What is Angular (Angular 2)?

- Next version of most successful AngularJS 1.x
- Finally released on **14th Sep, 2016**. It is called Angular 2.0.0.
- It has been optimized for developer productivity, small payload size, and performance.
- Developed using TypeScript, which is Microsoft's extension of JavaScript that allows use of all ES 2015 (ECMAScript 6) features and adds type checking and object-oriented features like interfaces.
- You can write code in either JavaScript or TypeScript or Dart.
- Designed for Web, Mobile and Desktop Apps.
- Not an upgrade of Angular 1. It was completely rewritten from scratch.

Differences between AngularJS and Angular2

- 1.x called as AngularJS and 2.x, 4.x, 5.x, 6.x called as Angular.
- Components are used instead of Controllers and $scope.
- A component is a class with its own data and methods.
- Option to write code in different languages.
- Designed for Speed. Supposed to be 5 times faster than Angular 1.
- Designed for Mobile development also.
- More modular. It is broken into many packages.
- Data binding is done with no new directives. We bind to attributes of html elements.
- Event handling is done with DOM events and not directives.
- Simpler API

Angular Application can be test without setting up anything using Plunker

https://embed.plnkr.co/?show=preview&show=app%2Fapp.component.ts

or

https://stackblitz.com/angular/vvkdnxbjyrd

Setting up Local development Environment

We can do this in 3 ways

1.Quickstart

2.Git clone

3.Angular CLI

It is important to setup development environment for Angular in your system. Here are the steps to take up:

Install Node.js and NPM

Install Node.js and NPM. Node.js is used run JavaScript on server and provide environment for build tools.

NPM is used to manager packages related to JavaScript libraries. NPM itself is a Node application.

To install, go to https://nodejs.org/en/download and installer (.msi) for 32-bit or 64-bit. Do the same for other platforms like Mac etc.

Run .MSI file to install Node.js into your system. It is typically installed into c:\Program Files\nodejs folder.

It installs Node.js and NPM. It also sets system path to include them.

Make sure Node version is 4.x.x or higher and NPM version is 3.x.x. or higher.

Go to command prompt and check their versions.

```
node –v

npm –v
ng -v
```

Quickstart Seed

Quickstart seed, maintained on github, is quick way to get started with Angular local development.

Follow the steps below to clone and launch quickstart application.

1. Create a folder for project. Let us call it demo.

2. Download quickstart seed from https://github.com/angular/quickstart/archive/master.zip
3. Extract quickstart-master.zip into folder created above.
4. Install all packages mentioned in packages.json file using:

```
npm install
```

After npm downloaded required packages, start application using the following command. It starts server and monitors for changes in the application.

```
npm start
```

Delete unnecessary files from quickstart application using:

```
for /f %i in (non-essential-files.txt) do del %i /F /S /Q rd .git /s /q
rd e2e /s /q
```

Angular CLI:

ng new projname –Creating new project

ng new projname --routing

ng g c componentname

ng g d directivename

ng g p pipename

ng g s service name

ng g cl classname

ng g m module name

What does quickstart seed contain?

Quickstart seed is to provide a solid foundation for local development. It contains a lot more than what you need in the beginning.

The following are important component in /src folder.

src/app/app.component.ts
Defines AppComponent. It is the root component of what will become a tree of nested components.

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

@Component({
    selector: 'my-app',
    template: `<h1>Hello {{name}}</h1>` })
export class AppComponent { name = 'Angular'; }
```

src/app/app.module.ts
Defines AppModule, the root module that tells Angular how to assemble the application. Right now it declares only the AppComponent.

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

```
@NgModule({
    imports: [ BrowserModule ], declarations:
    [    AppComponent    ],    bootstrap:    [
    AppComponent ]
})
export class AppModule { }
```

src/main.ts
Compiles the application with the JIT compiler and bootstraps the application's main module (AppModule) to run in the browser.

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

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

platformBrowserDynamic().bootstrapModule(AppModule);

To upgrade from angular lower version to higher version use below url
https://update.angular.io/

Jit -Just in time
4.5
All files are compiled in browser
each file compile separately
suitable for local development

Aot -Ahead of time
4.6
All files in commandline
all code compiled like bundles
suitable for production builds

Building Blocks

The following are important components of an Angular application.

1. Modules
2. Components
3. Templates
4. Metadata
5. Data binding
6. Directives
7. Pipes
8. Services
9. Dependency injection
10. Ajax Call
11. Forms
12. Routing

Module

- ❖ A module is a class that is decorated with @NgModule decorator
- ❖ Every application contains at least one module called root module, conventionally called as AppModule.
- ❖ NgModule decorator provides information about module using properties listed below:

- **Declaration** – classes that belong to this module. They may be components, directives and pipes.
- **Exports** – The subset of declarations that should be visible to other modules.
- **Imports** – Specifies modules whose exported classes are needed in this module.
- **Bootstrap** – Specifies the main application view – root component. It is the base for the rest of the application.

- Modules are executed within their own scope, not in the global scope; this means that variables, functions, classes, etc. declared in a module are not visible outside the module unless they are explicitly exported using one of the exportforms. Conversely, to consume a variable, function, class, interface, etc. exported from a different module, it has to be imported using one of the import forms.

- Modules are declarative; the relationships between modules are specified in terms of imports and exports at the file level.

The following code shows how to create a simple module:

AppModule.ts

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

@NgModule({
   imports: [ BrowserModule],
   declarations: [ FirstComponent],
    bootstrap: [ FirstComponent ]
})
export class AppModule { }
```

Note: A module in Angular is a class decorated with @NgModule decorator, whereas a module in JavaScript is a file and all object defined in the file belongs to that module. JavaScript module exports some object using **export** keyword and they can be imported in other modules using **import** statement.

Angular ships a couple of JavaScript modules each beginning with @angular prefix. In order to use objects in those modules, we need to import them using import statement in JavaScript.

```
import { Component } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
```

Component

- A component controls a part of the screen called view.
- Every component is a class with its own data and code.
- A component may depend on services that are injected using dependency injection.
- The template, metadata, and component together describe a view.
- Components are decorated with @Component decorator through which we specify **template** and **selector** (tag) related to component.
- Properties like **templateUrl** and **providers** can also be used.
- Component can be accessed as like a html tag or attribute or class.

FirstComponent.ts

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

@Component({
    selector: 'my-first', // tag to be used in view,[my-first],.my-first
    template : './first.component.html'
})
export class FirstComponent {
      title : string = "Aspire Technologies";
}
```

Templates

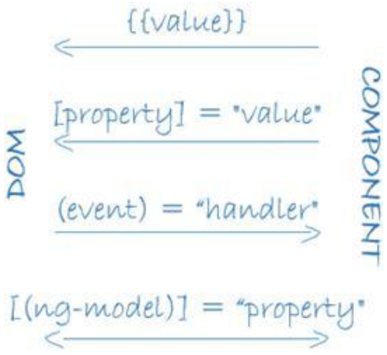
- You define a component's view with its companion template.
- A template is a form of HTML that tells Angular how to render the component.
- A template is a HTML page that interacts with user.
- It uses component properties, interpolation and directives to interact with user.
- Angular manipulates DOM using data binding.

Metadata

- Metadata provided using decorators inform Angular how to process a class.
- For example, @Component decorator tells Angular to treat a class as a component and also provides additional information through attributes of decorator (like selector, template etc.)

Data Binding

- Data from objects should be bound to HTML elements and vice-versa, known as data binding.
- Angular takes care of data binding.
- Enclosing property (attribute of HTML element) copies value to property.
- Enclosing event in parentheses () will assign event handler to event.
- Interpolation allows value of an expression to be used in HTML
- The ng-model is used to for two way data binding.



Binding types can be grouped into three categories distinguished by the direction of data flow: from the *source-to-view*, from *view-to-source*, and in the two-way sequence: *view-to-source-to-view*:

The following are available syntaxes related to data binding:

```
{{expression}}
[target]="expression" bind-
target="expression"
```

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

<img bind-src="heroImageUrl">
```

Note: The brackets tell Angular to evaluate the template expression. If you omit the brackets, Angular treats the string as a constant and *initializes the target property* with that string. It does *not* evaluate the string!

Interpolation

- When you put properties in view template using {{ }}, it is called interpolation.
- Angular updates the display whenever property changes.
- The context of the expression is component instance that is associated with the view.

```
{{ template expression }}
```

- The expression is evaluated , converted to string and then sent to client
- From the expression, we can call methods of component associated with view.

- The target of the template expression may be HTML element, a component, or a directive.
- In case of assigning value to a property, template expression may appear in double quotes.

```
<span [hidden]=”isNew”>Modifed </span>
{{ title }}
<img     src=”{{filename}}” />
```

You often have a choice between interpolation and property binding.

```
<img src="{{imageUrl}}"> is the <i>interpolated</i> image</p>

<img [src]="imageUrl"> is the <i>property bound</i> image</p>
```

When setting an element property to a non-string data value, you must use *property binding*.

Attribute binding

- You can set the value of an attribute directly with an attribute binding.
- You must use attribute binding when there is no element property to bind.
- Attribute binding syntax resembles property binding. Instead of an element property between brackets, start with the prefix **attr**, followed by a dot (.) and the name of the attribute. You then set the attribute value, using an expression that resolves to a string.

```
<td [attr.colspan]="1 + 1">One-Two</td>
```

Class binding

- You can add and remove CSS class names from and 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]

```
<div [class.special]="isSpecial">Class binding is special</div>
```

Angular adds the class when the template expression evaluates to truthy. It removes the class when the expression is falsy.

NgClass directive is usually preferred when managing multiple class names at the same time.

Style binding

- You can set inline styles with a style 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].

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

NgStyle directive **is** generally preferred when setting several inline styles at the same time.

Event Handling

- 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 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. For example, $event.target.value return value of the target element.
- If the name fails to match an element event, Angular reports an error.

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

Two-way binding

It displays a data property and update that property when the user makes changes.
Angular offers a special *two-way data binding* syntax for this purpose, **[(x)]**. It is also called as banana in a box.

Angular **NgModel** directive is a bridge that enables two-way binding to form elements.

Directives

- A directive transforms DOM according to instructions given.
- Components are also directives.
- Directives are two types - structural directives and attribute directives.
- Structural directives alter layout by adding, removing, and replacing elements in DOM.
- Attribute directives alter the appearance or behavior of an existing element. In templates they look like regular HTML attributes, hence the name.
- *ngFor and *ngIf are structural directives.
- ngModel and ngClass are attribute directives.

Built-in Directives

AngularJS (Angular 1.x) shipped with around 74 directives. But Angular cut that short and provides only a few important directives. It was made possible as many things that were done earlier with directives are now done using new data-binding syntax.

Attribute Directives

These attributes are used like attributes of HTML elements. They modify the behavior of HTML elements.

- 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

ngClass

- To add or remove many CSS classes at the same time, the NgClass directive may be the better choice.

- Bind ngClass to a key:value control object where each key of the object is a CSS class name; its value is true if the class should be added, false if it should be removed.

```
currentClasses = {"special": true, "big"  : true};
```

```
<div [ngClass]="currentClasses"> … </div>
```

ngStyle

- To set many inline styles at the same time, the NgStyle directive may be the better choice.
- Bind ngStyle to a key:value control object where each key of the object is a style name and its value is whatever is appropriate for that style.

ngModel

When developing data entry forms, you often both display a data property and update that property when the user makes changes.

```
<input [(ngModel)]="firstName">
```

Note: Before using the ngModel directive in a two-way data binding, you must import the FormsModule and add it to the Angular module's imports list.

app.attributedirectives.ts

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

@Component({
   selector: 'attribute-directives',
   templateUrl : 'app/app.attributedirectives.html', })
export class AttributeDirectives
{
     currentClasses = {            "special"        : true, "big"            : true };
}
```

app.attributedirectives.html

```
<html>
<head>
        <style>
              .special { color:red;}
              .big { font-size : 30pt}
         </style>
</head>
<body>
        <h1>Attribute Directives Demo</h1>
        <div [ngClass]="currentClasses">Aspire Technologies</div>
        <p><input type="checkbox"
        [(ngModel)]="currentClasses.special">Special</p>
         <p><input type="checkbox" (ngModel)]="currentClasses.big">Big </p>
        <p>{{ currentClasses.special}} </p>
        <p>{{ currentClasses.big}} </p>
 </body>
</html>
```

app.module.ts

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule } from '@angular/forms';
import {AttributeDirectives} from './app.attributedirectives';

@NgModule({
    imports: [ BrowserModule, FormsModule ],
    declarations: [ AttributeDirectives],
    bootstrap: [ AttributeDirectives ]
})
export class AppModule { }
```

Structural Directives

- These directives are responsible for HTML layout.
- They typically add, remove, and manipulate the host elements to which they are attached.

NgIf

- Conditionally add or remove an element from the DOM
- Adding and removing DOM elements is different from displaying and hiding element.

```
<div *ngIf="details">Included only when details is true</div>
```

NgFor

- Repeat a template for each item in a list
- NgFor is a repeater directive — a way to present a list of items. You define a block of HTML that defines how a single item should be displayed.

```
<div *ngFor="let topic of topics”>{{topic}}</div>

<div *ngFor="let topic of topics; let i=index">{{i + 1}} - {{topic}}</div>
```

NgSwitch

- It is like the JavaScript switch statement. It can display one element from among several possible elements, based on a switch condition. Angular puts only the selected element into the DOM.
- NgSwitch is actually a set of three, cooperating directives: NgSwitch, NgSwitchCase, and NgSwitchDefault.

```
<div [ngSwitch]="mode">
    <h1     *ngSwitchCase="'f'">Flight            {{flight}} </h1>
    <h2     *ngSwitchCase="'t'">Train             {{train}}  </h2>
    <h3     *ngSwitchDefault>Private Transport </h3>
</div>
```

Pipes:

- A pipe takes data as input and transforms it to required output.

- Angular provides built-in pipes to many common requirements.
- Pipes are used in template.
- Pipes can also take optional parameters to fine tune output.
- To add parameters to a pipe, follow the pipe name with a colon (:) and then the parameter value (such as currency:'EUR'). If the pipe accepts multiple parameters, separate the values with colons (such as slice:1:5)
- You can chain pipes together so that output of one pipe goes to another pipe as input.

```
{{  birthday  |  date:  "MM/dd/yy"  }}  {{
birthday | date | uppercase}}
```

Built-in pipes

The following are built-in pipes.

- DatePipe
- UpperCasePipe, LowerCasePipe, TitleCasePipe
- CurrencyPipe, PercentPipe , DecimalPipe
- SlicePipe
- JsonPipe

DatePipe

Formats a date according to locale rules.

```
date_expression | date[:format]
```

```
{{  dateObj  |  date  }}  //  output  is  'Apr  11,  2017'  {{  dateObj  |
date:'medium' }}
{{ dateObj | date:'shortTime'}} // output is '9:43 PM'
{{ dateObj | date:'mmss' }}                    // output is '43:11'
```

Format can be any of the following:

- 'medium': equivalent to 'yMMMdjms' (e.g. Sep 3, 2010, 12:05:08 PM for en-US)
- 'short': equivalent to 'yMdjm' (e.g. 9/3/2010, 12:05 PM for en-US)
- 'fullDate': equivalent to 'yMMMMEEEEd' (e.g. Friday, September 3, 2010 for en-US)

- 'longDate': equivalent to 'yMMMMd' (e.g. September 3, 2010 for en-US)
- 'mediumDate': equivalent to 'yMMMd' (e.g. Sep 3, 2010 for en-US)
- 'shortDate': equivalent to 'yMd' (e.g. 9/3/2010 for en-US)
- 'mediumTime': equivalent to 'jms' (e.g. 12:05:08 PM for en-US)
- 'shortTime': equivalent to 'jm' (e.g. 12:05 PM for en-US)

COMPONENT	SYMBOL
year	y
month	M
day	d
weekday	E
hour	j
hour12	h
hour24	H
minute	m
second	s
timezone	z
timezone	Z
timezone	a

DecimalPipe

Format a number.

```
number_expression | number[:digitInfo]
```

digitInfo is a string which has a following format:

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

- minIntegerDigits is the minimum number of integer digits to use. Defaults to 1.
- minFractionDigits is the minimum number of digits after fraction. Defaults to 0.
- maxFractionDigits is the maximum number of digits after fraction. Defaults to 3.

CurrencyPipe

Formats a number as currency using locale rules.

number_expression | currency[:currencyCode[:symbolDisplay[:digitInfo]]]

- **currencyCode** is the ISO 4217 currency code, such as USD for the US dollar and EUR for the euro.
- **symbolDisplay** is a boolean indicating whether to use the currency symbol or code - true: use symbol (e.g. $), false(default): use code (e.g. USD).

PercentPipe

Formats a number as a percentage.

```
number_expression | percent[:digitInfo]
```

```
{{   .20 | percent }}                              //   results 20%
{{   0.125539 | percent:'2.2-3' }}                 // results in 12.554 %
```

SlicePipe

Creates a new List or String containing a subset (slice) of the elements.

```
array_or_string_expression | slice:start[:end]
```

start : The starting index of the subset to return.

end   : The ending index of the subset to return.

```
{{str   |   slice:0:4}}
{{str    |    slice:-4}}
{{str | slice:4:0}}
```

Creating Custom Pipes

It is possible to create custom pipes by following steps given below:

- Create a class and decorate it with @**Pipe** decorator.
- The pipe class implements **PipeTransform**
- interface's transform method that accepts an input value followed by optional parameters and returns the transformed value.
- There will be one additional argument to the transform method for each parameter passed to the pipe.
- The @Pipe decorator allows you to define the pipe name that you'll use within template expressions. It must be a valid JavaScript identifier.
- You must include your pipe in the declarations array of the AppModule.

App.PipesDemo.ts

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

@Component({
   selector: 'my-pipes',
   templateUrl : '/app/app.pipesdemo.html' })
export class PipesDemo{

}

@Pipe({name: 'brackets'})
export class BracketsPipe implements PipeTransform {
   transform(value: string, newcase : string = 'n') {
           if ( newcase == 'u')
                   value = value.toUpperCase(); else
                 if ( newcase == "l")
                       value = value.toLowerCase();


           return "[" + value + "]";
   }
}
```

App.module.ts

```
import { NgModule }                    from '@angular/core';
import { BrowserModule } from '@angular/platform-browser'; import {
PipesDemo,BracketsPipe} from './app.pipesdemo';

@NgModule({
   imports: [ BrowserModule], declarations: [
   PipesDemo , BracketsPipe], bootstrap: [ PipesDemo ]
})
export class AppModule { }
```

app.pipesdemo.html

```
{{ "Angular" | brackets }} <br/>
{{ "Angular" | brackets:'u' }}
```

Simple Application

The following simple application shows how to put all pieces of Angular Application together.

1. Create a component in TypeScript – **app/course.component.ts**

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

@Component({
    selector: 'angular-course',
    templateUrl : 'app/course.component.html', })
export class CourseComponent { name =
        'Angular';
        topics = ["Components" ,"Data Binding" , "DI"
                                ,"Forms" , "Http"];

      }
```

2. Create a module in **app/app.module.ts**

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

@NgModule({
    imports: [ BrowserModule], declarations: [
    FirstComponent],          bootstrap:          [
    FirstComponent ]
})
export class AppModule { }
```

3. Create HTML page that contains the template for

```
<html>
<body>
```

```
<h1> {{name}} </h1> <h2>Important
Topics </h2> <ul>
        <li *ngFor="let topic of topics"> {{ topic }} </li> </ul>
</body>
</html>
```

4. Create **app/main.ts** to bootstrap AppModule.

```
import { platformBrowserDynamic }
                          from '@angular/platform-browser-dynamic'; import {
AppModule }
                        from './app.module';

platformBrowserDynamic().bootstrapModule(AppModule);
```

5. Create main hosting page – **index**.**html**

```
<!DOCTYPE html>
<html>
    <head>
        <title>Angular QuickStart</title>
        <base href="/">
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-
scale=1">
        <link rel="stylesheet" href="styles.css">

        <!-- Polyfill(s) for older browsers -->
        <script src="node_modules/core-js/client/shim.min.js"></script>

        <script src="node_modules/zone.js/dist/zone.js"></script>
        <script src="node_modules/systemjs/dist/system.src.js"></script>

        <script src="systemjs.config.js"></script>
        <script>
          System.import('main.js').catch(function(err){
console.error(err); });
        </script>
    </head>

    <body>
        <angular-course></angular-course>
    </body>
</html>
```

Template reference variables (#var)

- A template reference variable is often a reference to a DOM element within a template. It can also be a reference to an Angular component or directive or a web component.
- Use the hash symbol (#) to declare a reference variable.
- You can refer to a template reference variable anywhere in the template. The phone variable declared on this <input> is consumed in a <button> on the other side of the template

```
<input #phone placeholder="phone number">

<button (click)="callPhone(phone.value)">Call</button>
```

Template Statement

A template **statement** responds to an **event** raised by a binding target such as an element, component, or directive.

```
<button (click)="deleteHero()">Delete hero</button>
```

The template statement parser differs from the template expression parser and specifically supports both basic assignment (=) and chaining expressions (with ; or ,).

However, certain JavaScript syntax is not allowed:

- new
- increment and decrement operators, ++ and --

- operator assignment, such as += and -=
- the bitwise operators | and &
- the template expression operators

The statement context may also refer to properties of the template's own context.

In the following examples, the template $event object, a template input variable (let hero), and a template reference variable (#heroForm) are passed to an event handling method of the component.

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

<button *ngFor="let hero of heroes"
(click)="deleteHero(hero)">{{hero.name}}</button>

<form #heroForm (ngSubmit)="onSubmit(heroForm)"> ...
</form>
```

Template statements cannot refer to anything in the global namespace. They can't refer to window or document. They can't call console.log or Math.max.