Angular

Intro, Binding, Directives and Pipes

ITI – Assiut Branch
Eng. Hany Saad

What's Angular?

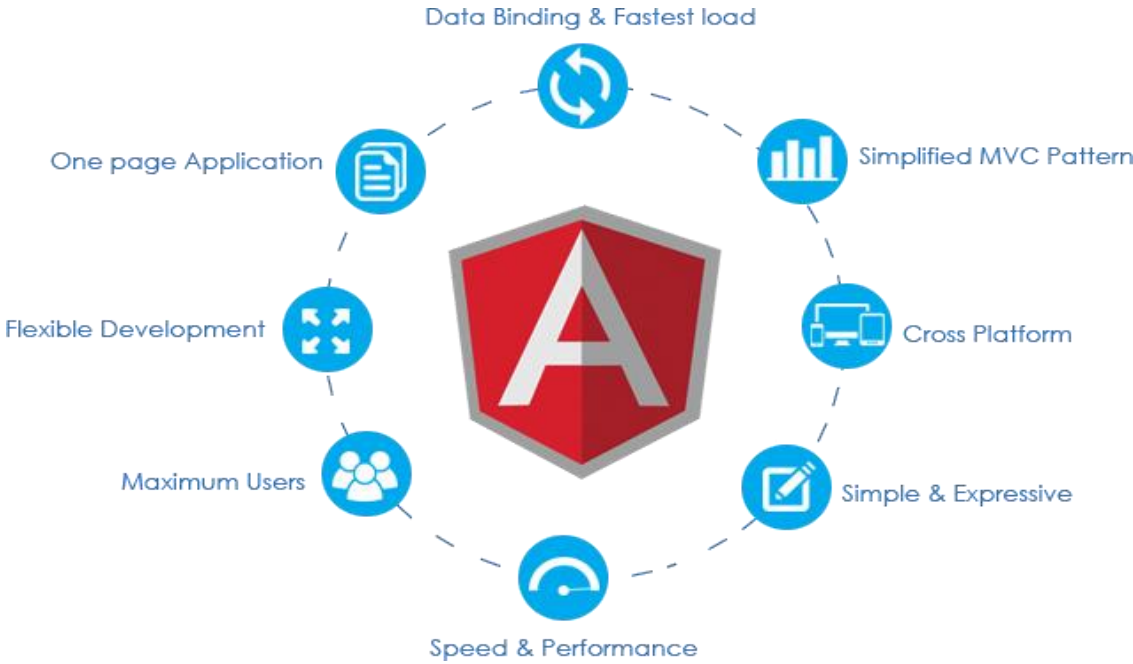
Angular is a development platform, built on TypeScript. As a platform, Angular includes:

•A component-based framework for building scalable web applications

•A collection of well-integrated libraries that cover a wide variety of features, including routing, forms management, client-server communication, and more

•A suite of developer tools to help you develop, build, test, and update your code

With Angular, you're taking advantage of a platform that can scale from single-developer projects to enterprise-level applications. Angular is designed to make updating as straightforward as possible, so take advantage of the latest developments with a minimum of effort.



Other libraries like Angular

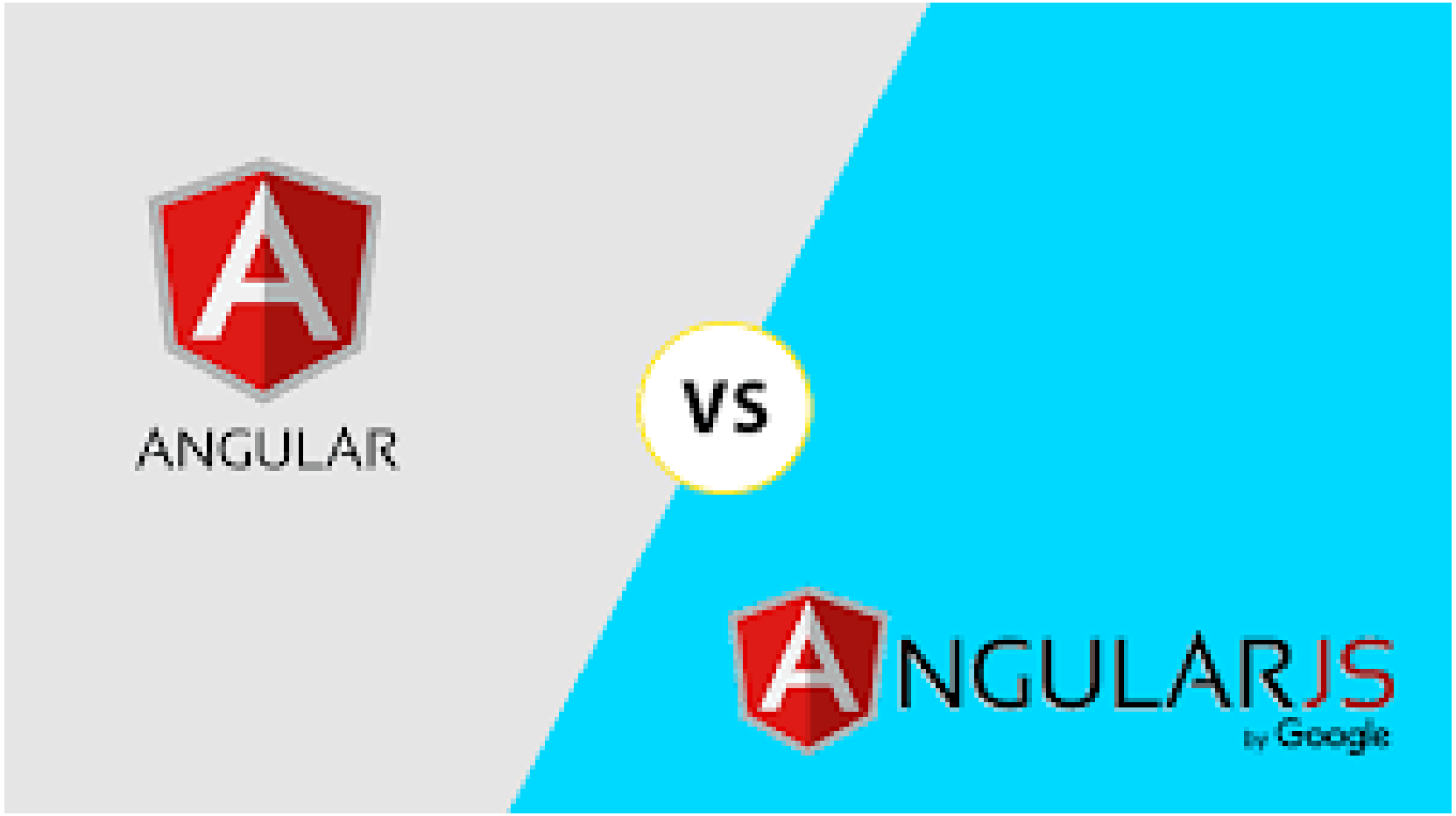
2020 MARKET LEADING
MAJOR FRAMEWORKS

ANGULAR
Released: 2016 (v2)
Current version: 8
Size: ~170KB (Modern ES2015 Browsers), ~250KB
Support: Google

REACT
Released: 2013
Current version: 16.12
Size: ~100KB
Support: Facebook

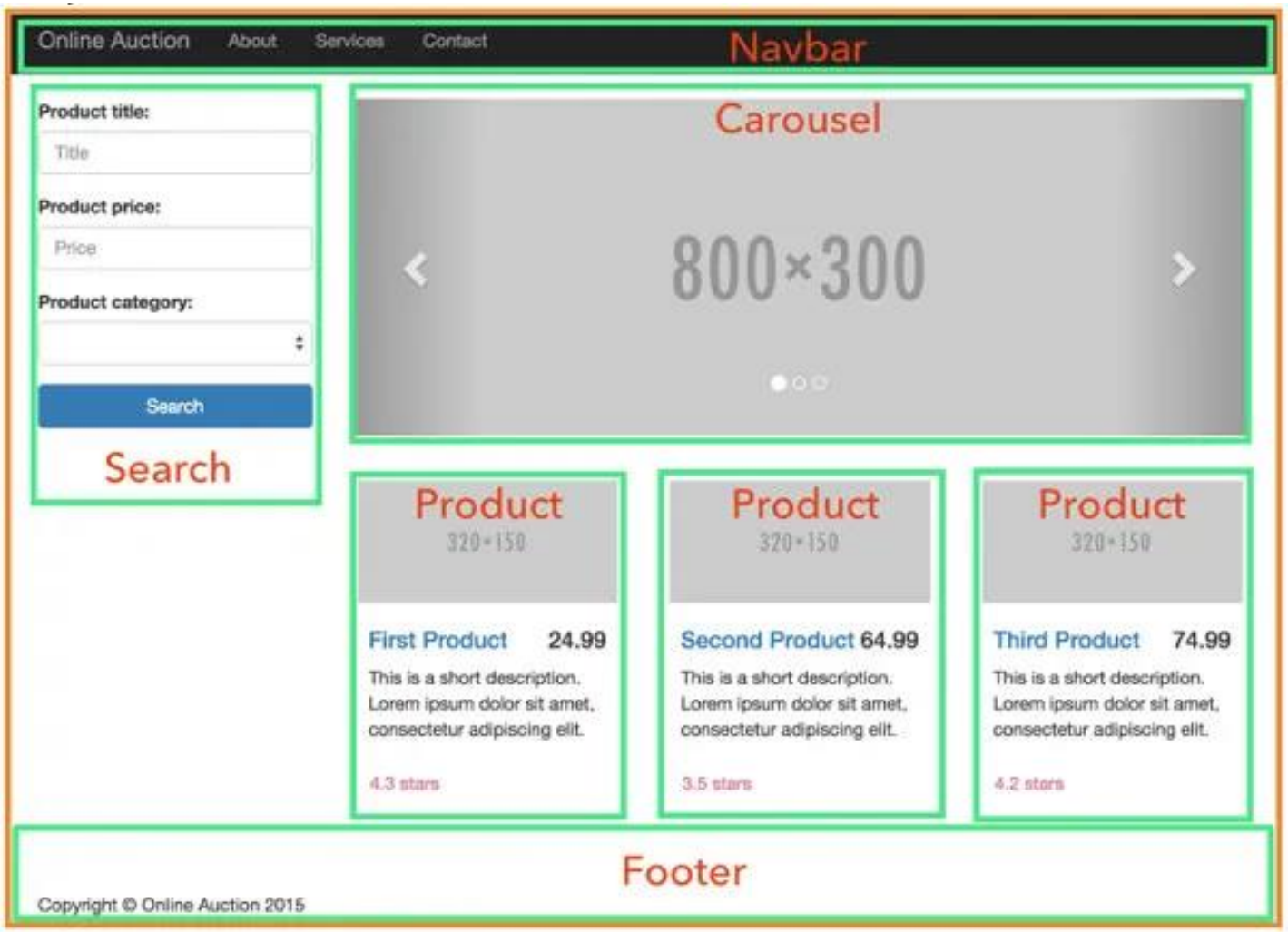
VUE
Released: 2014
Current version: 2.6.11
Size: ~80KB
Support: Open Source Community

AngularJS vs. Angular



Angular components

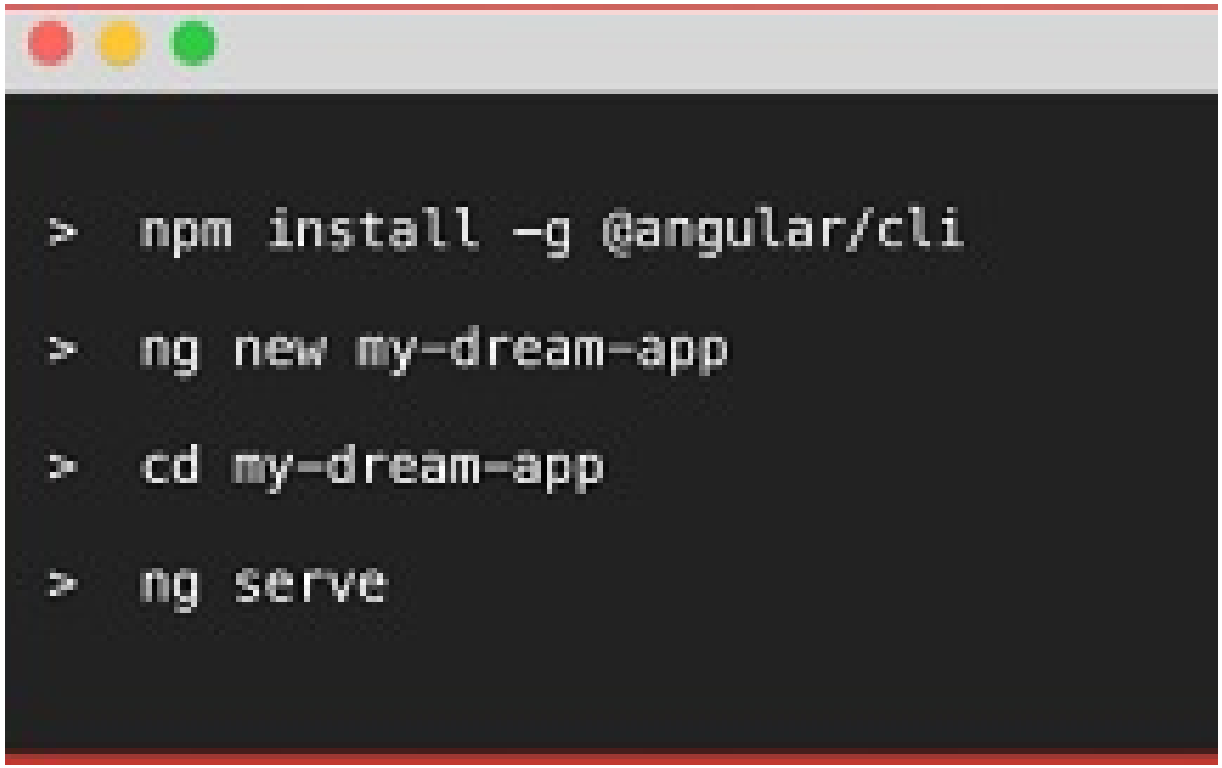
Think Components! Everything is a component



Get Started…

Online Tutorial: https://angular.io/guide/quickstart

Your First project…

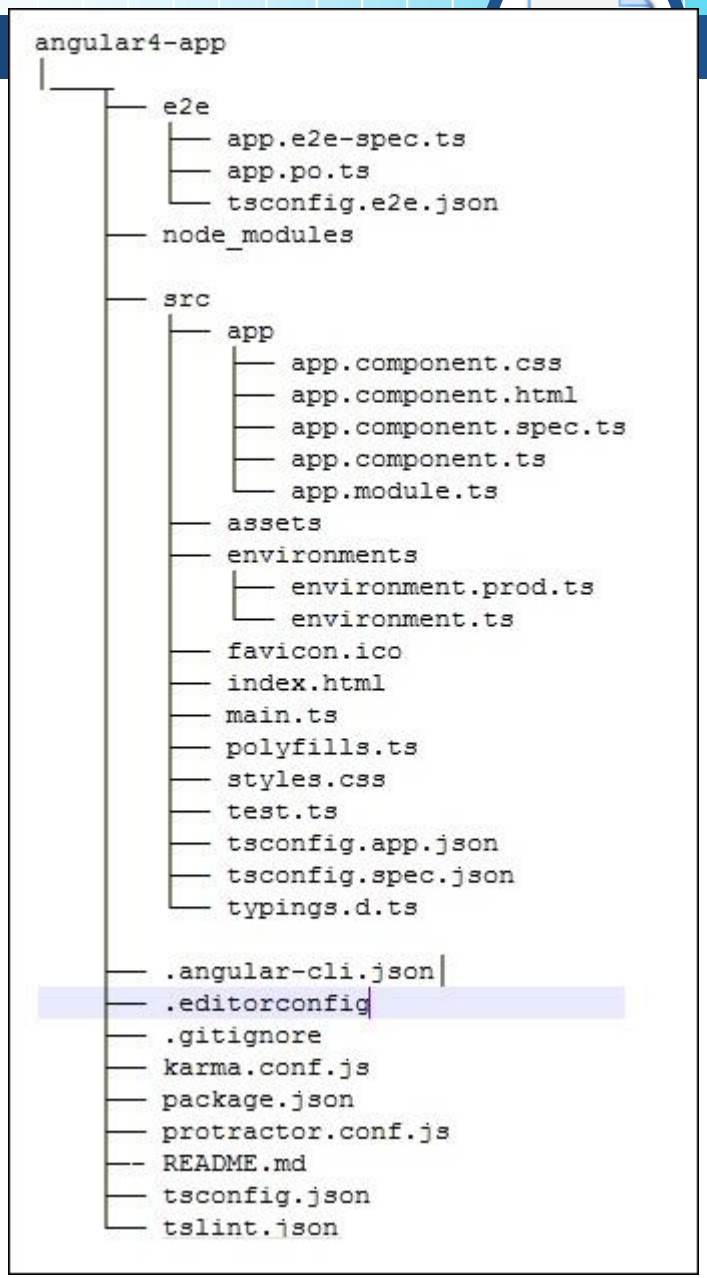


Note: You can change the port using the following command:
>ng serve --host 0.0.0.0 –port 4205

Project structure

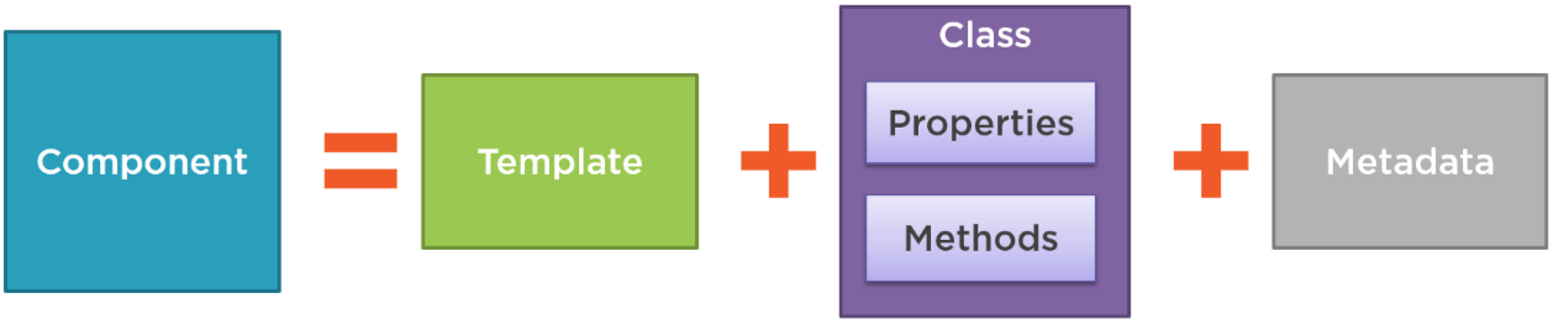
https://angular.io/guide/quickstart#project-file-review

https://www.tutorialspoint.com/angular4/angular4_project_setup.htm



Component

What Is a Component?



- View layout
- Created with HTML
- Includes binding and directives

- Code supporting the view
- Created with TypeScript
- Properties: data
- Methods: logic

- Extra data for Angular
- Defined with a decorator

Component (Cont.)

Major part of the development with Angular 4 is done in the components. Components are basically classes that interact with the .html file of the component, which gets displayed on the browser. We have seen the file structure in one of our previous chapters. The file structure has the app component and it consists of the following files –

- **app.component.css**
- **app.component.html**
- **app.component.spec.ts**
- **app.component.ts**
- **app.module.ts**

Component (Cont.)


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

@Component({
   selector: 'app-root',
   templateUrl: './app.component.html',
   styleUrls: ['./app.component.css']
})
export class AppComponent {
   title = 'app';
}
```

When we hit the url in the **http://localhost:4200/** browser, it first executes the index.html file which is shown below –

```
<!doctype html>
<html lang = "en">
   <head>
      <meta charset = "utf-8">
      <title>Angular 4App</title>
      <base href = "/">
      <meta name="viewport" content="width = device-width, initial-scale = 1">
      <link rel = "icon" type = "image/x-icon" href = "favicon.ico">
   </head>

   <body>
      <app-root></app-root>
   </body>
</html>
```

Component (Cont.)

Decorator

A decorator is a function that is invoked with @ prefix and followed by class, method, property.

Ex:

@Input()

description: string;

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

@Component({
   selector: 'app-root',
   templateUrl: './app.component.html',
   styleUrls: ['./app.component.css']
})
export class AppComponent {
   title = 'app';
}
```

Module

Module in Angular refers to a place where you can group the components, directives, pipes, and services, which are related to the application.

In case you are developing a website, the header, footer, left, center and the right section become part of a module.

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppComponent } from './app.component';

@NgModule({
   declarations: [
      AppComponent
   ],
   imports: [
      BrowserModule
   ],
   providers: [],
   bootstrap: [AppComponent]
})

export class AppModule { }
```

Module (Cont.)

main.ts

main.ts is the file from where we start our project development. It starts with importing the basic module which we need. Right now if you see angular/core, angular/platform-browser-dynamic, app.module and environment is imported by default during angular-cli installation and project setup.

```
import { enableProdMode } from '@angular/core';
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

import { AppModule } from './app/app.module';
import { environment } from './environments/environment';

if (environment.production) {
   enableProdMode();
}
platformBrowserDynamic().bootstrapModule(AppModule);
```

Angular Interpolation

Interpolation {{...}}

Interpolation refers to embedding expressions into marked up text. By default, interpolation uses as its delimiter the double curly braces, `{{` and `}}`.

In the following snippet, `{{ currentCustomer }}` is an example of interpolation.

src/app/app.component.html

```
<h3>Current customer: {{ currentCustomer }}</h3>
```

The text between the braces is often the name of a component property. Angular replaces that name with the string value of the corresponding component property.

src/app/app.component.html

```
<p>{{title}}</p>
<div><img src="{{itemImageUrl}}"></div>
```

Note: You can use ? After object name in expression to check if it's null.

```
<span [style.text-decoration]="lineThrough">
  {{prefix}} {{hero?.name}}
</span>
```

Angular Interpolation (Cont.)

Expression context

The *expression context* is typically the *component* instance. In the following snippets, the `recommended` within double curly braces and the `itemImageUrl2` in quotes refer to properties of the `AppComponent`.

src/app/app.component.html

```
<h4>{{recommended}}</h4>
<img [src]="itemImageUrl2">
```

An expression may also refer to properties of the *template's* context such as a template input variable,

`let customer`, or a template reference variable, `#customerInput`.

src/app/app.component.html (template input variable)

```
<ul>
  <li *ngFor="let customer of customers">{{customer.name}}</li>
</ul>
```

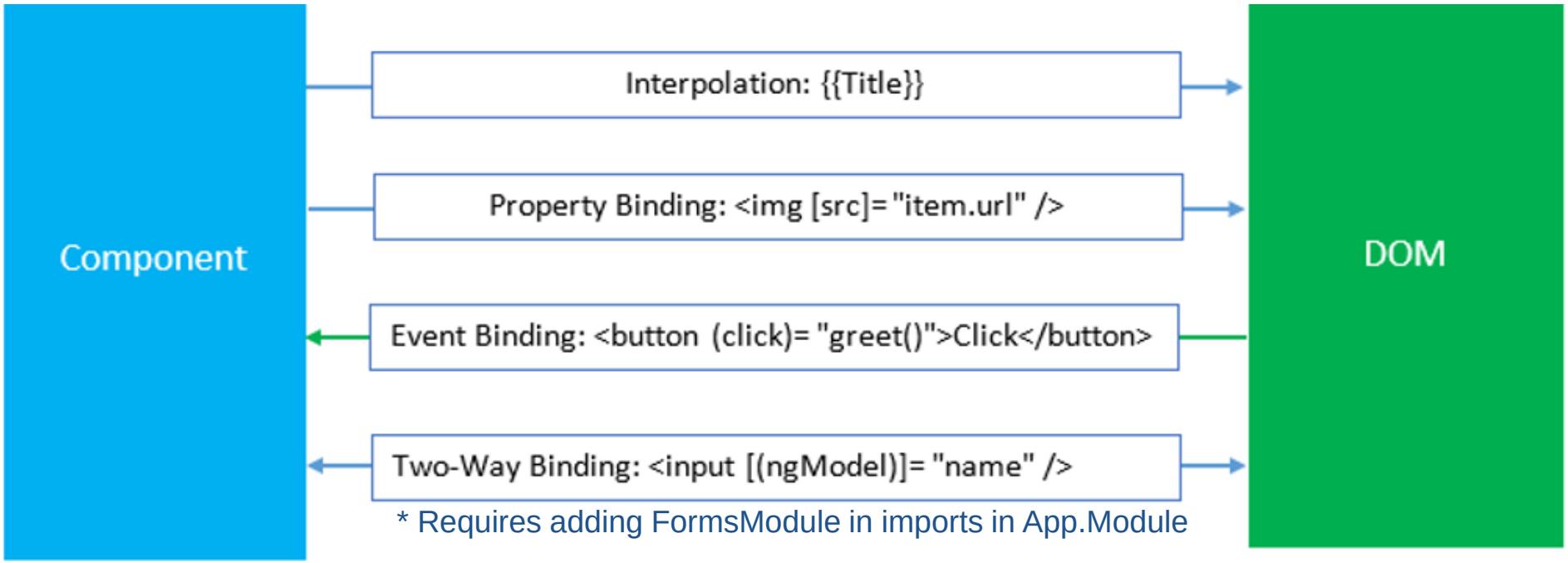
src/app/app.component.html (template reference variable)

```
<input #customerInput>{{customerInput.value}}</label>
```

The context for terms in an expression is a blend of the *template variables*, the directive's *context* object (if it has one), and the component's *members*. If you reference a name that belongs to more than one of these namespaces, the template variable name takes precedence, followed by a name in the directive's *context*, and, lastly, the component's member names.

Template variables: https://angular.io/guide/template-reference-variables

Data Binding



* Requires adding FormsModule in imports in App.Module

Property Binding (Cont.)

Property binding ([property])

Write a template **property binding** to set a property of a view element. The binding sets the property to the value of a template expression.

The most common property binding sets an element property to a component property value. An example is binding the `src` property of an image element to a component's `heroImageUrl` property:

src/app/app.component.html

```
<img [src]="heroImageUrl">
```

Another example is disabling a button when the component says that it `isUnchanged`:

src/app/app.component.html

```
<button [disabled]="isUnchanged">Cancel is disabled</button>
```

Another is setting a property of a directive:

src/app/app.component.html

```
<div [ngClass]="classes">[ngClass] binding to the classes property</div>
```

Property Binding (Cont.)

Class binding

You can add and remove CSS class names from an element's `class` attribute with a **class binding**.

Class binding syntax resembles property binding. Instead of an element property between brackets, start with the prefix `class`, optionally followed by a dot (`.`) and the name of a CSS class: `[class.class-name]`.

The following examples show how to add and remove the application's "special" class with class bindings. Here's how to set the attribute without binding:

src/app/app.component.html

```
<!-- standard class attribute setting  -->
<div class="bad curly special">Bad curly special</div>
```

You can replace that with a binding to a string of the desired class names; this is an all-or-nothing, replacement binding.

src/app/app.component.html

```
<!-- reset/override all class names with a binding  -->
<div class="bad curly special"
     [class]="badCurly">Bad curly</div>
```

Finally, you can bind to a specific class name. Angular adds the class when the template expression evaluates to truthy. It removes the class when the expression is falsy.

src/app/app.component.html

```
<!-- toggle the "special" class on/off with a property -->
<div [class.special]="isSpecial">The class binding is special</div>

<!-- binding to `class.special` trumps the class attribute -->
<div class="special"
     [class.special]="!isSpecial">This one is not so special</div>
```

Property Binding (Cont.)

Style binding

You can set inline styles with a **style binding**.

Style binding syntax resembles property binding. Instead of an element property between brackets, start with the prefix `style`, followed by a dot (`.`) and the name of a CSS style property: `[style.style-property]`.

src/app/app.component.html

```
<button [style.color]="isSpecial ? 'red': 'green'">Red</button>
<button [style.background-color]="canSave ? 'cyan': 'grey'" >Save</button>
```

Some style binding styles have a unit extension. The following example conditionally sets the font size in “em” and “%” units .

src/app/app.component.html

```
<button [style.font-size.em]="isSpecial ? 3 : 1" >Big</button>
<button [style.font-size.%]="!isSpecial ? 150 : 50" >Small</button>
```

Event Binding

Event binding ((event))

The bindings directives you've met so far flow data in one direction: **from a component to an element**.

Users don't just stare at the screen. They enter text into input boxes. They pick items from lists. They click buttons. Such user actions may result in a flow of data in the opposite direction: **from an element to a component**.

The only way to know about a user action is to listen for certain events such as keystrokes, mouse movements, clicks, and touches. You declare your interest in user actions through Angular event binding.

Event binding syntax consists of a **target event** name within parentheses on the left of an equal sign, and a quoted template statement on the right. The following event binding listens for the button's click events, calling the component's `onSave()` method whenever a click occurs:

src/app/app.component.html

```
<button (click)="onSave()">Save</button>
```

Event Binding (Cont.)

$event and event handling statements

In an event binding, Angular sets up an event handler for the target event.

When the event is raised, the handler executes the template statement. The template statement typically involves a receiver, which performs an action in response to the event, such as storing a value from the HTML control into a model.

The binding conveys information about the event, including data values, through an **event object named** `$event`.

The shape of the event object is determined by the target event. If the target event is a native DOM element event, then `$event` is a DOM event object, with properties such as `target` and `target.value`.

Consider this example:

src/app/app.component.html

```
<input [value]="currentHero.name"
       (input)="currentHero.name=$event.target.value" >
```

This code sets the input box `value` property by binding to the `name` property. To listen for changes to the value, the code binds to the input box's `input` event. When the user makes changes, the `input` event is raised, and the binding executes the statement within a context that includes the DOM event object, `$event`.

To update the `name` property, the changed text is retrieved by following the path `$event.target.value`.

Event Binding (Cont).

Get user input from the $event object

DOM events carry a payload of information that may be useful to the component. This section shows how to bind to the keyup event of an input box to get the user's input after each keystroke.

The following code listens to the keyup event and passes the entire event payload (`$event`) to the component event handler.

src/app/keyup.components.ts (template v.1)

```
template: `
  <input (keyup)="onKey($event)">
  <p>{{values}}</p>
`
```

When a user presses and releases a key, the keyup event occurs, and Angular provides a corresponding DOM event object in the `$event` variable which this code passes as a parameter to the component's `onKey()` method.

src/app/keyup.components.ts (class v.1)

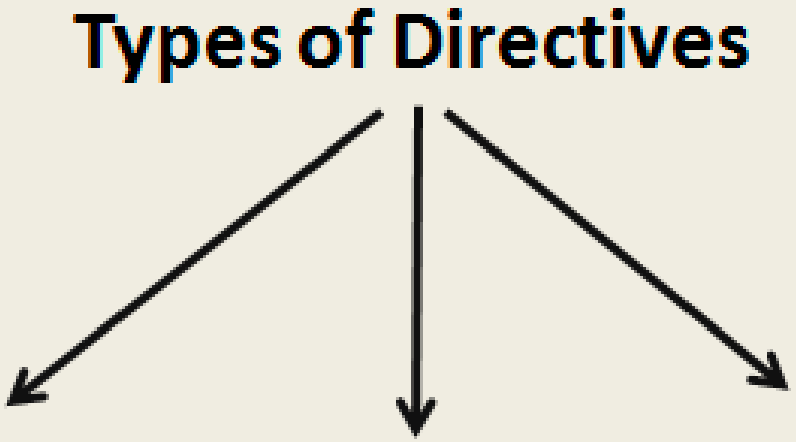
```
export class KeyUpComponent_v1 {
  values = '';

  onKey(event: any) { // without type info
    this.values += event.target.value + ' | ';
  }
}
```

The properties of an `$event` object vary depending on the type of DOM event. For example, a mouse event includes different information than an input box editing event.

More Details: https://angular.io/guide/template-syntax#event-binding---event-
https://angular.io/guide/user-input

Directives



Component

Directives with a template

Structure Directives

Change layout of the elements

- NgIf
- NgFor
- NgSwitch

Attribute Directives

Change appearance or behavior of a particular element

- NgClass
- NgStyle
- NgModel

Directives (Cont.)

Built-in directives

Directives are classes that add additional behavior to elements in your Angular applications. Use Angular's built-in directives to manage forms, lists, styles, and what users see.

The different types of Angular directives are as follows:

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

Directives (Cont.)

Built-in structural directives

Structural directives are responsible for HTML layout. They shape or reshape the DOM's structure, typically by adding, removing, and manipulating the host elements to which they are attached.

This section introduces the most common built-in structural directives:

- `NgIf` —conditionally creates or disposes of subviews from the template.
- `NgFor` —repeat a node for each item in a list.
- `NgSwitch` —a set of directives that switch among alternative views.

Directives - ngFor


```
1  <tr *ngFor=" hero in heroes">
2      <td>{{hero.name}}</td>
3  </tr>
```

Finding the index of a list element

A very common requirement is to add to a list the numeric index position of its element. We can get the index of the current element by using the `index` variable:

```
1  <tr *ngFor="let hero of heroes; let i = index">
2      <td>{{hero.name}}</td>
3      <td>{{i}}</td>
4  </tr>
```

More Details: https://blog.angular-university.io/angular-2-ngfor/
https://codecraft.tv/courses/angular/built-in-directives/ngfor/

Directives - ngIf


```
<div *ngIf="show">Text to show</div>
```

```
<div *ngIf="show; else elseBlock">Text to show</div>
<ng-template #elseBlock>Alternate text while primary text is hidden</ng-template>
```

```
<div *ngIf="show; then thenBlock; else elseBlock">this is ignored</div>
<ng-template #primaryBlock>Primary text to show</ng-template>
<ng-template #secondaryBlock>Secondary text to show</ng-template>
<ng-template #elseBlock>Alternate text while primary text is hidden</ng-template>
```

More Details: https://angular.io/api/common/NgIf
https://codecraft.tv/courses/angular/built-in-directives/ngif-and-ngswitch/

Directives - ngSwitch


```
<div class='card'>
  <div class='card-header'>
    ngSwitch Example
  </div>
  <div class="card-body">
    Input string : <input type='text' [(ngModel)]="num" />

    <div [ngSwitch]="num">
      <div *ngSwitchCase="'1'">One</div>
      <div *ngSwitchCase="'2'">Two</div>
      <div *ngSwitchCase="'3'">Three</div>
      <div *ngSwitchCase="'4'">Four</div>
      <div *ngSwitchCase="'5'">Five</div>
      <div *ngSwitchDefault>This is Default</div>
    </div>
  </div>
</div>
```

More Details: https://codecraft.tv/courses/angular/built-in-directives/ngif-and-ngswitch/

Built-in attribute directives

Built-in *attribute* directives

Attribute directives listen to and modify the behavior of other HTML elements, attributes, properties, and components. They are usually applied to elements as if they were HTML attributes, hence the name.

Many details are covered in the *Attribute Directives* guide. Many NgModules such as the `RouterModule` and the `FormsModule` define their own attribute directives. This section is an introduction to the most commonly used attribute directives:

- `NgClass` - add and remove a set of CSS classes
- `NgStyle` - add and remove a set of HTML styles
- `NgModel` - two-way data binding to an HTML form element

Built-in attribute directives - ngStyle

NgStyle

The `NgStyle` directive lets you set a given DOM elements style properties.

One way to set styles is by using the `NgStyle` directive and assigning it an *object literal*, like so:

```
<div [ngStyle]="{'background-color':'green'}"></div>
```

This sets the background color of the `div` to green.

`ngStyle` becomes much more useful when the value is *dynamic*. The *values* in the object literal that we assign to `ngStyle` can be javascript expressions which are evaluated and the result of that expression is used as the value of the css property, like this:

HTML

```
<div [ngStyle]="{'background-color':person.country === 'UK' ? 'green' : 'red' }"></div>
```

More Details: https://codecraft.tv/courses/angular/built-in-directives/ngstyle-and-ngclass/

Built-in attribute directives - ngClass

NgClass

The `NgClass` directive allows you to set the CSS class dynamically for a DOM element.

> **Tip**
>
> The `NgClass` directive will feel very similar to what `ngClass` used to do in Angular 1.

There are two ways to use this directive, the first is by passing an object literal to the directive, like so:

```
[ngClass]="{'text-success':true}"
```

When using an object literal, the keys are the classes which are added to the element if the value of the key evaluates to true.

So in the above example, since the value is `true` this will set the class `text-success` onto the element the directive is attached to.

The value can also be an *expression*, so we can re-write the above to be.

```
[ngClass]="{'text-success':person.country === 'UK'}"
```

More Details: https://codecraft.tv/courses/angular/built-in-directives/ngstyle-and-ngclass/
https://angular.io/guide/template-syntax
https://angular.io/guide/template-syntax#built-in-attribute-directives

Directives – Custom Directives

You can create custom **attribute directive** and custom **Structural directive**

// https://www.agiliq.com/blog/2020/05/custom-attribute-directives-in-angular/
// https://angular.io/guide/attribute-directives
// https://angular.io/guide/structural-directives

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';
    }
}
```

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>
```

Directives – Custom attribute directive


```
@Directive({
  selector: '[appLightBox]'
})
export class LightBoxDirective implements OnChanges, OnInit {
  @Input() hoverColor: string = 'yellow';
  @Input('appLightBox') defaultColor: string = "white";
  constructor(private elem: ElementRef) {
    // this.elem.nativeElement.style="border: white 3px solid";
  }
  ngOnInit(): void {
  }
  ngOnChanges(): void {
    this.elem.nativeElement.style = "border: " + this.defaultColor + " 5px solid"
  }
  @HostListener('mouseover') onMouseOver() {
    this.elem.nativeElement.style = "border: " + this.hoverColor + " 5px solid"
    console.log(this.hoverColor);
  }
  @HostListener('mouseout') onMouseOut() {
    // this.elem.nativeElement.style="border: white 3px solid";
    this.elem.nativeElement.style = "border: " + this.defaultColor + " 5px solid"
  }

}
```

In HTML:

```
<img [src]="prd.ImgURL" alt="prd" appLightBox="blue" hoverColor="red">
 <!--<img [src]="prd.ImgURL" alt="prd" appLightBox hoverColor="red" defaultColor="blue"> -->
```

Pipes

It takes integers, strings, arrays, and date as input separated with | to be converted in the format as required and display the same in the browser.

```
<div style = "width:100%;">
   <div style = "width:40%;float:left;border:solid 1px black;">
      <h1>Uppercase Pipe</h1>
      <b>{{title | uppercase}}</b><br/>

      <h1>Lowercase Pipe</h1>
      <b>{{title | lowercase}}</b>

      <h1>Currency Pipe</h1>
      <b>{{6589.23 | currency:"USD"}}</b><br/>
      <b>{{6589.23 | currency:"USD":true}}</b> //Boolean true is used to get the sig

      <h1>Date pipe</h1>
      <b>{{todaydate | date:'d/M/y'}}</b><br/>
      <b>{{todaydate | date:'shortTime'}}</b>

      <h1>Decimal Pipe</h1>
      <b>{{ 454.78787814 | number: '3.4-4' }}</b> // 3 is for main integer, 4 -4 are
   </div>

   <div style = "width:40%;float:left;border:solid 1px black;">
      <h1>Json Pipe</h1>
      <b>{{ jsonval | json }}</b>
      <h1>Percent Pipe</h1>
      <b>{{00.54565 | percent}}</b>
      <h1>Slice Pipe</h1>
      <b>{{months | slice:2:6}}</b>
      // here 2 and 6 refers to the start and the end index
   </div>
</div>
```

{minIntegerDigits}.{minFractionDigits}-{maxFractionDigits}.

Custom Pipes

Custom pipes

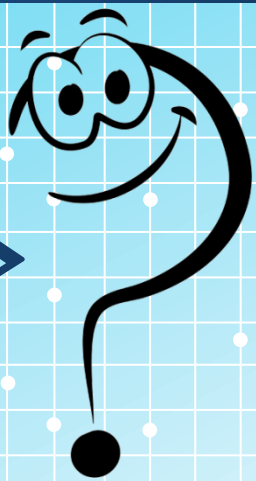
You can write your own custom pipes. Here's a custom pipe named `ExponentialStrengthPipe` that can boost a hero's powers:

src/app/exponential-strength.pipe.ts

```
import { Pipe, PipeTransform } from '@angular/core';
/*
 * Raise the value exponentially
 * Takes an exponent argument that defaults to 1.
 * Usage:
 *   value | exponentialStrength:exponent
 * Example:
 *   {{ 2 | exponentialStrength:10 }}
 *   formats to: 1024
*/
@Pipe({name: 'exponentialStrength'})
export class ExponentialStrengthPipe implements PipeTransform {
  transform(value: number, exponent: string): number {
    let exp = parseFloat(exponent);
    return Math.pow(value, isNaN(exp) ? 1 : exp);
  }
}
```

More: https://angular.io/guide/pipes
https://appdividend.com/2018/12/21/angular-7-pipes-tutorial-example/

<SCRIPT >



</SCRIPT>

```
<script>document.writeln("Thank
You!")</script>
```