Attribute Directives

An **Attribute** directive changes the appearance or behavior of a DOM element.

Try the Attribute Directive example / download example.

Directives overview

There are three kinds of directives in Angular:

1. Components—directives with a template.
2. Structural directives—change the DOM layout by adding and removing DOM elements.
3. Attribute directives—change the appearance or behavior of an element, component, or another directive.

Components are the most common of the three directives. You saw a component for the first time in the Getting Started tutorial.

Structural Directives change the structure of the view. Two examples are NgFor and NgIf. Learn about them in the Structural Directives guide.

Attribute directives are used as attributes of elements. The built-in NgStyle directive in the Template Syntax guide, for example, can change several element styles at the same time.

Build a simple attribute directive

An attribute directive minimally requires building a controller class annotated with `@Directive`, which specifies the selector that identifies the attribute. The controller class implements the desired directive behavior.

This page demonstrates building a simple *appHighlight* attribute directive to set an element's background color when the user hovers over that element. You can apply it like this:

src/app/app.component.html (applied)

```
<p appHighlight>Highlight me!</p>
```

Please note that directives *do not* support namespaces.

src/app/app.component.avoid.html (unsupported)

```
<p app:Highlight>This is invalid</p>
```

Write the directive code

Create the directive class file in a terminal window with the CLI command `ng generate directive`.

```
ng generate directive highlight
```

The CLI creates `src/app/highlight.directive.ts`, a corresponding test file `src/app/highlight.directive.spec.ts`, and *declares* the directive class in the root `AppModule`.

> *Directives* must be declared in Angular Modules in the same manner as *components*.

The generated `src/app/highlight.directive.ts` is as follows:

src/app/highlight.directive.ts

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

@Directive({
  selector: '[appHighlight]'
})
export class HighlightDirective {
  constructor() { }
}
```

The imported `Directive` symbol provides Angular the `@Directive` decorator.

The `@Directive` decorator's lone configuration property specifies the directive's CSS attribute selector, `[appHighlight]`.

It's the brackets (`[]`) that make it an attribute selector. Angular locates each element in the template that has an attribute named `appHighlight` and applies the logic of this directive to that element.

The *attribute selector* pattern explains the name of this kind of directive.

> ### Why not "highlight"?
>
> Though *highlight* would be a more concise selector than *appHighlight* and it would work, the best practice is to prefix selector names to ensure they don't conflict with standard HTML attributes. This also reduces the risk of colliding with third-party directive names. The CLI added the `app` prefix for you.
>
> Make sure you do **not** prefix the `highlight` directive name with `ng` because that prefix is reserved for Angular and using it could cause bugs that are difficult to diagnose.

After the `@Directive` metadata comes the directive's controller class, called `HighlightDirective`, which contains the (currently empty) logic for the directive. Exporting `HighlightDirective` makes the directive accessible.

Now edit the generated `src/app/highlight.directive.ts` to look as follows:

src/app/highlight.directive.ts

```
import { Directive, ElementRef } from '@angular/core';

@Directive({
  selector: '[appHighlight]'
})
export class HighlightDirective {
    constructor(el: ElementRef) {
       el.nativeElement.style.backgroundColor = 'yellow';
    }
}
```

The `import` statement specifies an additional `ElementRef` symbol from the Angular `core` library:

You use the `ElementRef` in the directive's constructor to inject a reference to the host DOM element, the element to which you applied `appHighlight`.

`ElementRef` grants direct access to the host DOM element through its `nativeElement` property.

This first implementation sets the background color of the host element to yellow.

Apply the attribute directive

To use the new `HighlightDirective`, add a paragraph (`<p>`) element to the template of the root `AppComponent` and apply the directive as an attribute.

src/app/app.component.html

```
<p appHighlight>Highlight me!</p>
```

Now run the application to see the `HighlightDirective` in action.

```
ng serve
```

To summarize, Angular found the `appHighlight` attribute on the **host** `<p>` element. It created an instance of the `HighlightDirective` class and injected a reference to the `<p>` element into the directive's constructor which sets the `<p>` element's background style to yellow.

Respond to user-initiated events

Currently, `appHighlight` simply sets an element color. The directive could be more dynamic. It could detect when the user mouses into or out of the element and respond by setting or clearing the highlight color.

Begin by adding `HostListener` to the list of imported symbols.

src/app/highlight.directive.ts (imports)

```
import { Directive, ElementRef, HostListener } from '@angular/core';
```

Then add two eventhandlers that respond when the mouse enters or leaves, each adorned by the `HostListener` decorator.

src/app/highlight.directive.ts (mouse-methods)

```
@HostListener('mouseenter') onMouseEnter() {
  this.highlight('yellow');
}

@HostListener('mouseleave') onMouseLeave() {
  this.highlight(null);
}

private highlight(color: string) {
  this.el.nativeElement.style.backgroundColor = color;
}
```

The `@HostListener` decorator lets you subscribe to events of the DOM element that hosts an attribute directive, the `<p>` in this case.

> Of course you could reach into the DOM with standard JavaScript and attach event listeners manually. There are at least three problems with *that* approach:
>
> 1. You have to write the listeners correctly.
> 2. The code must *detach* the listener when the directive is destroyed to avoid memory leaks.
> 3. Talking to DOM API directly isn't a best practice.

The handlers delegate to a helper method that sets the color on the host DOM element, `el`.

The helper method, `highlight`, was extracted from the constructor. The revised constructor simply declares the injected `el: ElementRef`.

src/app/highlight.directive.ts (constructor)

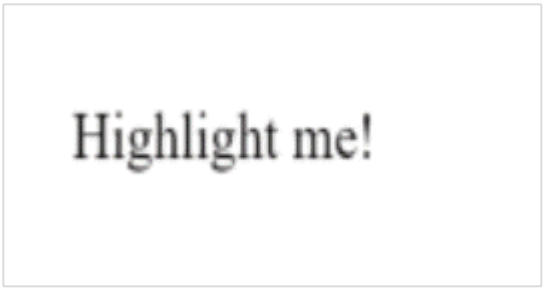
```
constructor(private el: ElementRef) { }
```

Here's the updated directive in full:

src/app/highlight.directive.ts

```
import { Directive, ElementRef, HostListener } from '@angular/core';

@Directive({
  selector: '[appHighlight]'
})
export class HighlightDirective {
  constructor(private el: ElementRef) { }

  @HostListener('mouseenter') onMouseEnter() {
    this.highlight('yellow');
  }

  @HostListener('mouseleave') onMouseLeave() {
    this.highlight(null);
  }

  private highlight(color: string) {
    this.el.nativeElement.style.backgroundColor = color;
  }
}
```

Run the app and confirm that the background color appears when the mouse hovers over the `p` and disappears as it moves out.



Pass values into the directive with an *@Input* data binding

Currently the highlight color is hard-coded *within* the directive. That's inflexible. In this section, you give the developer the power to set the highlight color while applying the directive.

Begin by adding `Input` to the list of symbols imported from `@angular/core`.

src/app/highlight.directive.ts (imports)

```
import { Directive, ElementRef, HostListener, Input } from '@angular/core';
```

Add a `highlightColor` property to the directive class like this:

src/app/highlight.directive.ts (highlightColor)

```
@Input() highlightColor: string;
```

Binding to an *@Input* property

Notice the `@Input` decorator. It adds metadata to the class that makes the directive's `highlightColor` property available for binding.

It's called an *input* property because data flows from the binding expression *into* the directive. Without that input metadata, Angular rejects the binding; see below for more about that.

Try it by adding the following directive binding variations to the `AppComponent` template:

src/app/app.component.html (excerpt)

```
<p appHighlight highlightColor="yellow">Highlighted in yellow</p>
<p appHighlight [highlightColor]="'orange'">Highlighted in orange</p>
```

Add a `color` property to the `AppComponent`.

src/app/app.component.ts (class)

```
export class AppComponent {
  color = 'yellow';
}
```

Let it control the highlight color with a property binding.

src/app/app.component.html (excerpt)

```
<p appHighlight [highlightColor]="color">Highlighted with parent component's
color</p>
```

That's good, but it would be nice to *simultaneously* apply the directive and set the color *in the same attribute* like this.

src/app/app.component.html (color)

```
<p [appHighlight]="color">Highlight me!</p>
```

The `[appHighlight]` attribute binding both applies the highlighting directive to the `<p>` element and sets the directive's highlight color with a property binding. You're re-using the directive's attribute selector (`[appHighlight]`) to do both jobs. That's a crisp, compact syntax.

You'll have to rename the directive's `highlightColor` property to `appHighlight` because that's now the color property binding name.

src/app/highlight.directive.ts (renamed to match directive selector)

```
@Input() appHighlight: string;
```

This is disagreeable. The word, `appHighlight`, is a terrible property name and it doesn't convey the property's intent.

Bind to an *@Input* alias

Fortunately you can name the directive property whatever you want *and **alias it*** for binding purposes.

Restore the original property name and specify the selector as the alias in the argument to `@Input`.

src/app/highlight.directive.ts (color property with alias)

```
@Input('appHighlight') highlightColor: string;
```

Inside the directive the property is known as `highlightColor`. *Outside* the directive, where you bind to it, it's known as `appHighlight`.

You get the best of both worlds: the property name you want and the binding syntax you want:

src/app/app.component.html (color)

```
<p [appHighlight]="color">Highlight me!</p>
```

Now that you're binding via the alias to the `highlightColor`, modify the `onMouseEnter()` method to use that property. If someone neglects to bind to `appHighlightColor`, highlight the host element in red:

src/app/highlight.directive.ts (mouse enter)

```
@HostListener('mouseenter') onMouseEnter() {
  this.highlight(this.highlightColor || 'red');
}
```

Here's the latest version of the directive class.

src/app/highlight.directive.ts (excerpt)

```
import { Directive, ElementRef, HostListener, Input } from '@angular/core';

@Directive({
  selector: '[appHighlight]'
})
export class HighlightDirective {

  constructor(private el: ElementRef) { }

  @Input('appHighlight') highlightColor: string;

  @HostListener('mouseenter') onMouseEnter() {
    this.highlight(this.highlightColor || 'red');
  }

  @HostListener('mouseleave') onMouseLeave() {
    this.highlight(null);
  }

  private highlight(color: string) {
    this.el.nativeElement.style.backgroundColor = color;
  }
}
```

Write a harness to try it

It may be difficult to imagine how this directive actually works. In this section, you'll turn `AppComponent` into a harness that lets you pick the highlight color with a radio button and bind your color choice to the directive.

Update `app.component.html` as follows:

src/app/app.component.html (v2)

```
<h1>My First Attribute Directive</h1>

<h4>Pick a highlight color</h4>
<div>
  <input type="radio" name="colors" (click)="color='lightgreen'">Green
  <input type="radio" name="colors" (click)="color='yellow'">Yellow
  <input type="radio" name="colors" (click)="color='cyan'">Cyan
</div>
<p [appHighlight]="color">Highlight me!</p>
```

Revise the `AppComponent.color` so that it has no initial value.

src/app/app.component.ts (class)

```
export class AppComponent {
  color: string;
}
```

Here are the harness and directive in action.

Bind to a second property

This highlight directive has a single customizable property. In a real app, it may need more.

At the moment, the default color—the color that prevails until the user picks a highlight color—is hard-coded as "red". Let the template developer set the default color.

Add a second **input** property to `HighlightDirective` called `defaultColor`:

src/app/highlight.directive.ts (defaultColor)

```
@Input() defaultColor: string;
```

Revise the directive's `onMouseEnter` so that it first tries to highlight with the `highlightColor`, then with the `defaultColor`, and falls back to "red" if both properties are undefined.

src/app/highlight.directive.ts (mouse-enter)

```
@HostListener('mouseenter') onMouseEnter() {
  this.highlight(this.highlightColor || this.defaultColor || 'red');
}
```

How do you bind to a second property when you're already binding to the `appHighlight` attribute name?

As with components, you can add as many directive property bindings as you need by stringing them along in the template. The developer should be able to write the following template HTML to both bind to the `AppComponent.color` and fall back to "violet" as the default color.

src/app/app.component.html (defaultColor)

```
<p [appHighlight]="color" defaultColor="violet">
  Highlight me too!
</p>
```

Angular knows that the `defaultColor` binding belongs to the `HighlightDirective` because you made it *public* with the `@Input` decorator.

Here's how the harness should work when you're done coding.

Summary

This page covered how to:

- Build an **attribute directive** that modifies the behavior of an element.
- Apply the directive to an element in a template.
- Respond to **events** that change the directive's behavior.
- **Bind** values to the directive.

The final source code follows:

app/app.component.ts *app/app.component.html* *app/highlight.directive.ts*

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html'
})
export class AppComponent {
  color: string;
}
```

You can also experience and download the Attribute Directive example / download example.

Appendix: Why add *@Input*?

In this demo, the `highlightColor` property is an ***input*** property of the `HighlightDirective`. You've seen it applied without an alias:

src/app/highlight.directive.ts (color)

```
@Input() highlightColor: string;
```

You've seen it with an alias:

src/app/highlight.directive.ts (color)

```
@Input('appHighlight') highlightColor: string;
```

Either way, the `@Input` decorator tells Angular that this property is *public* and available for binding by a parent component. Without `@Input`, Angular refuses to bind to the property.

You've bound template HTML to component properties before and never used `@Input`. What's different?

The difference is a matter of trust. Angular treats a component's template as *belonging* to the component. The component and its template trust each other implicitly. Therefore, the component's own template may bind to *any* property of that component, with or without the `@Input` decorator.

But a component or directive shouldn't blindly trust *other* components and directives. The properties of a component or directive are hidden from binding by default. They are *private* from an Angular binding perspective. When adorned with the `@Input` decorator, the property becomes *public* from an Angular binding perspective. Only then can it be bound by some other component or directive.

You can tell if `@Input` is needed by the position of the property name in a binding.

- When it appears in the template expression to the ***right*** of the equals (=), it belongs to the template's component and does not require the `@Input` decorator.
- When it appears in **square brackets** ([ ]) to the **left** of the equals (=), the property belongs to some *other* component or directive; that property must be adorned with the `@Input` decorator.

Now apply that reasoning to the following example:

src/app/app.component.html (color)

```
<p [appHighlight]="color">Highlight me!</p>
```

- The `color` property in the expression on the right belongs to the template's component. The template and its component trust each other. The `color` property doesn't require the `@Input` decorator.
- The `appHighlight` property on the left refers to an *aliased* property of the `HighlightDirective`, not a property of the template's component. There are trust issues. Therefore, the directive property must carry the `@Input` decorator.