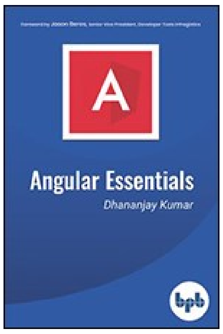


Angular Essentials: The Essential Guide to Learn Angular

by Dhananjay Kumar

BPB Publications. (c) 2019. Copying Prohibited.

Chapter 13: Advanced Components

In this chapter, you will learn about Content Projection and some other ways of component communication. Following topics will be covered in this chapter:

- Content Projection
- Using ViewChild
- Using ContentChild

Content Projection

In Angular, content projection is used to project content in a component. Content projection allows you to insert a shadow DOM in your component. To put it simply, if you want to insert HTML elements or other components in a component, then you do that using the concept of content projection. In Angular, you achieve content projection using `< ng-content>< /ng-content >`. You can make reusable components and scalable application by right use of content projection.

Using Content Projection

To understand content projection, let us consider **GreetComponent** as shown in the *code listing 13.1.*

Code Listing 13.1

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

@Component({
     selector: 'app-greet',
     template: `{{message}}'
})
export class GreetComponent {
        @Input() message: string;
}
```

Using the `@Input()` decorator, you can pass string, numbers or object an to the `GreetComponent`, but what if you need to pass different types of data to the `GreetComponent` such as:

- Inner HTML
- HTML Elements
- Styled HTML
- Another Component

To project styled HTML or another component, content projection is used.

Code Listing 13.2

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

@Component({
     selector: 'app-greet',
     template: `<div>
     <ng-content></ng-content>
     </div>'
})
export class GreetComponent {
      @Input() message: string;
}
```

In the *Code listing 13.2,* you are projecting styled HTML to the `GreetComponent` and you'll get the output as in *figure 13.1:*

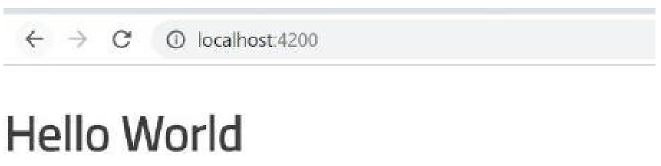


Figure 13.1

This is an example of Single Slot Content Projection. Whatever you pass to the `GreetComponent` will be projected. So, let us pass more than one HTML element to the `GreetComponent` as shown in the *code listing 13.3.*

Code Listing 13.3

```
<div>
   <app-greet>
      <h2>Hello World</h2>
      <button>Say Hello</button>
<p>This is Content Projection</p>
   </app-greet>
</div>
```

Here we are passing three HTML elements to the `GreetComponent,` and all of them will be projected. You will get the output as shown in the *figure 13.2*.

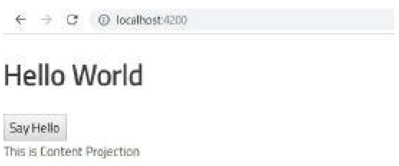
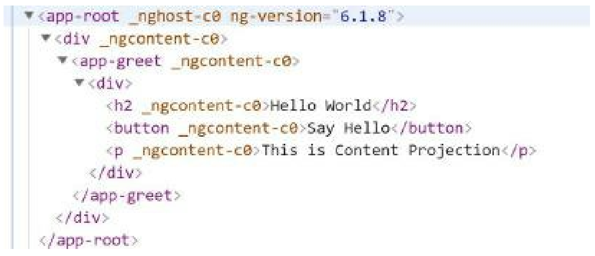


Figure 13.2

In the DOM, you can see that inside the `GreetComponent,` all HTML elements are projected. See *figure 13.3*.



```
▼<app-root _nghost-c0 ng-version="6.1.8">
  ▼<div _ngcontent-c0>
    ▼<app-greet _ngcontent-c0>
      ▼<div>
          <h2 _ngcontent-c0>Hello World</h2>
          <button _ngcontent-c0>Say Hello</button>
          <p _ngcontent-c0>This is Content Projection</p>
        </div>
      </app-greet>
    </div>
  </app-root>
```

Figure 13.3

Multi Slot Projection

You may have a requirement to project elements in multiple slots of the component. Multiple slots mean you have more than one `<ng-content>`. Let us modify `GreetComponent` as shown in the *code listing 13.4:*

Code Listing 13.4

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

@Component({
     selector: 'app-greet',
     template: `<div>
     <h2>{{message}}</h2>
     <ng-content></ng-content>
     <br/>
     <button (click)='sayHello()'>Hello</button>
     <ng-content></ng-content>
     </div>'
})
export class GreetComponent {
     message = 'Greetings';
     sayHello() {
        console.log('hello');
     }
}
```

Here we're using `ng-content` two times. Now, the question is, do we select a particular `ng-content` to project particular element? You can select a particular slot for projection using the `<ng-content>` selector. There are four types of selectors:

1. Project using tag selector
2. Project using class selector
3. Project using id selector
4. Project using attribute selector

You can use the tag selector for multi-slot projection as shown in the *code listing 13.5.*

Code Listing 13.5

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

@Component({
     selector: 'app-greet',
     template: `<div>
     <h2>{{message}}</h2>
     <ng-content select="p"></ng-content>
     <br/>
     <button (click)='sayHello()'>Hello</button>
     <ng-content select="button"></ng-content>
     </div>'
})
export class GreetComponent {
        message = 'Greetings';
        sayHello() {
           console.log('hello');
        }
}
```

Next, you can project content to the `GreetComponent` as shown in the *code listing 13.6:*

Code Listing 13.6

```
<div>
   <app-greet>
        <p>Jason</p>
     <button style ='background:red'>Register me</button>
   </app-greet>
   <app-greet>
      <p>Jason</p>
         <button style ='background:blue'>Register me</
 button>
     </app-greet>
</div>
```

As you can see, we are using the `GreetComponent` twice and projecting different **p** and button elements with different style.

The problem with using tag selectors is that all **p** elements will get projected to the `GreetComponent.` In many scenarios, you may not want that and can use other selectors such as a class selector or an attribute selector, as shown in the *code listing 13.7:*

Code Listing 13.7

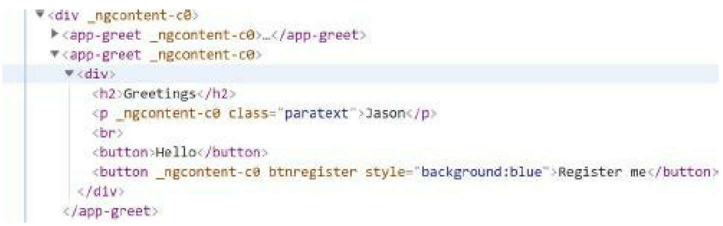
```
import { Component } from '@angular/core';

@Component({
     selector: 'app-greet',
     template: `<div>
     <h2>{{message}}</h2>
     <ng-content select=".paratext"></ng-content>
     <br/>
     <button (click)='sayHello()'>Hello</button>
     <ng-content select="[btnRegister]"></ng-content>
     </div>'
})
export class GreetComponent {
     message = 'Greetings';
     sayHello() {
           console.log('hello');
        }
}
```

Next, you can project content to the **GreetComponent** as shown in the *code listing 13.8:*

Code Listing 13.8

```
<div>
     <app-greet>
        <p class='paratext'>Jason</p>
      <button btnRegister style ='background:red'>Register
me</button>
      </app-greet>
      <app-greet>
        <p class='paratext'>Jason</p>
      <button btnRegister style ='background:blue'>Register
me</button>
     </app-greet>
</div>
```

You'll get the same output as above, however this time you are using the class name and attribute to project the content. When you inspect an element on the DOM, you will find the attribute name and the class name of the projected element as shown in the *figure 13.4*:



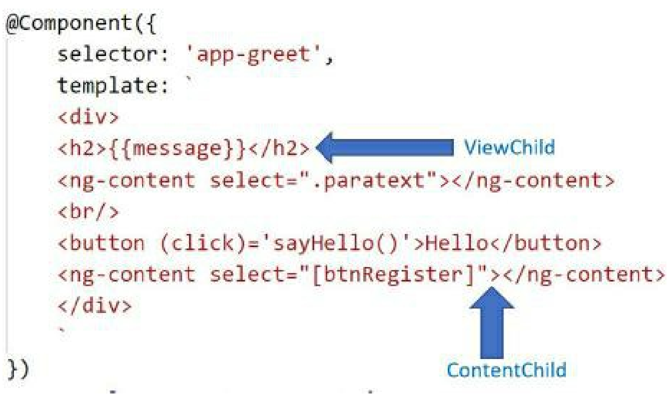
```
▼<div _ngcontent-c0>
  ▶<app-greet _ngcontent-c0>…</app-greet>
  ▼<app-greet _ngcontent-c0>
    ▼<div>
        <h2>Greetings</h2>
        <p _ngcontent-c0 class="paratext">Jason</p>
        <br>
        <button>Hello</button>
        <button _ngcontent-c0 btnregister style="background:blue">Register me</button>
      </div>
    </app-greet>
```

Figure 13.4

Content Projection is very useful to project HTML, to other component.

ViewChild

In the previous section, you learnt about various ways of Component Communication. One of them is **ViewChild** and **ContentChild**. Since now, you know about Content Projection, you should be able to understand **ContentChild** besides **ViewChild**. Essentially **ViewChild** and **ContentChild** are used for component communication in Angular. Therefore, if a parent component wants access of child component then it uses **ViewChild** or **ContentChild**.



```
@Component({
    selector: 'app-greet',
    template: `
    <div>
    <h2>{{message}}</h2>
    <ng-content select=".paratext"></ng-content>
    <br/>
    <button (click)='sayHello()'>Hello</button>
    <ng-content select="[btnRegister]"></ng-content>
    </div>
    `
})
```

Figure 13.5

Any component, directive, or element which is part of a template is **ViewChild** and any component or element which is projected in the template is **`ContentChild`**.

If you want to access the following inside the Parent Component, use **@** **`ViewChild`** decorator of Angular.

- Child Component
- Directive
- DOM Element

`ViewChild` returns the first element that matches the selector. Let us assume that we have a component **`MessageComponent`** as shown in the *code listing 13.9:*

Code Listing 13.9

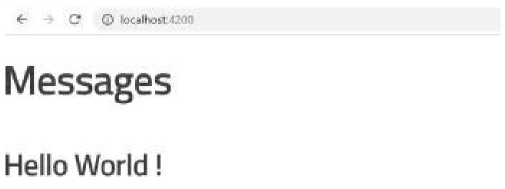
```
import { Component, Input } from '@angular/core';
@Component({
     selector: 'app-message',
     template: `<h2>{{message}}</h2>'
})
export class MessageComponent {
        @Input() message: string;
}
```

We are using **`MessageComponent`** inside **`AppComponent`** as shown in *code listing 13.10:*

Code Listing 13.10

```
import { Component, OnInit } from '@angular/core';
@Component({
        selector: 'app-root',
        template: `
     <div>
     <h1>Messages</h1>
     <app-message [message]='message'></app-message>
     </div>'
})
export class AppComponent implements OnInit {
        message: any;
        ngOnInit() {
         this.message 'Hello World  !';
      }
}
```

On running you will get output as *image 13.6:*



Messages

Hello World !

Figure 13.6

Here, **`MessageComponent`** has become child of **`AppComponent`**. Therefore, we can access it as a **`ViewChild`**. Definition of **`ViewChild`** is:

The Child Element which is located inside the component template,

Here **`MessageComponent`** is located inside template of **`AppComponent`**,

so it can be accessed as **`ViewChild`**. See *code listing 13.11.*

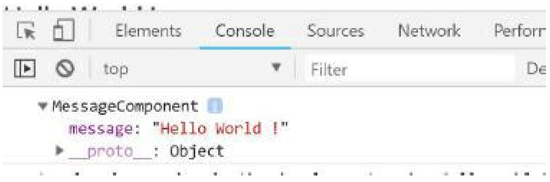
Code Listing 13.11

```
export class AppComponent implements OnInit,
AfterViewInit {
     message: any;
           @ViewChild(MessageComponent) messageViewChild:
MessageComponent;
   ngAfterViewInit() {
     console.log(this.messageViewChild);
   }
   ngOnInit() {
      this.message = 'Hello World !';
     }
}
```

We need to do following tasks to work with **`ViewChild`:**

- Import ViewChild and AfterViewInit from @angular/core
- Implement AfterViewInit life cycle hook to component class
- Create a variable with decorator @ViewChild
- Access that inside ngAfterViewInit life cycle hook

In the output console you will find reference of **`MessageComponent`**, also if you can notice that_proto_of **`MessageComponent`** is set to Object. See *figure 13.7*.



```
▼MessageComponent
   message: "Hello World !"
  ▶__proto__: Object
```

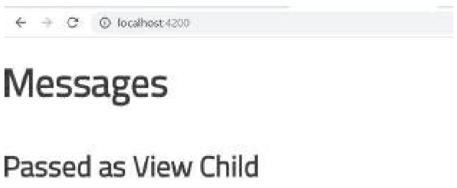
Figure 13.7

Now let us try to change value of **`MessageComponent`** property. See *code listing 13.12.*

Code Listing 13.12

```
   ngAfterViewInit()   {
      console.log(this.messageViewChild);
          this.messageViewChild.message  =   "Passed  as  View
Child';
   }
```

Here we are changing the value of **`viewChild`** property, you will notice that value has been changed and you are getting output as shown in the *figure 13.8:*

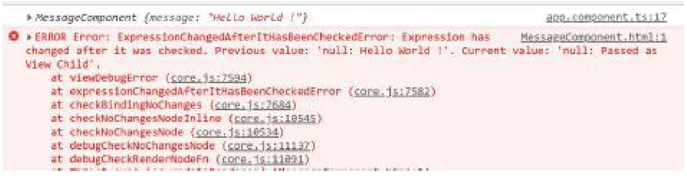


Messages

Passed as View Child

Figure 13.8

However, in the console you will find an error: *Expression has changed after it was last checked***.** See *figure 13.9.*



```
▶MessageComponent {message: "Hello World !"}                    app.component.ts:17
▶ERROR Error: ExpressionChangedAfterItHasBeenCheckedError: Expression has    MessageComponent.html:1
changed after it was checked. Previous value: 'null: Hello World !'. Current value: 'null: Passed as
View Child'.
    at viewDebugError (core.js:7594)
    at expressionChangedAfterItHasBeenCheckedError (core.js:7582)
    at checkBindingNoChanges (core.js:7684)
    at checkNoChangesNodeInline (core.js:10545)
    at checkNoChangesNode (core.js:10534)
    at debugCheckNoChangesNode (core.js:11137)
    at debugCheckRenderNodeFn (core.js:11091)
```

Figure 13.9

This error can be fixed in two ways,

- By changing the ViewChild property in ngAfterContentInit life cycle hook

- Manually calling change detection using ChangeDetectorRef

To fix it in **`ngAfterContentInit`** life cycle hook, you need to implement **`AfterContentInit`** interface as in *code listing 13.13.*

Code Listing 13.13

```
ngAfterContentInit()   {
      this.messageViewChild.message  =   'Passed  as  View
Child';
   }
```

Only problem with this approach is when you work with more than one **`ViewChild`** also known as **`ViewChildren.`** Reference of **`ViewChildren`** is not available in **`ngAfterContentInit`** life cycle hook. In that case, to fix the above error, you will have to use a change detection mechanism. To use the change detection mechanism:

- Import ChangeDetectorRef from @angular/core
- Inject it to the constructor of Component class
- Call detectChanges() method after ViewChild property is changed

You can use manual change detection as shown in *code listing 13.14:*

Code Listing 13.14

```
constructor(private cd:  ChangeDetectorRef)   {}

   ngAfterViewInit()  {
      console.log(this.messageViewChild);
      this.messageViewChild.message =  'Passed as View
Child';
      this.cd.detectChanges();
   }
```

Manually calling change detection will fix *Expression has changed after it was last checked,* error and it can be used with **`ViewChildren`** also.

To understand **`ViewChildren,`** let us consider **`AppComponent`** class created as shown in *code listing 13.15:*

Code Listing 13.15

```
import { Component, OnInit } from '@angular/core';
@Component({
        selector: 'app-root',
        template: `
     <div>
     <h1>Messages</h1>
<app-message *ngFor="let f of messages" [message]='f'></
app-message>
     </div>'
})
export class AppComponent implements OnInit {
        messages: any;
        ngOnInit() {
           this.messages = this.getMessage();
        }
        getMessage() {
           return [
              'Hello India',
              'Which team is winning Super Bowl? ',
              'Have you checked Ignite UI ?',
              'Take your broken heart and make it to the
art'
           ];
        }
}
```

We are using `MessageComponent` inside a `*ngFor` directive hence there are multiple references of `MessageComponent.` We can access it now as `ViewChildren` and `QueryList` as shown in the *code listing 13.16:*

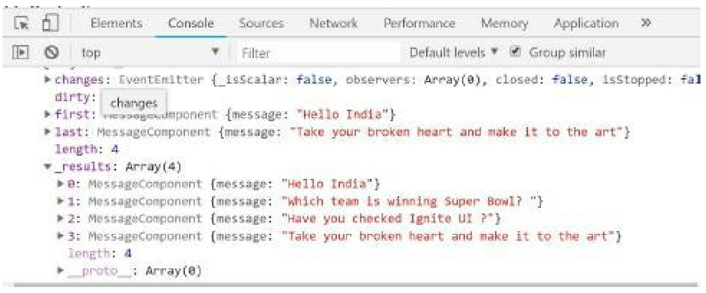
Code Listing 13.16

```
@ViewChildren(MessageComponent)  messageViewChildren:
QueryList<MessageComponent>;
   ngAfterViewInit()   {
      console.log(this.messageViewChildren);
   }
```

To work with **ViewChildren** and **QueryList,** you need to do the following tasks:

- Import ViewChildren , QueryList , AfterViewInit from @angular/ core
- Make reference of ViewChildren with type QueryList
- Access ViewChildren reference in ngAfterViewInit() life cycle hook

In the output, you will get various reference of `MessageComponent` as `ViewChildern` as shown in the *figure 13.10*:



```
▸changes: EventEmitter {_isScalar: false, observers: Array(0), closed: false, isStopped: fal
 dirty:
▸first: MessageComponent {message: "Hello India"}
▸last: MessageComponent {message: "Take your broken heart and make it to the art"}
 length: 4
▾_results: Array(4)
  ▸0: MessageComponent {message: "Hello India"}
  ▸1: MessageComponent {message: "Which team is winning Super Bowl? "}
  ▸2: MessageComponent {message: "Have you checked Ignite UI ?"}
  ▸3: MessageComponent {message: "Take your broken heart and make it to the art"}
   length: 4
  ▸__proto__: Array(0)
```

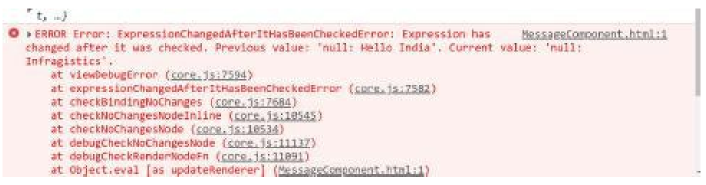
Figure 13.10

Now let us try to update properties of `ViewChildren` as shown in the *code listing 13.17:*

Code Listing 13.17

```
ngAfterViewInit()   {
   console.log(this.messageViewChildren);
   this.messageViewChildren.forEach((item)   => {  item,
message =  'Infragistics';   });
   }
```

As you see, we are iterating through each item of ViewChildren and updating each property. This will update property value but again you will get the error, *Expression has changed after it was last checked* as shown in the *figure 13.11*:



```
 t, …}
▸ERROR Error: ExpressionChangedAfterItHasBeenCheckedError: Expression has     MessageComponent.html:1
changed after it was checked. Previous value: 'null: Hello India'. Current value: 'null:
Infragistics'.
    at viewDebugError (core.js:7594)
    at expressionChangedAfterItHasBeenCheckedError (core.js:7582)
    at checkBindingNoChanges (core.js:7684)
    at checkNoChangesNodeInline (core.js:10545)
    at checkNoChangesNode (core.js:10534)
    at debugCheckNoChangesNode (core.js:11137)
    at debugCheckRenderNodeFn (core.js:11091)
    at Object.eval [as updateRenderer] (MessageComponent.html:1)
```

Figure 13.11

You can again fix it by manually calling change detection like `ViewChild.` Keep in mind that we do not have `ViewChildren` reference available in `AfterContentInit` life cycle hook. You will get undefined in `ngAfterContentInit()` life cycle hook for `ViewChildren` reference as shown in the *code listing 13.18:*

Code Listing 13.18

```
ngAfterContentInit()   {
   console.log(this.messageViewChildren);  // undefined
   }
```

However, you can manually call change detection to fix error: *Expression has changed after it was last checked*

To use a change detection mechanism:

- Import ChangeDetectorRef from @angular/core
- Inject it to the constructor of Component class
- Call detectChanges() method after ViewChild property is changed

You can use a manual change detection like shown in *code listing 13.19:*

Code Listing 13.19

```
@ViewChildren(MessageComponent)  messageViewChildren:
QueryList<MessageComponent>;
      constructor(private cd:  ChangeDetectorRef)    {
      }
      ngAfterViewInit()    {
         console.log(this.messageViewChildren);
         this.messageViewChildren.forEach((item)    => {   item,
message =  'Infragistics';    });
         this.cd.detectChanges();
   }
```

In this way, you can work with **`ViewChild`** and **`ViewChildren`**.

ContentChild

Let us start with understanding about **`ContentChild`**. Any element which is located inside the template, is **`ContentChild`**. To understand it let us consider **`MessageContainerComponent`** as in the *code listing 13.20.*

Code Listing 13.20

```
import { Component } from '@angular/core';
@Component({
     selector: 'app-messagecontainer',
     template: `
     <div>
     <h3>{{greetMessage}}</h3>
     <ng-content select="app-message"></ng-content>
     </div>
     '
})
export class MessageContainerComponent {
     greetMessage = 'Ignite UI Rocks!';
}
```

In this component, we are using Angular Content Projection. You learnt about it in last section.

Any element or component projected inside **`<ng-content>`** becomes a **`ContentChild`**. If you want to access and communicate with **`MessageComponent`** projected inside **`MessageContainerComponent`**, you need to read it as **`ContentChild`**.

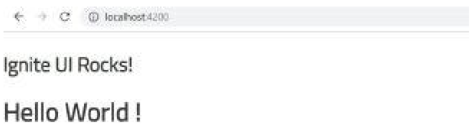
Before we go ahead and learn to use **`ContentChild`**, first see how **`MessageContainerComponent`** is used and **`MessageComponent`** is projected in the *code listing 13.21:*

Code Listing 13.21

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

@Component({
     selector: 'app-root',
     template: `
     <div>
     <app-messagecontainer>
     <app-message [message]='message'></app-message>
     </app-messagecontainer>
     </div>'
})
export class AppComponent implements OnInit {
     message: any;
     ngOnInit() {
        this.message = 'Hello World !';
     }
}
```

As you see in the above listing that in the `AppComponent`, we are using `MessageContainerComponent` and passing `MessageComponent` to be projected inside it. Since `MessageComponent` is used in `MessageContainerComponent` using content projection, it becomes `ContentChild`.

Now, you will get output as shown in *figure 13.12*:



Ignite UI Rocks!

Hello World !

Figure 13.12

Since `MessageComponnet` is projected and is being used inside the template of `MessageContainerComponent`, it can be used as `ContentChild` as shown in the *code listing 13.22*:

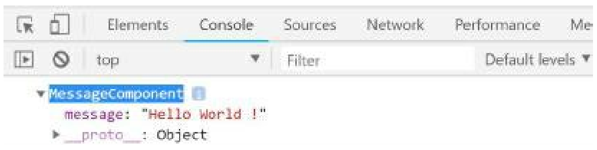
Code Listing 13.22

```
import { Component, ContentChild, AfterContentInit }
from '@angular/core';
import { MessageComponent } from './message.component';
@Component({
     selector: 'app-messagecontainer',
     template: `
     <div>
     <h3>{{greetMessage}}</h3>
     <ng-content select="app-message"></ng-content>
     </div>
     '
})
export class MessageContainerComponent implements
AfterContentInit {
     greetMessage = 'Ignite UI Rocks!';
                 @ContentChild(MessageComponent)
MessageComponnetContentChild: MessageComponent;
        ngAfterContentInit() {
           console.log(this.MessageComponentContentChild);
        }
}
```

We need to do the following tasks:

- Import ContentChild and AfterContentInit from @angular/core
- Implement AfterContentInit life cycle hook to component class
- Create a variable with decorator @ContentChild
- Access that inside ngAfterContentInit life cycle hook

In the output console you will find a reference of **MessageComponent,** also if you can notice that_proto_of **MessageComponent** is set to **Object.** See *figure 13.13*.



```
▾MessageComponent
   message: "Hello World !"
 ▸ __proto__: Object
```

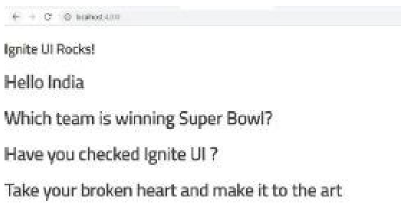
Figure 13.13

You can modify the `ContentChild`property inside `ngAfterContentInit` life cycle hook of the component. Let us assume that there is more than one `MessageComponent` is projected as shown in the *code listing 13.23:*

Code Listing 13.23

```
import { Component, OnInit } from '@angular/core';
@Component({
     selector: 'app-root',
     template: `
     <div>
        <app-messagecontainer>
           <app-message *ngFor='let m of messages'
[message]='m'></app-message>
        </app-messagecontainer>
     </div>'
})
export class AppComponent implements OnInit {
     messages: any;
     ngOnInit() {
        this.messages = this.getMessage();
     }
     getMessage() {
        return [
              'Hello India',
              'Which team is winning Super Bowl? ',
              'Have you checked Ignite UI ?',
              'Take your broken heart and make it to the
art'
           ];
     }
}
```

In the output, you will get many `MessageComponent` projected as *figure 13.14:*



Ignite UI Rocks!

Hello India

Which team is winning Super Bowl?

Have you checked Ignite UI ?

Take your broken heart and make it to the art

Figure 13.14

Now we have more than one `ContentChild`, so we need to access them as `ContentChildren` as shown in the *code listing 13.24:*

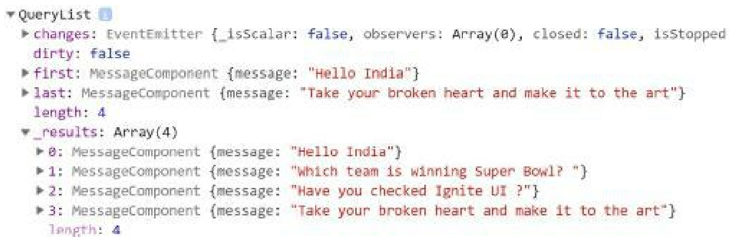
Code Listing 13.24

```
export class MessageContainerComponent implements
AfterContentInit  {
      greetMessage =  'Ignite UI Rocks!';
               @ContentChildren(MessageComponent)
M e s s a g e C o m p o n e n t C o n t e n t C h i l d :
QueryList<MessageComponent>;
     ngAfterContentInit() {
           console.log(this.MessageComponentContentChild);
        }
}
```

To work with `ContentChildren` and `Query-List`, you need to do following tasks:

- Import ContentChildren , QueryList , AfterContentInit from @ angular/core
- Make reference of ContentChildren with type QueryList
- Access ContentChildren reference in ngAfterContentInit() life cycle hook

In the output, you will get various reference of `MessageComponent` as `ContentChildren` as shown in the *figure 13.15:*



```
▼QueryList
  ▶changes: EventEmitter {_isScalar: false, observers: Array(0), closed: false, isStopped
   dirty: false
  ▶first: MessageComponent {message: "Hello India"}
  ▶last: MessageComponent {message: "Take your broken heart and make it to the art"}
   length: 4
  ▼_results: Array(4)
    ▶0: MessageComponent {message: "Hello India"}
    ▶1: MessageComponent {message: "Which team is winning Super Bowl? "}
    ▶2: MessageComponent {message: "Have you checked Ignite UI ?"}
    ▶3: MessageComponent {message: "Take your broken heart and make it to the art"}
     length: 4
```

Figure 13.15

You can query each item in **`ContentChildren`** and modify property as shown in the *code listing 13.25:*

Code Listing 13.25

```
ngAfterContentInit()   {
         this.MessageComponentContentChild.forEach((m)
=> m.message =  'Foo');
      }
}
```

In this way, you can work with **`ContentChildren`** in Angular.

Summary

In this chapter, we learnt about projecting content and different ways of component communication which advance the use of components in Angular. In this chapter you learnt about following topics:

- Content Projection
- ViewChild
- ContentChild
- ViewChildren & ContentChildren