decrement</button>
       <child-component></child-component>
       ` ,
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'Parent calls an @ViewChild()';

  @ViewChild(ChildComponent, {static:true}) child: ChildComponent;

  increment() {
    this.child.increment();
  }

  decrement() {
    this.child.decrement();
  }

}

```

Using Template Reference Variable

You can make use of Template Reference Variable instead of the component type.

For Example, you can assign a Template Reference Variable to a component.

```
<child-component #child></child-component>

```

and then use it in the ViewChild query to get the reference to the component.

```
@ViewChild("child", { static: true }) child: ChildComponent;
```

Injecting HTML Element Using ElementRef

The `Viewchild` can also be used to query HTML elements.

First, assign a Template variable (`#para` in the example below) to the HTML element. You can then use the `ViewChild` to query the element.

`ViewChild` returns a `ElementRef`, which is nothing but a wrapper around the native HTML element.

```
import {AfterViewInit, Component, ElementRef, ViewChild} from '@angular/core';

@Component({
   selector: 'htmlelement',
   template: `
     <p #para>Some text</p>
   `,
})
export class HTMLElementComponent implements AfterViewInit {

   @ViewChild('para',{static:false}) para: ElementRef;

   ngAfterViewInit() {
     console.log(this.para.nativeElement.innerHTML);
     this.para.nativeElement.innerHTML="new text"
   }
}
```

Multiple Instances

There could be multiple instances of the same component or element in the Template.

```
<child-component></child-component>
<child-component></child-component>
```

The `ViewChild` always returns the first component.

```
@ViewChild(ChildComponent, {static:true}) child: ChildComponent;
```

To get all the instances of the Child Component, we can make use of the `ViewChildren`, which we cover later in this tutorial.

ViewChild returns Undefined

ViewChild Returning undefined is one of the common errors, we encounter when we use them.

The error is due to the fact that we try to use the value, before the `ViewChild` initializes it.

For Example, the code below results in `Cannot read property 'increment' of undefined`. i.e. the component's view is not yet initialized when the `constructor` is run. Hence, the Angular yet to update `child` variable with the reference to the `ChildComponet`.

```

export class AppComponent {
  title = 'Parent calls an @ViewChild()';

  @ViewChild(ChildComponent, {static:true}) child: ChildComponent;

  constructor() {
    this.child.increment()
  }

}

//
Cannot read property 'increment' of undefined

```

The solution is to wait until the Angular Initializes the View. Angular raises the `AfterViewInit` life cycle hook once it completes the View Initialization. So we can use the `ngAfterViewInit` to access the `child` variable.

```

  ngAfterViewInit() {
    this.child.increment()
  }

```

Now, the code does not give any errors.

> The above code will also work with the `ngOnInit` Life cycle hook. But it is not guaranteed to work all the time as the Angular might not initialize all parts of the view, before raising the `ngOnInit` hook. Hence it is always better to use the `ngAfterViewInit` hook.

Also, when ViewChild updates the values also depends on the static option

Using Static Option in ViewChild

We used the {static:true} in the above code.

The static option determines the timing of the ViewChild query resolution.

- static:true will resolves ViewChild ***before*** **any** change detection is run.
- static:false will resolves it ***after*** **every** change detection run.

The value of the static becomes important when the child is rendered dynamically. For Example inside a ngIf or ngSwitch etc.

For Example consider the following code, where we have moved the child-component inside the ngIf .

```
1
2  //child.component.html
3
4  <h1>ViewChild Example</h1>
5
6  <input type="checkbox" id="showCounter" name="showCounter" [(ngModel)]="showCoun
7
8  <ng-container  *ngIf="showCounter">
9
10   <p> current count is {{child?.count}} </p>
11
12   <button (click)="increment()">Increment</button>
13   <button (click)="decrement()">decrement</button>
14
15   <child-component></child-component>
16
17 </ng-container>
18
```

```
1
2  //child.component.ts
3
4  import { Component, ViewChild, AfterViewInit, OnInit, ChangeDetectorRef } from '@angula
5  import { ChildComponent } from './child.component';
6
7  @Component({
8    selector: 'app-root',
9    templateUrl: 'app.component.html' ,
10   styleUrls: ['./app.component.css']
11 })
12 export class AppComponent {
13   title = 'ViewChild Example)';
14
15   showCounter: boolean = true
16
17   @ViewChild(ChildComponent, { static: true }) child: ChildComponent;
18
19   increment() {
20     this.child.increment();
21   }
22
23   decrement() {
24     this.child.decrement();
25   }
26
27 }
28
```

The above code results in a `TypeError: Cannot read property 'increment' of undefined` . The error occurs even if we assign `true` to `showCounter`

Because in the above case Angular does not render the child component immediately. But after the first change detection which detects the value of `showCounter` and renders the child component.

Since we used `static: true` , the angular will try to resolve the `ViewChild` before the first change detection is run. Hence the child variable always will be `undefined` .

Now, change the `static: false`. Now the code will work correctly. I.e because after every change detection the Angular updates the `ViewChild`.

Using the Read Option in ViewChild

A Single element can be associated with multiple types.

For Example, consider the following code. `#nameInput` template variable is now associated with both `input` & `ngModel`

```
<input #nameInput [(ngModel)]="name">
```

The `viewChild` code below, returns the instance of the `input` element as `elementRef`.

```
@ViewChild('nameInput',{static:false}) nameVar;
```

If we want to get the instance of the `ngModel`, then we use the `Read` token and ask for the type.

```
@ViewChild('nameInput',{static:false, read: NgModel}) inRef;

@ViewChild('nameInput',{static:false, read: ElementRef}) elRef;
@ViewChild('nameInput', {static:false, read: ViewContainerRef }) vcRef;
```

Every element in Angular is always has a `ElementRef` and `ViewContainerRef` associated with it. If the element is a component or directive then there is always a component or directive instance. You can also apply more than one directive to an element.

The `ViewChild` without `read` token always returns the component instance if it is a component. If not it returns the `elementRef`.

Injecting a Provider from the Child Component

You can also inject the services provided in the child component.

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

@Component({
  selector: 'app-child',
  template: `<h1>Child With Provider</h1>`,
  providers: [{ provide: 'Token', useValue: 'Value' }]
})

export class ChildComponent{
}
```

And in the Parent component, you can access the provider using the `read` property.

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

@Component({
  selector: 'app-root',
```

```
  template: `<app-child></app-child>`,
})

export class AppComponent{
   @ViewChild(ChildComponent , { read:'Token', static:false } ) childToken: string;
}

```

Injecting TemplateRef

You can access the TemplateRef as shown below

```

<ng-template #sayHelloTemplate>
  <p> Say Hello</p>
</ng-template>

```

Component code

```

@ViewChild("sayHelloTemplate", { static: false }) tempRef: TemplateRef;

```

ViewChildren

ViewChildren decorator is used to getting the **list of element references** from the View.

ViewChildren is different from the ViewChild . ViewChild always returns the reference to a single element. If there are multiple elements the ViewChild returns the first matching element,

ViewChildren always returns all the elements as a QueryList. You can iterate through the list and access each element.

Syntax

The Syntax of the `viewChildren` is as shown below. It is very much similar to syntax of `viewChild` except for the `static` option.

```
1
2 ViewChildren(selector: string | Function | Type<any>, opts: { read?: any; }): any
3
```

The `ViewChildren` is always resolved after the change detection is run. i.e why it does not have `static` option. And also you cannot refer to it in the `ngOnInit` hook as it is yet to initialize.

QueryList

The `QueryList` is the return type of `ViewChildren` and `contentChildren` .

`QueryList` stores the items returned by the `viewChildren` or `contentChildren` in a list.

The Angular updates this list, whenever the state of the application change. It does it on each change detection.

The `QueryList` also Implements an `iterable` interface. Which means you can iterate over it using `for (var i of items)` or use it with `ngFor` in template `*ngFor="let i of items"`.

Changes can be observed by subscribing to the changes `Observable`.

You can use the following methods & properties.

- `first`: returns the first item in the list.
- `last`: get the last item in the list.
- `length`: get the length of the items.
- `changes`: Is an observable. It emits a new value, whenever the Angular adds, removes or moves the child elements.

It also supports JavaScript array methods like `map()`, `filter()`, `find()`, `reduce()`, `forEach()`, `some()`. etc

ViewChildren Example

In the example below, we have three input elements all using the ngModel directive

We use the `ViewChildren` to get the `QueryList` of all input elements

Finally, we can use the `this.modelRefList.forEach` to loop through the `QueryList` and access each element.

```
1
2 import { ViewChild, Component, ViewChildren, QueryList, AfterViewInit } from '@angular/c
3 import { NgModel } from '@angular/forms';
4
5 @Component({
6   selector: 'app-viewchildren1',
7   template: `
```

```
    <h1>ViewChildren Example</h1>

    <input name="firstName" [(ngModel)]="firstName">
    <input name="midlleName" [(ngModel)]="middleName">
    <input name="lastName" [(ngModel)]="lastName">

    <button (click)="show()">Show</button>

  `,
})

export class ViewChildrenExample1Component {

  firstName;
  middleName;
  lastName;

  @ViewChildren(NgModel) modelRefList: QueryList<NgModel>;

  show() {

    this.modelRefList.forEach(element => {
      console.log(element)
      //console.log(element.value)
    });
  }
}

```

Listening for QueryList Changes

We can subscribe to the changes observable to find if any new elements are added/removed or moved.

In the example below, we have included `ngIf` directive to hide/show the input elements.

We subscribe to the `changes` observable in the component class. Every time we use the `ngIf` to hide or add the component, the `changes` observable emits the latest `QueryList`.

```
1
2  import { ViewChild, Component, ViewChildren, QueryList, AfterViewInit } from '@angular/c
3  import { NgModel } from '@angular/forms';
4
5  @Component({
6    selector: 'app-viewchildren2',
7    template: `
8       <h1>ViewChildren Example</h1>
9
10      <input *ngIf="showFirstName" name="firstName" [(ngModel)]="firstName">
11      <input *ngIf="showMiddleName" name="midlleName" [(ngModel)]="middleName">
12      <input *ngIf="showlastName" name="lastName" [(ngModel)]="lastName">
13
14
15      <input type="checkbox" id="showFirstName" name="showFirstName" [(ngModel)]="sh
16      <input type="checkbox" id="showMiddleName" name="showMiddleName" [(ngModel)]
17      <input type="checkbox" id="showlastName" name="showlastName" [(ngModel)]="sho
18
19      <button (click)="show()">Show</button>
20
21   `,
22
23 })
24
25 export class ViewChildrenExample2Component implements AfterViewInit {
26
27   firstName;
28   middleName;
29   lastName;
30
31   showFirstName=true;
32   showMiddleName=true;
33   showlastName=true;
34
35   @ViewChildren(NgModel) modelRefList: QueryList<NgModel>;
36
37   ngAfterViewInit() {
38
39     this,this.modelRefList.changes
40       .subscribe(data => {
41         console.log(data)
42       }
43     )
44   }
45
46
47   show() {
48     this.modelRefList.forEach(element => {
```

```
49      console.log(element)
50      //console.log(element.value)
51    });
52
53  }
54 }
55
```

Reference

1. ViewChild API
2. ViewChildren API
3. QueryList API
4. ElementRef API
5. ViewContainerRef API

Read More

1. ngModel
2. ElementRef

← HostBinding & HostListener | Angular Tutorial | Elementref →

Related Posts