

# ANGULAR UNIVERSITY



PREMIUM QUALITY  
ANGULAR TUTORIALS

[angular-university.io](https://angular-university.io)

# Angular ng-content and Content Projection: A Complete Guide - How To Use ng-content To Improve Component API Design

One of the Angular features that help us the most in building reusable components is Content Projection and `ng-content`.

In this post, we are going to learn how to use this feature to design components that have a very simple but still powerful API - the post is as much about **component design** as it's about content projection!

In order to learn content projection, let's use it build a small component (a **Font Awesome Input Box**). We are going to see how content projection works, when to use it and why, and how it can improve a lot the design of some of our components.

The final component that we are about to build is available [here](#) in the Angular Package Format.

## Table Of Contents

In this post we will cover the following topics:

- What Problem is Content Projection Trying to Solve?
- An example of a component that would benefit from content projection

- Component Design Problem 1 – Supporting all the HTML Properties of an HTML Input
- Component Design Problem 2 – Integration with Angular Forms
- Component Design Problem 3 – Capturing plain browser events of elements inside a template
- Component Design Problem 4 – Custom third party input properties
- The Key Problem With The Initial Design
- Designing the Same Component Using Content Projection
- How To apply styles to elements projected via `ng-content`
- Interacting with Projected Content (inside `ng-content`)
- Multi-Slot Content Projection
- Conclusions

## What Problem is Content Projection Trying to Solve?

Let's start at the beginning: in order to understand content projection, we need to first understand what set of problems is the feature trying to solve.

This is the best way to make sure that we will not misuse the feature as well! So let's implement a small component without using content projection, and see what problems we run into.

## What We Are About To Build

Our Font Awesome Input Box component is designed to look and feel just like a plain HTML Input, except that it has a small Icon inside the

text box.

The icon can be any of the icons available in the [Font Awesome](#) open source version, let's have a look at the component!

## Encapsulating a common HTML Pattern

Adding an icon inside an input box is a very common HTML pattern that makes the input much easier to identify by the user. For example, have a look at the following text boxes:

### Examples of Font Awesome Inputs

Normal Input:

Email:

Password:

Stripe:

Paypal:

### Examples of Material Design Inputs

Lock:

Id Number:

Receipt Number:

Notice that with the presence of both the icon and the text placeholder, we hardly need to have also the field label to the left, which is especially useful on mobile.

## How Does This Component Work?

As we know, normal HTML inputs cannot display an image. But this component does look like a native input, as it also has a blue focus border and supports Tab/ Shift-Tab.

So how does this work? The component is internally implemented using this very common HTML pattern:

- Inside the component, there is a plain HTML input and a icon, wrapped in a DIV
- we have hidden the borders of the plain HTML input, and we have added some similar looking borders to the wrapping DIV
- we have then detected the `focus` and `blur` events in the input field, and we have used those events to add and remove a focus border to the wrapping DIV

So as you can see, we still had to use a couple of tricks to make this component look and behave as a plain HTML input!

## Design Goals

Let's take this very common HTML pattern and make it available as an Angular component. We would like the component to be:

- easily combinable with other Angular components and directives
- have good integration with Angular Forms

With this in mind, let's have a look at an initial attempt of implementing this design *without content projection*, and see what problems we run into.

Let's first see how the component would be used:

```
1
2  @Component({
3    selector: 'app-root',
4    template: `
5
6      <h1>FA Input</h1>
7
8      <fa-input icon="envelope" (value)="onNewValue($event)"></fa-input>
9
10     <fa-input icon="lock" (value)="onNewValue($event)"></fa-input>
11   `
12 })
13 export class AppComponent {
14   onNewValue(val) {
15     console.log(val);
16   }
17 }
18
```

So as we can see, the component is a custom HTML element named `fa-input`, that takes as input an icon name, and outputs the values of the text box.

This is a pretty natural choice for implementing this component, and it's likely the design we could come up on a first attempt.

But there are several major problems with this design, let's have a look at how the component was implemented to understand why, and then see how Content Projection will provide a clean solution for those issues.

## Component API Design - An Initial Attempt

This is the initial implementation of our component:

```

1
2 @Component({
3   selector: 'fa-input',
4   template: `
5     <i class="fa" [ngClass]="classes"></i>
6     <input (focus)="inputFocus = true" (blur)="inputFocus = false"
7       (keyup)="value.emit(input.value)" #input>
8   `,
9   styleUrls: ['./fa-input.component.css']
10 })
11 export class FaInputComponent {
12   @Input() icon: string;
13   @Output() value = new EventEmitter<string>();
14   inputFocus = false;
15
16   get classes() {
17     const cssClasses = {
18       fa: true
19     };
20     cssClasses['fa-' + this.icon] = true;
21     return cssClasses;
22   }
23
24   @HostBinding('class.focus')
25   get focus() {
26     console.log(this.inputFocus);
27     return this.inputFocus;
28   }
29 }
30

```

*Try to guess what is the biggest problem of this design, while we break down this implementation step-by-step.*

As we can see on the template, the main idea is that the component is made of an icon and a plain HTML input.

Before going over the component class implementation, let's have a look at its styling:

```
1
2  :host {
3    border: 1px solid grey;
4  }
5
6  input {
7    border: none;
8    outline: none;
9  }
10
11 :host(.focus) {
12   border: 1px solid blue;
13 }
14
```

Based on these styles, we can see that:

- the HTML input inside the component had its borders and outline removed
- but we added a similar border to the host element, creating the illusion that the component is a plain HTML input
- the focus of the input is simulated by adding a `focus` css class to the host element itself

Let's go back to the component class and see how all the pieces of the puzzle are glued together:

- as part of the public API of the component we have a `icon` string property that defines which icon should be shown (an envelope, a lock, etc.)



- the component has a custom output event named `value`, that emits new values whenever the value of the text input changes
- to implement the focus functionality, we are detecting the focus and blur events on the native html input, and based on that we add or remove the `focus` CSS class on the host element via `@HostBinding`

And this implementation does work! But if we start using this component in our application, we will quickly run into a series of problems. We are going to list here 4 of them, starting with:

## Component Design Problem 1 - Supporting all the HTML Properties of an HTML Input

Our component is meant to be used in place of a plain HTML input, but it does not support any of its standard properties. For example, this is a plain input of type email with autocomplete turned off and a placeholder:

```
1  
2 <input type="email" autocomplete="off" placeholder='Email'>  
3
```

All these standard browser properties are not supported by our component, and these are only a few of the properties that have this problem.

There are currently 31 HTML properties [listed at W3schools](#) for inputs and this does not include all the HTML ARIA [Accessibility attributes](#).

To support all these attributes we would have to do something like this:

```

1
2 @Component({
3   selector: 'fa-input',
4   template: `
5     <i class="fa" [ngClass]="classes"></i>
6     <input (focus)="inputFocus = true" (blur)="inputFocus = false"
7         (keyup)="value.emit(input.value)" #input
8         [placeholder]="placeholder"
9         [type]="type",
10        [autocomplete]="autocomplete">
11   `
12 })
13 export class FaInputComponent {
14   ...
15   @Input() placeholder:string;
16   @Input() type:string;
17   @Input() autocomplete:string;
18   ...
19 }

```

So, in summary, we would have to forward all these component input properties to the internal HTML text box used inside the template.

This would be quite cumbersome but still doable. The issue is that there are other related problems with the current design of this component.

## Component Design Problem 2 - Integration with Angular Forms

Another problem is, what if we would like this input to be part of an Angular Form?

In that case, we would have to also forward all the form properties such as for example `formControlName` to the plain input as well.

## Component Design Problem 3 - Detection of plain browser events

What if we would like to detect a standard browser DOM event on that input? Such as for example the `keydown` event?

This could still be solved by bubbling all the events that don't bubble by default from the input up the component tree, and provide a similarly named event at the level of the component.

This would be quite cumbersome but still doable. But now, we get into a situation for which we don't have a good workaround for.

## Component Design Problem 4 - Custom third party properties

While building forms, third party systems might expect certain HTML custom `data-` properties to be filled in, for scenarios where a full page submission occurs (instead of sending an Ajax request).

This would prove even more troublesome to solve than the situations before, because we don't know upfront the name of those properties.

At this point, we can see that there is a large variety of very common use cases that are not well supported by this design.

So what is the major problem with this component design?

## The Key Problem With This Design

The key issue is that we are hiding the HTML input inside the component template.

Due to that, we are creating a barrier between the external template that knows the custom properties that need to be applied to the input and

the plain HTML input itself.

*and that is the root cause of all the design problems listed above!*

Hiding the input inside the component template causes a whole set of issues, because we need to forward properties and events in and out of the template to support many use cases.

The good news is that using content projection we will be able to **support all these use cases**, and much more.

## Designing the Same Component Using Content Projection

Let's now redesign the API of this component. Instead of hiding the input element inside the component, let's provide it as a *content element* of the component itself:

```
1  @Component({
2    selector: 'app-root',
3    template: `
4
5      <h1>FA Input</h1>
6
7      <fa-input icon="envelope">
8
9        <input type="email" placeholder="Email">
10
11      </fa-input>
12    `})
13  export class AppComponent {
14
15  }
16
```

Notice that we did not provide the form text field as a component input property. Instead, we have added it in the `content` part of the `fa-input` custom HTML tag.

This type of API is actually very common in several standard HTML elements, such as for example select boxes or lists:

```
1
2 <select>
3   <option value="volvo">Volvo</option>
4   <option value="saab">Saab</option>
5   <option value="mercedes">Mercedes</option>
6   <option value="audi">Audi</option>
7 </select>
8
9 <ul>
10  <li>Coffee</li>
11  <li>Tea</li>
12  <li>Milk</li>
13 </ul>
14
```

Angular Core does allow us to do something similar in our components!

We can actually query anything in the content part of the component HTML tag and use it in the internal template as a configuration API, using the `@ContentChild` and `@ContentChildren` decorators.

But we can do more than that, we can also if necessary take anything that is inside the content section, and use it *directly* inside the component.

This means that we can take the HTML input that is inside the `fa-input` content part, and use it directly inside the Font Awesome template, by

projecting it using the `ng-content` Angular Core Directive:



```
1
2 @Component({
3   selector: 'fa-input',
4   template: `
5
6     <i class="fa" [ngClass]="classes"></i>
7
8     <ng-content></ng-content>
9
10  `,
11   styleUrls: ['./fa-input.component.css']
12 })
13 export class FaInputComponent {
14
15   @Input() icon: string;
16
17   get classes() {
18     const cssClasses = {
19       fa: true
20     };
21     cssClasses['fa-' + this.icon] = true;
22     return cssClasses;
23   }
24 }
```

This new version of the component is still incomplete: it does not support yet the focus simulation functionality.

But it solves all the problems listed above, because we have direct access to the HTML input!

Also, this new version has accidentally created a new problem - have a look at the input box now:

# FA Input

 Email	 Password
---	--

Do you notice the duplicate border? It looks the styling that we had put in place to remove the border from the HTML input is not working anymore!

Also, the focus functionality is missing.

It looks like despite the several problems that we have solved by using `ng-content`, we still have at this point two major questions about it:

- how to style projected content?
- how to interact with projected content?

## How To Apply styles to elements projected via ng-content

Let's start by understanding why the styles we had in place no longer work with `ng-content`. The current input styles look like this, and we can find them inside the `fa-input.component.css` file:

```
1
2  input {
3    border: none;
4    outline: none;
5  }
6
```

Here is why this does not work anymore: because these styles sit inside a file linked to the component, they will have applied at startup time an attribute that is unique to all the HTML elements inside that particular component template:

```
1
2  input[_ngcontent-c0] {
3    border: none;
4    outline: none;
5  }
6
```

So what is this strange identifier? Let's have a look at the runtime HTML on the page for our component:

```
1
2  <fa-input icon="envelope" _ngghost-c0="">
3
4    <i _ngcontent-c0="" class="fa fa-envelope"></i>
5
6    <input placeholder="Email" type="email">
7
8  </fa-input>
9
```

This is a simplified version of the HTML, that helps better understand what is going on:

- we can see that each element of `fa-input` template, in this case, the icon tag gets applied a `_ngcontent-c0` attribute, which is unique to this component
- all the styles of the component are then scoped to only elements containing this attribute



- which means that the styles of the component will **NOT** affect the projected input, because if you notice it does not have the special attribute `_ngcontent-c0`
- this is normal because the input comes from another template other than the `fa-input` template

## Styling projected content

In order to style the projected input and remove the double border, we need to change the styles to something like this:

```
1
2 :host /deep/ input {
3   border: none;
4   outline: none;
5 }
6
```

So how do these new styles work? Let's break this down:

- we are prefixing the style with the `:host` selector, meaning that the styles will be applied only inside this component
- we are then applying the `/deep/` modifier, which means that the style will no longer be scoped only to HTML elements of this particular component, but it will also affect any descendant elements

To see how this works in practice, this is the actual CSS at runtime:

```
1
2 [_nghost-c0] input {
3   border: none;
4   outline: none;
```

```
5 }  
6
```

So as we can see, the styles are still scoped to the component only, but they will leak through to any inputs inside the component at runtime: including the projected HTML input!

So this shows how to style projected content if necessary. Now let's fix the second part of the puzzle: how to interact with projected content, and simulate the focus functionality?

## Interacting with Projected Content inside ng-content

To be able to simulate the focus functionality, we need the `fa-input` input component to know that the projected input focus was either activated or blurred.

We cannot interact with the `ng-content` tag, and for example define event listeners on top of it.

Instead, the best way to interact with the projected input is to start by applying a new separate directive to the input.

Let's then create a directive named `inputRef`, and apply it to the HTML Input:

```
1  
2 <h1>FA Input</h1>  
3  
4 <fa-input icon="envelope">  
5  
6 <input inputRef type="email" placeholder="Email">
```

7

8 `</fa-input>`

We will take the opportunity to use that same directive to track if the input has the focus or not:

```
1
2 @Directive({
3   selector: '[inputRef]'
4 })
5 export class InputRefDirective {
6   focus = false;
7
8   @HostListener("focus")
9   onFocus() {
10     this.focus = true;
11   }
12
13   @HostListener("blur")
14   onBlur() {
15     this.focus = false;
16   }
17 }
18
```

Here is what is going on in this directive:

- we have defined a focus property, that will be true or false depending if the native Input to which the directive was applied has the focus or not
- the native focus and blur DOM events are being detected using the `@HostListener` decorator

We can now use this directive and have it injected inside the Font Awesome Input component, effectively allowing us to interact with projected content!

Let's see what that would look like:

```
1
2 @Component({
3   selector: 'fa-input',
4   template: `
5     <i class="fa" [ngClass]="classes"></i>
6     <ng-content></ng-content>
7   `,
8   styleUrls: ['./fa-input.component.css']
9 })
10 export class FaInputComponent {
11
12   @Input() icon: string;
13
14   @ContentChild(InputRefDirective)
15   input: InputRefDirective;
16
17   @HostBinding("class.focus")
18   get focus() {
19     return this.input ? this.input.focus : false;
20   }
21
22   get classes() {
23     const cssClasses = {
24       fa: true
25     };
26     cssClasses['fa-' + this.icon] = true;
27     return cssClasses;
28   }
29 }
30
```

As we can see, we have simply used the `@ContentChild` decorator to inject the `inputRef` directive inside the Font Awesome Input component.

Then using this directive and the boolean `focus` property, we have then set the CSS class named `focus` on the host elements using the `@HostBinding` decorator.

With this new implementation in place, we have now a fully functioning component, that is super-simple to use and supports implicitly all the HTML input properties, accessibility, third party properties and Angular Forms - all of that made possible by the use of content projection.

In the current implementation, we have so far been projecting the whole content of `fa-input`. But what if we would like to project only part of it?

## Multi-Slot Content Projection

Let's now say that we would like to project not only the HTML input itself, but also the icon inside the input. This is also possible using content projection.

In the content part of the the `fa-input` tag we can put multiple types of content, for example:

```
1
2 <fa-input icon="envelope">
3
4   <i class="fa fa-envelope"></i>
5
6   <input inputRef type="email" placeholder="Email">
7
8 </fa-input>
9
```

We can then consume the different types of content available inside the `fa-input` tag, by using the `select` property of `ng-content`:

```

1
2 @Component({
3   selector: 'fa-input',
4   template: `
5
6     <ng-content select="i"></ng-content>
7
8     <ng-content select="input"></ng-content>
9
10  `})
11 export class FaInputComponent {
12   ...
13 }
14

```

These two selectors are looking for a particular element type (an input or an icon tag), but we could also look for an element with a given CSS class, and combine multiple selectors.

For example, this selector would look for inputs with a given CSS class named `test-class`:

```

1
2 <ng-content select="input.test-class"></ng-content>
3

```

It's also possible to capture content that does not match any selector. For example, this would also inject the input element into the template:

```

1
2 @Component({
3   selector: 'fa-input',
4   template: `
5
6     <ng-content select="i"></ng-content>
7

```

```
8      <ng-content></ng-content>
9
10    `})
11    export class FaInputComponent {
12      ...
13    }
14
15
```

In this context, the `<ng-content></ng-content>` tag without any selector would fetch all the content that *did not match any of the other selectors*:

In this case, that would mean all the content which is not an icon tag, which would be the HTML Input.

## Conclusions

As we have seen, it's as important to know how the `ng-content` core directive works, as to know the typical scenarios and use cases on which we would like to use it.

The `ng-content` core directive allows for component designs where certain internal details of the template are not hidden inside the component but instead are provided as an input, which in certain cases really simplifies the design.