# Angular Essentials

**ORIGINAL BY GIL FINK,** CEO AND SENIOR CONSULTANT, SPARXYS
**WRITTEN BY DHANANJAY KUMAR,** DEVELOPER EVANGELIST, INFRAGISTICS

CONTENTS





## ANGULAR ESSENTIALS

Angular is a widely used web application platform and framework created and maintained by Google. It serves as a total rewrite to AngularJS, and the "Angular" name is meant to include all versions of the framework starting from 2 and up.

TypeScript is the core of Angular, being the language upon which Angular is written. As such, Angular implements major and core functionalities as TypeScript libraries while building client applications with additional HTML.

For a variety of reasons, Angular has grown in popularity with developers. It lends itself to maintenance ease with its component and class-based system, modular building, hierarchical structure, and simple, declarative templates. Furthermore, its cross-platform capabilities are advantageous to enterprise and SMB developers, including its speed with server-side rendering.

This Refcard will go over the essential pieces of Angular and the main concepts behind working with the ever-growing platform for web-based applications.

## HOW IS ANGULAR DIFFERENT FROM ANGULARJS?

In the past, you might have worked with or learned about AngularJS. There are a few main differences between the two that you need to know about:

- **Modularity**: More of Angular's core functionalities have moved to modules.
- **Hierarchy**: Angular has an architecture built around a hierarchy of components.
- **Syntax**: Angular has a different expression syntax for event and property binding.
- **Dynamic loading**: Angular will load libraries into memory at run-time, retrieve and execute functions, and then unload the library from memory.
- **Iterative callbacks**: Using RxJS, Angular makes it easier to compose asynchronous or callback-based code.
- **Asynchronous template compilation**: Angular, without controllers and the concept of "scope," makes it easier to pause template rendering and compile templates to generate the defined code.

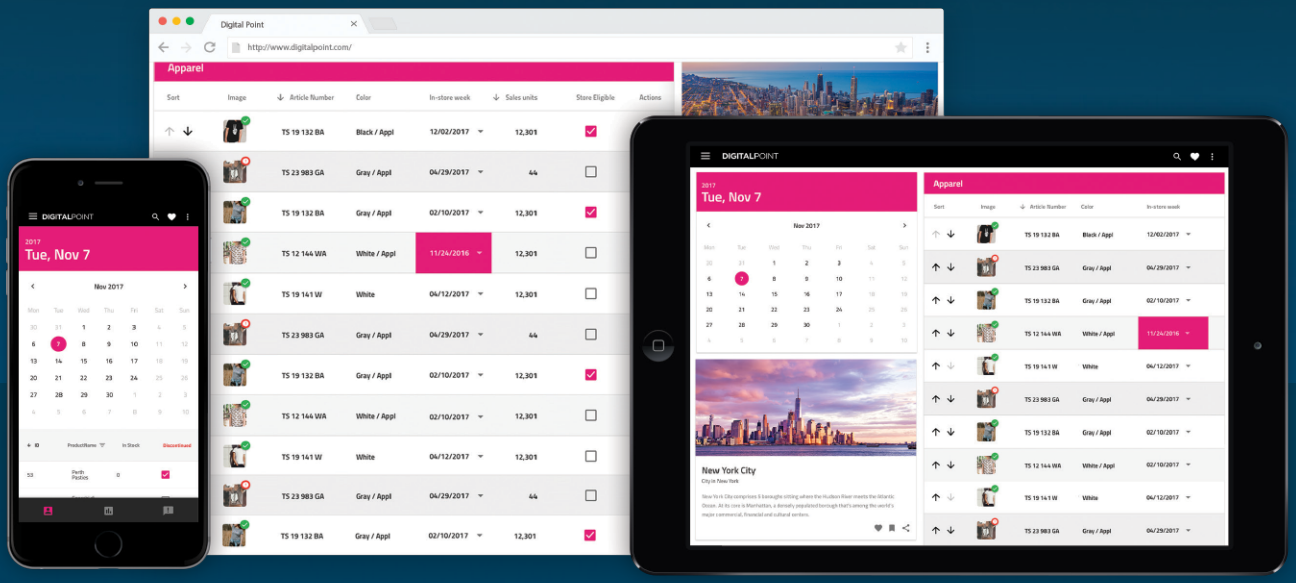
INFRAGISTICS

# Ignite UI for Angular

## Specialized Angular UI Components for Modern Experiences Across Mobile & Desktop

Open Source | True Angular | Sketch UI Kit | Material Design | A11y

**Download Your Free Trial**




### Ignite UI for Angular includes:

**Grids & Lists**
Data Grid
List View
Combo

**Charts**
Category Chart
Financial Chart

**Gauges**
Bullet Graph
Linear Gauge
Radial Gauge

**Styling & Themes**
Theming
Shadows

**Data Entry & Display**
Drop Down
Buttons
Button Group
Checkbox
Switch
Radio Button
Label
Input
Badge

**Data Entry & Display**
Icon
Mask Directive
Input Groups
Linear Progress
Circular Progress
Virtualization
Chip

**Interactions**
Dialog Window
Slider
Ripple
Toggle
Overlay

**Menus**
Navigation Drawer
Navbar

**Services**
CSV Exporter
Excel Exporter

**Layouts**
Layout Manager
Carousel
Bottom Navigation
Card
Avatar
Tabs

**Notifications**
Snack Bar
Toast

**Scheduling**
Calendar
Date Picker
Time Picker

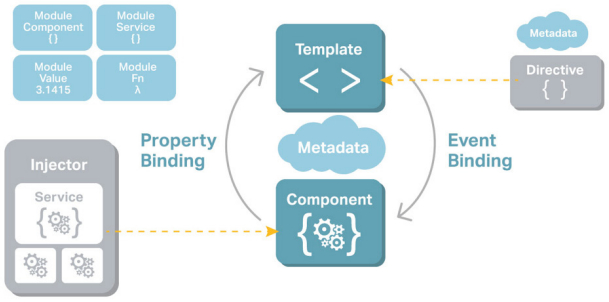
- **TypeScript**: Angular includes ES6 and its superset, TypeScript.

## ANGULAR'S BASIC ARCHITECTURE

Here's a brief overview of the architecture involved and the building blocks that I'll cover in this piece:

- **NgModules**: Declares a compilation context for a set of components that is dedicated to an application domain, a workflow, or a related set of capabilities.
- **Components**: Defines a class that contains application data and logic and works with an HTML template that defines a view.
- **Template**: Combines HTML with Angular markup that can modify HTML elements before they're displayed.
- **Directive**: Attaches custom behavior to elements in the DOM.
- **Two-way data binding**: Coordinates the parts of a template with the parts of a component.
- **Services**: Typically, a class used to increase modularity and reusability with a narrow and well-defined purpose.
- **Dependency injection**: Provides components with needed services and gives access to a service class.
- **Routing**: Defines a navigation path among the different application states lets you view application hierarchies.

This diagram best represents the relationship between the building blocks:




## SETTING UP THE ENVIRONMENT

In order to set up the environment, you should start by downloading Angular with the Angular CLI tool. If you have a machine that doesn't have Node.js and npm installed, make sure to download and install them here.

Then, you'll run a global install of the Angular CLI:

```
npm install --g \@angular/cli
```

### NGMODULES

NgModules are excellent for organizing related items, and they function to configure both the injector and compiler. You'll find that that the NgModule is named as such, since it's a class that is marked by the `@NgModule` decorator.

This decorator had the information on how to compile a component's template and how to create an injector at runtime, all within a metadata object. As you could guess, `@NgModule` serves to identify and bridge the gap between both its own directives, components, and pipes, and external components that rely on these pieces.

The `exports` property also makes some of the module's make-up public, ensuring that the external components can effectively use them.

As a last bit, `@NgModule` also adds services providers to the application dependency injectors, foundationally making the application more adaptable.

### ANGULAR BOOTSTRAPPING

Understanding that an `NgModule` describes how an application's parts are to work and fit together, it makes sense that every Angular application has at least one Angular module. This core module functions as the "root" module for the application, and the one that you would bootstrap to launch the application.

There are three basic components to the root module, which we'll discuss briefly.

### DECLARATIONS ARRAY

Components used in an `NgModule` need to be added to the declarations array as a way to tell Angular that these specific components belong to this specific module. On top of this, since only declarables can be added to the declarations array, you'll find that the array will be populated with various components, directives, and pipes.

Keep in mind that declarables can only belong to one module.

### IMPORTS ARRAY

The Imports array contains all dependent modules.

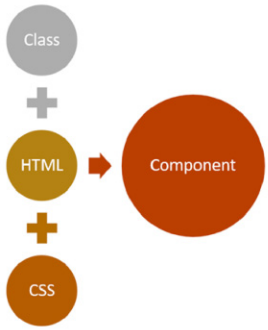
### PROVIDERS ARRAY

The Providers array contains all Services dependencies.

## COMPONENTS

In Angular applications, what you see in the browser (or elsewhere) is a component. A component consists of the following parts:

1. A TypeScript class called the Component class
2. An HTML file called the template of the component
3. An optional CSS file for the styling of the component




A component is a type of directive with its own template. Whatever you see in an Angular application is a component.

## CREATING A COMPONENT

You can use an Angular CLI command to generate a component as shown below:

```
ng generate component Product
```

This command will generate PoductComponent as shown below:

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

@Component({
  selector: 'app-product',
  templateUrl: './product.component.html',
styleUrls: ['./product.component.scss']
})
export class ProductComponent implements OnInit {

  constructor() { }

  ngOnInit() {
  }

}
```

A component is a class decorated with the `@Component` decorator. There are mainly four steps to create a component:

1. Create a class and export it. This class will contain data and the logic.
2. Decorate the class with `@component` metadata. Metadata describes the component and sets the value for different properties.
3. Import the required libraries and modules to create the component.
4. Create a template of the component and optionally style of the component.

As you can see, the generated `ProductComponent` consists of:

- A class to hold data and the logic.
- HTML template and styles to display data in the app. It is also called a view and is seen by the user on the screen to interact.
- Metadata that defines the behavior of a component. Component metadata is applied to the class using the `@Component` decorator. Different behavior of the component can be passed as properties of the object, which is an input parameter of the `@Component` decorator.

## COMPONENT METADATA

The `@Component` decorator decorates a class as a component. It is a function that takes an object as a parameter. In the `@Component` decorator, we can set the values of different properties to set the behavior of the component. The most used properties are as follows:

- `template`
- `templateUrl`
- `Providers`
- `styles`
- `styleUrls`
- `selector`
- `encapsulation`
- `changeDetection`
- `animations`
- `viewProviders`

Apart from the above-mentioned properties, there are also other properties. Let's look into these important properties one by one.

### *TEMPLATE AND TEMPLATEURL*

A template is the part of the component that gets rendered on the page. We can create a template in two ways:

1. **Inline template**: `template` property
2. **Template in an external file**: `templateUrl` property

To create an inline template, the "tilt" is a symbol used to create multiple lines in the template. A single-line inline template can be created using either single quotes or double quotes. For the inline template, set the value of template property. A complex template can be created in an external HTML file and can be set using the `templateUrl` property.

### *SELECTOR*

A component can be used using the selector. In the above example, the selector property is set to `<app-product>`. We can use the component on the template of other components using its selector.

### *STYLES AND STYLEURLS*

A component can have its own styles or it can refer to various other external style sheets. To work with styles, `@Component` metadata has `styles` and `styleUrls` properties. We can create inline styles by

setting the value of the `styles` property. We can set external style using `styleUrls` property.

#### *PROVIDERS*

To inject a service in a component, you pass that to the providers array. Component metadata has an array type property called the `provider`. In the providers, we pass a list of services being injected in the component. We will cover this in detail in further sections.

#### *CHANGEDETECTION*

This property determines how the change detector will work for the component. We set the `ChangeDetectionStrategy` of the component in the property. There are two possible values:

1. `Default`
2. `onPush`

We will cover this property in detail in further sections.

#### *ENCAPSULATION*

This property determines whether Angular will create a shadow DOM for a component. It determines the `ViewEncapsulation` mode of the component. There are four possible values:

1. Emulated (this is the default)
2. Native
3. None
4. ShadowDom

### TEMPLATE

When you generate a component using Angular CLI, by default, `selector`, `templateUrl`, and `styleUrl` properties are set. For ProductComponent, the template is in external HTML file *product.component.html*.

```
<p>
  product works!
</p>
```

You can pass data and capture events between the component class and its template using data binding. We will cover this in detail in further sections.

### USING A COMPONENT

A component can be used inside an Angular application in various ways:

1. As a root component.
2. As a child component. We can use a component inside another Component.
3. Navigate to a component using routing. In this case, the component will be loaded in `RouterOutlet`.
4. Dynamically loading component using `ComponentFactoryResolver`.

The component must be part of a module. To use a component in a module, first import it and then pass it to declaration array of the module.

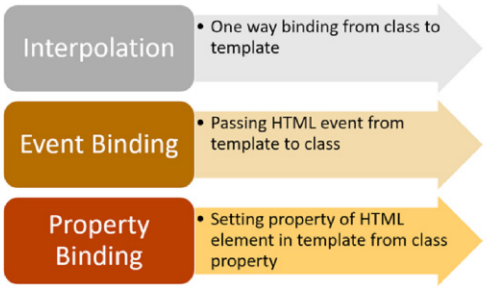
```
@NgModule({
    declarations: [
        AppComponent,
        ProductComponent
    ],
```

### DATA BINDING

In Angular, data binding determines how data will flow in between the component class and component template.

Angular provides us three types of data bindings:

1. Interpolation
2. Property binding
3. Event binding




### INTERPOLATION

Angular interpolation is one-way data binding. It is used to pass data from the component class to the template. The syntax of interpolation is `{{propertyname}}`.

Let's say that we have component class as shown below:

```
export class AppComponent {

      product = {
          title: 'Cricket Bat',
          price: 500
      };
  }
```

We need to pass the product from the component class to the template. Keep in mind that to keep the example simple, we're hard-coding the value of the product object; however, in a real scenario, data could be fetched from the database using the API. We can display the value of the product object using interpolation, as shown in the listing below:

```
   <h1>Product</h1>
<h2>Title : {{product.title}}</h2>
<h2>Price : {{product.price}}</h2>
```

Using interpolation, data is passed from the component class to the template. Ideally, whenever the value of the product object is

changed, the template will be updated with the updated value of the product object.

In Angular, there is something called the change detector service that makes sure that the value of the property in the component class and the template are in sync with each other.

Therefore, if you want to display data in Angular, you should use interpolation data binding.

### PROPERTY BINDING

Angular provides you with a second type of binding called property binding. The syntax of property binding is the square bracket: `[]`. It allows for setting the property of HTML elements on a template with the property from the component class.

So, let's say that you have a component class like the one below:

```
export class AppComponent {
        btnHeight = 100;
       btnWidth = 100;
   }
```

Now, you can set the height and width properties of a button on a template with the properties of the component class using property binding.

```
<button
    [style.height.px] = 'btnHeight'
    [style.width.px] = 'btnWidth' >
        Add Product
</button
```

Angular property binding is used to set the property of HTML elements with the properties of the component class. You can also set properties of other HTML elements like image, list, table, etc. Whenever the property's value in the component class changes, the HTML element property will be updated in the property binding.

### EVENT BINDING

Angular provides you with a third type of binding to capture events raised on templates in a component class. For instance, there's a button on the component template that allows you to call a function in the component class. You can do this using event binding. The syntax behind event binding is `(eventname)`.

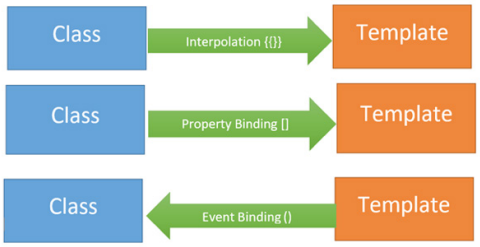
For this example, you might have a component class like this:

```
export class AppComponent {
     addProduct() {
         console.log('add product');
     }
}
```

You want to call the `addProduct` function on the click of a button on the template. You can do this using event binding:

```
<h1>Product</h1>
<button (click)='addProduct()'>
    Add Product
</button>
```

Angular provides you these three bindings. In event binding, data flows from template to class and in property binding and interpolation, data flows from class to template.




Angular does not have built-in two-way data binding; however, by combining property binding and event binding, you can achieve two-way data binding.


Angular provides us a directive, `ngModel`, to achieve two-way data binding, and it's very easy to use. First, import `FormsModule`, and then you can create two-way data binding:

```
export class AppComponent {

     name = 'foo';
 }
```

We can two-way data-bind the name property with an input box:

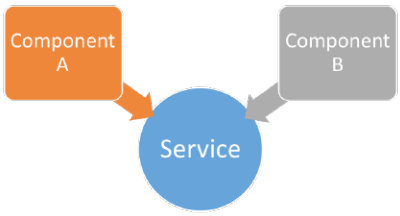
```
<input type="text" [(ngModel)]='name' />

<h2>{{name}}</h2>
```

As you see, we are using `[(ngModel)]` to create two-way data-binding between input control and name property. Whenever a user changes the value of the input box, the name property will be updated and vice versa.

### COMPONENT COMMUNICATION

In Angular, components communicate to each other to share data such as object, string, number, array, or HTML.

To understand component communication, first, we need to understand relationship between components. For example, when two components are not related to each other, they communicate through an Angular service.

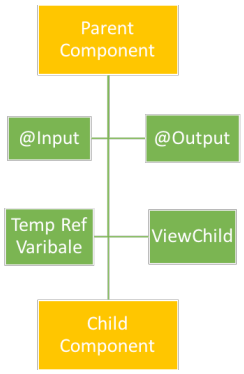



When you use a component inside another component, you create a component hierarchy. The component being used inside another component is known as the child component and the enclosing component is known as the parent component. As shown in the image below, in context of `AppComponent`, app-child is a child component and AppComponent is a parent component.

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

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

  <h1>Hello {{message}}</h1>
  <app-child></app-child> //child component
  `,
})
export class AppComponent { //parent component

  message = 'I am Parent';
}
```

Parent and child components can communicate to each other in following ways:

- `@Input()`
- `@Output()`
- `Temp Ref Variable`
- `ViewChild`
- `ContentChild`




When components are not related to each other, they communicate using services. Otherwise, they communicate using one of the various options depending on the communication criteria. Let's explore all the options one by one.

### @INPUT

You can pass data from a parent component to a child component using the `@Input` decorator. Data could be of any form such as the primitive type's string, number, object, array, etc.


To understand use of `@Input`, let's create a component:

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

@Component({
    selector: 'app-child',
    template: `<h2>Hi {{greetMessage}}</h2>`
})
export class AppChildComponent {

    greetMessage = 'I am Child';

}
```

Use the `AppChild` component inside `AppComponent`:

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

  <h1>Hello {{message}}</h1>
  <app-child></app-child>
  `,
})
export class AppComponent {
  message = 'I am Parent';
}
```

`AppComponent` is using `AppChildComponent`, so `AppComponent` is the parent component and `AppChildComponent` is the child component. To pass data, the `@Input` decorator uses the child component properties. To do this, we'll need to modify child `AppChildComponent` as shown in the listing below:

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

@Component({
    selector: 'app-child',
    template: `<h2>Hi {{greetMessage}}</h2>`
})
export class AppChildComponent implements OnInit {
    @Input() greetMessage: string;
    constructor() {

    }
    ngOnInit() {

    }
}
```

As you notice, we have modified the `greetMessage` property with the `@Input()` decorator. So essentially, in the child component, we have decorated the `greetMessage` property with the `@Input()` decorator so that value of the `greetMessage` property can be set from the parent component. Next, let's modify the parent component `AppComponent` to pass data to the child component.

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

@Component({
  selector: 'app-root',
  template: `
  <h1>Hello {{message}}</h1>
  <appchild [greetMessage]="childmessage"></appchild>
  `,
})
export class AppComponent {

  message = 'I am Parent';
  childmessage = 'I am passed from Parent to child
component';

}
```

From the parent component, we are setting the value of the child component's property `greetMessage`. To pass a value to the child component, we need to pass the child component property inside a square bracket and set its value to any property of the parent component. We are passing the value of the `childmessage` property from the parent component to the `greetMessage` property of the child component.

### @OUTPUT

You can emit the event from the child component to the parent component using the `@Output` decorator.


### TEMP REF VARIABLE

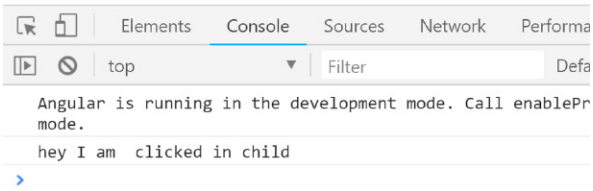
Angular is based on a one-directional data flow and does not have two-way data binding. So, we use `@Output` in a component to emit an event to another component. Let's modify `AppChildComponent` as shown in the listing below:

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

@Component({
    selector: 'app-child',
    template: `<button (click)="handleclick()">Click me<
button> `
})
export class AppChildComponent {

    handleclick() {

        console.log('hey I am  clicked in child');
    }
}
```

There is a button in the `AppChildComponent` template calling the function handleclick. Let's use the app-child component inside the `AppComponent` as shown in the listing below:

```
import { Component, OnInit } from '@angular/core';
@Component({
    selector: 'app-root',
    template: `<app-child></app-child>`
})
export class AppComponent implements OnInit {
    ngOnInit() {
    }
}
```

Here, we're using `AppChildComponent` inside `AppComponent`, thereby creating a parent/child kind of relationship in which `AppComponent` is the parent and `AppChildComponent` is the child. When we run the application, we'll see this message in the browser console:




So far, it's very simple to use event binding to get the button to call the function in the component. Now, let's tweak the requirement a bit. What if you want to execute a function of `AppComponent` on the click event of a button inside `AppChildComponent`?

To do this, you will have to emit the button-click event from `AppChildComponent`. Import `EventEmitter` and output from `@angular/core`.

Here, we are going to emit an event and pass a parameter to the event. Modify `AppChildComponent` as shown in next code listing:

```
import { Component, EventEmitter, Output } from '@angular
core';
@Component({
    selector: 'app-child',
    template: `<button (click)="valueChanged()">Click me
<button> `
```

CODE CONTINUED ON NEXT COLUMN
CODE CONTINUED ON NEXT PAGE

```
})
export class AppChildComponent {
    @Output() valueChange = new EventEmitter();
    counter = 0;
    valueChanged() {

        this.counter = this.counter + 1;
        this.valueChange.emit(this.counter);

    }
}
```

We performed the following tasks in the `AppChildComponent` class:

- Created a variable called counter that will be passed as the parameter of the emitted event.
- Created an `EventEmitter valueChange` that will be emitted to the parent component.
- Created a function named `valueChanged()`. This function is called on the click event of the button, and inside the function, the event `valueChange` is emitted.
- While emitting the `valueChange` event, the value of the counter is passed as a parameter.

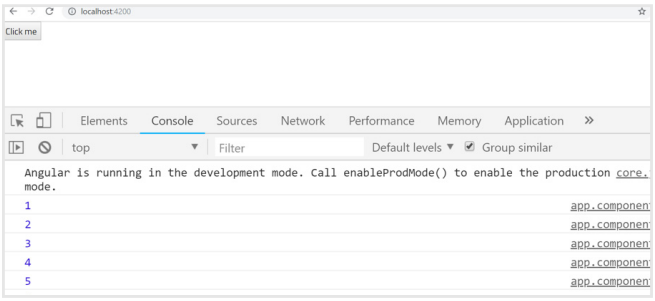
In the parent component `AppComponent`, the child component `AppChildComponent` can be used as shown in the listing below:

```
import { Component, OnInit } from '@angular/core';
@Component({
    selector: 'app-root',
    template: `<app-child
(valueChange)='displayCounter($event)'></app-child>`
})
export class AppComponent implements OnInit {
    ngOnInit() {
    }
    displayCounter(count) {
        console.log(count);
    }
}
```

Right now, we are performing the following tasks in the `AppComponent` class:

- Using `<app-child>` in the template.
- In the `<app-child>` element, using event binding to use the `valueChange` event.
- Calling the `displayCounter` function on the `valueChange` event.
- In the `displayCounter` function, printing the value of the counter passed from the `AppChildComponent`.

As you can see, the function of `AppComponent` is called upon the click event of the button placed on the `AppChildComponent`. This is can be done with `@Output` and `EventEmitter`. When you run the application and click the button, you can see the value of the counter in the browser console. Each time you click on the button, the counter value is increased by 1.




## DIRECTIVES

Directives create DOM elements and change their structure or behavior in an Angular application. There are three types of directives in Angular:

1. **Components**: Directives with templates.
2. **Attribute directives**: Change appearance and behavior of an element, component, or other directive.
3. **Structural directives**: Change DOM layout by adding or removing elements.

The basic difference between a component and a directive is that a component has a template, whereas an attribute or structural directive does not have a template. Angular has provided us many inbuilt structural and attribute directives. Inbuilt structural directives are `*ngFor` and `*ngIf`. Attribute directives are `NgStyle` and `NgModel`.

### USING STRUCTURAL DIRECTIVES

`*ngIf` is a structure directive used to provide an "if" condition to the statements to get executed. If the expression evaluates to a False value, the elements are removed from the DOM, whereas if it evaluates to True, the element is added to the DOM.

Consider the below listing. In this, the `*ngIf` directive will add `div` in DOM if the value of `showMessage` property is True.

```
@Component({
  selector: 'app-message',
  template:`
  <div *ngIf = 'showMessage'>
    Show Message
  </div>
  `
})
```

CODE CONTINUED ON NEXT PAGE

```
export class AppMessageComponent {

        showMessage  = true;
  }
```

Keep in mind that `*ngIf` does not hide or show a DOM element. Rather, it adds or removes depending on the condition.

The `*ngFor` structure directive creates DOM elements in a loop. Consider the below listing. In this, the `*ngFor` directive will add rows in a table depending on number of items in the data array.

```
@Component({

  selector: 'app-message',
  template:`
  <table>
   <tr *ngFor='let f of data'>
   <td>{{f.name}}</td>
   </tr>
  </table>
  `
})
```

CODE CONTINUED ON NEXT COLUMN

```
export class AppMessageComponent {

        data = [
          {name : 'foo'},
          {name: 'koo'}
        ];
  }
```

In most cases, you won't have to create custom structural directives; built-in directives should be enough.



DZone, Inc.
150 Preston Executive Dr. Cary, NC 27513
888.678.0399  919.678.0300