Contents

Angular:

Is written in TypeScript (TS)

TS is a pre-compiler

Developed by Microsoft, open source

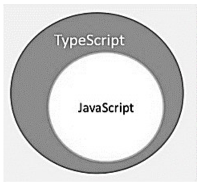
Browser does not recognize TS
It compiles in to JavaScript that can then be used (in .NET Core etc.)
Angular 2/4/5 is very different from Angular1 (AngularJS)

TypeScript:

1) What is TS and why to use it?

What is TypeScript?

By definition, “TypeScript is JavaScript for application-scale development.”
TypeScript is a strongly typed, object oriented, compiled language
It was designed by Anders Hejlsberg (designer of C#) at Microsoft
TypeScript is both a language and a set of tools
TypeScript is a typed superset of JavaScript compiled to JavaScript
In other words, TypeScript is JavaScript plus some additional features.



Features of TypeScript

TypeScript is just JavaScript. TypeScript starts with JavaScript and ends with JavaScript. Typescript adopts the basic building blocks of your program from JavaScript. Hence, you only need to know JavaScript to use TypeScript. All TypeScript code is converted into its JavaScript equivalent for the purpose of execution.

TypeScript supports other JS libraries. Compiled TypeScript can be consumed from any JavaScript code. TypeScript-generated JavaScript can reuse all of the existing JavaScript frameworks, tools, and libraries.

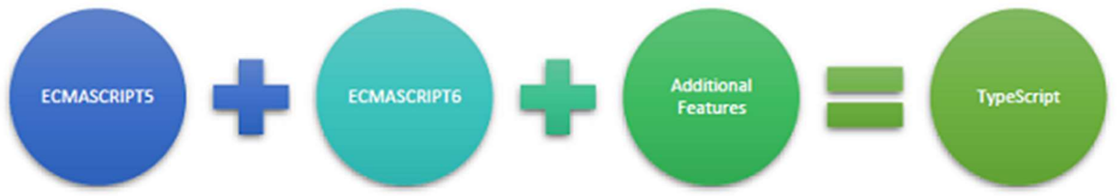
JavaScript is TypeScript. This means that any valid **.js** file can be renamed to **.ts** and compiled with other TypeScript files.

TypeScript is portable. TypeScript is portable across browsers, devices, and operating systems. It can run on any environment that JavaScript runs on. Unlike its counterparts, TypeScript doesn’t need a dedicated VM or a specific runtime environment to execute.

TypeScript and ECMAScript

The ECMAScript (ES) (**European Computer Manufacturers Association** (**ECMA**)) specification is a standardized specification of a scripting language. There are six editions of ECMA-262 published.

Version 6 of the standard is codenamed "Harmony". TypeScript is aligned with the ECMAScript6 specification.



The ECMAScript specification is a standardized specification of a scripting language developed by Brendan Eich of Netscape; initially it was named Mocha, later LiveScript, and finally JavaScript. In December 1995, Sun Microsystems and Netscape announced JavaScript in a press release.

Why Use TypeScript?

TypeScript is superior to its other counterparts like CoffeeScript and Dart programming languages in a way that TypeScript is extended JavaScript. In contrast, languages like Dart, CoffeeScript are new languages in themselves and require language-specific execution environment.

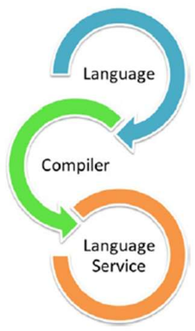
The benefits of TypeScript include –

- **Compilation** – JavaScript is an interpreted language. Hence, it needs to be run to test that it is valid. It means you write all the codes just to find no output, in case there is an error. Hence, you have to spend hours trying to find bugs in the code. The TypeScript transpiler provides the error-checking feature. TypeScript will compile the code and generate compilation errors, if it finds some sort of syntax errors. This helps to highlight errors before the script is run.
- **Strong Static Typing** – JavaScript is not strongly typed. TypeScript comes with an optional static typing and type inference system through the TLS (TypeScript Language Service). The type of a variable, declared with no type, may be inferred by the TLS based on its value.

Components of TypeScript

At its heart, TypeScript has the following three components –

- **Language** – It comprises of the syntax, keywords, and type annotations.
- **The TypeScript Compiler** – The TypeScript compiler (tsc) converts the instructions written in TypeScript to its JavaScript equivalent.
- **The TypeScript Language Service** – The "Language Service" exposes an additional layer around the core compiler pipeline that are editor-like applications. The language service supports the common set of a typical editor operations like statement completions, signature help, code formatting and outlining, colorization, etc.



Integrates easily in to JS projects
Can be used in Angular and WinJS (Windows Library for JS)
It's an open source lib by Microsoft
Powerful features like classes and modules
integrated with Gulp or Grunt (task runners)
    Grunt: Task runner. Managing the flow of developing projects.
    Gulp: Similar. Compile TS code to JS automatically
For small projects, JS is fine. But for larger projects, use TS for better organization and control and debugging features.

www.typescriptlang.org/handbook

2) Install TS and basic compiling

Requires Node.js
Check if NodeJS + NodeJS Package Manager (npm) is installed or not
    node -v
    npm -v
    Or install from https://nodejs.org/en/
Install TS:
    npm install -g typescript
        OR go to typescriptlang.org and install the editor
        -g means install globally, so available to all projects.
    run TSC to check TS installation
    tsc –version
Transpiler / Transpilation
    Process of compiling Typescript (.ts) to Javascript (.js)

Coding using TS

1) First TS (using VS Code):

    Create a folder "helloWorld" and navigate to that
    Create file called helloWorld.ts
        console.log("Hello World");
        from cmd prompt, TSC helloWorld.ts
        creates helloWorld.js

run with node helloWorld.js

Add a func:

```
// remove the console.log("Hello World");
function hello(string: String) {
    console.log("Hello " + string);
}

hello("Ajay");
```

from cmd: node helloWorld.js

shows Hello World

but node helloWorld.ts will not work as node does not recognize TS, only JS

so, TSC helloWorld.ts, updates the js file

run node helloWorld.js

shows Hello Ajay

change code to: hello(12)

tsc helloWorld.ts

error: number not assignable to param of type string

change to hello("Neo");

compile and run. works.

Transpiling Code from TS to JS with a Class:

- Create a file person.ts with the following code:

```
class Person {
  public firstName: string;
  public lastName: string;

  constructor (firstName: string, lastName: string) {
    this.firstName = firstName;
    this.lastName = lastName;
  }
}
```

- Compile with tsc person.ts
- Generates person.js
- Open it and show
- Add a private field and a function accessor

```
class Person {
  public firstName: string;
  public lastName: string;
  private _fullName: string;

  constructor (firstName: string, lastName: string) {
    this.firstName = firstName;
    this.lastName = lastName;
  }

  get fullName(): string {
    return `${this.firstName} ${this.lastName}`;
  }
}
```

- Compile with tsc person.ts
  - Gives an error
  - The proper way to define an accessor in javascript is using Object.defineProperty() that is only available in ES5 and onward, but as default, the transpiler tries to create ES3 code. To overcome this problem we have to tell the transpiler that it should target ES5 and not ES3.
- Compile with tsc person.ts --target ES5
  - Works! Show the person.js file
- Change the code to the following:

```
class Person {
    public firstName: string;
    public lastName: string;
    private _fullName: string;

    constructor(firstName: string, lastName: string) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    fullName(): string {
        return `${this.firstName} ${this.lastName}`;
    }
}

var obj = new Person('John', "Smith");
console.log(obj.fullName());
```

- Compile with tsc person.ts
- Run with node person.js
  - Works!!!

1.1: Sample with Interface

A) greeter.ts:

```
class Student {
        fullName: string;
        constructor(public firstName: string, public middleInitial: string, public lastName: string) {
                this.fullName = firstName + " " + middleInitial + " " + lastName;
        }
}
```

```
interface Person {
	firstName: string;
	lastName: string;
}
function greeter(person : Person) {
	return "Hello, " + person.firstName + " " + person.lastName;
}
let user = new Student("Jane", "M.", "User");
document.body.innerHTML = greeter(user);
```

- Run TSC greeter.ts
- Generates greeter.js

B) greeter.html

```
<!DOCTYPE html>
<html>
	<head><title>TypeScript Greeter</title></head>
	<body>
		<script src="greeter.js"></script>
	</body>
</html>
```

C) Open greeter.html in browser.

1) Type system of TS:

Basic types intro
new folder, types
file types.ts
	define type with a colon
	var n: Number = 1;
	compile and show .js
		does not have "Number"
	n = "Ajay"
	compile, gives error
	but .js has n = "Ajay". So TS is basically warning us that it "may" not work, but JS it is fine
	so, fix it anyways
	either change value to numeric or type to "any"
		do not use "any" too much

2) Built-in types:

Boolean:
	open types.ts
	var isWinter : Boolean = false;

```
isWinter = 123;
compile: error??? No.

comment isWinter = 123;
var count: number = 5;
var name: string = "Neo";
```

Array:

```
var names : string[] ["Ajay", "Neo"];
This will give error:
        var name s = string[] ["Ajay", "Neo", 5];
to store multiple types, define as "any": var names : any[] ["Ajay", "Neo", 5];
```

Enum:

```
enum Starks { Jon, Bran, Heather, Catlyn };
var cat : Starks = Starks.Catlyn;
```

void:

```
function getName() : string {
        return "Ajay";    // return 1 will throw an error.
}

use "void" when not returning anything:
function getName() : void {
        console.log("some message");
}
```

3) Interfaces:

Interface is a defn for an object that tells TS what that obj is going to be
It is a blueprint for an object
in interfaces.ts:

```
function printName(stark) {
        console.log(stark.name);
}

printName({name: "Ajay"});
printName({label: "Joe"});
```

compile: success. but run will give error.
run:

```
Ajay
undefined
```

difficult to determine what went wrong. Interface to the rescue.
in interfaces.ts:

```
interface Stark {
    name: string;
}

function printName(stark: Stark) {
    console.log(stark.name);
}

printName({name: "Ajay"});
printName({label: "Joe"});
```
Compile: error, "name" is missing in type.
change to:
```
    printName({name: "Ajay"});
    printName({name: "Joe"});
```
runs.

modify:
```
    interface Stark {
        name: string;
        age: number;
    }
```
compile: error. 2 errors. both names do not have property age, as it is "required".
to make age optional:
```
    age?: number
```
compile: works

4) Classes

create classes.ts
```
    class Stark {
        name: string;
    }

    var ned = new Stark();
```
compile. look at the .js code
```
    ned.saying = "Winter is coming";
```
compile: error "property does not exist"
```
    class Stark {
        name: string;
        saying: string;
    }
```
// cannot have optional vars in classes
Classes can have ctors
```
    class Stark {
        name: string = "Brandon";        // default value.
        saying: string;
```

```
            static castle: string = "Winterfell"!;

            constructor() {
                this.saying = "Winterfell!";
            }
        }

        console.log(Stark.castle);
```
compile + run: winterfell!

add methods to classes:
```
        class Stark {
            name: string = "Brandon";       // default value.
            saying: string;
            static castle: string = "Winterfell"!;

            constructor() {
                this.saying = "Winterfell!";
            }

            hello(person: string) {
                console.log("Hello " + person);
            }
        }
        //console.log(Stark.castle);
        var ned = new Stark();
        ned.hello("Robert");
```

5) Inheritance:

use "extends"

create extends.ts
```
        class Person {
            name: string;

            constructor(name:string) {
                this.name = name;
            }

            dance() {
                console.log(this.name + " is dancing.");
            }
        }

        var person = new Person("Bryan");
        person.dance();
```

compile + run. works.

another class:

```
class AwesomePerson extends Person {
    // override func.
    dance() {
        console.log("Sooo awesome!");
    }
}

var robb = new AwesomePerson("Robb");
robb.dance();
```

compile+run:

```
Sooo awesome!
```

call base class' func:

```
class AwesomePerson extends Person {
    // override func.
    dance() {
        console.log("Sooo awesome!");
        super.dance();   // invoke func in base class.
    }
}

var robb = new AwesomePerson("Robb");
robb.dance();
```

compile+run:

```
Sooo awesome!
Robb is dancing
```

6) Modules

share code b/w files.

create timesTwo.ts

```
function timesTwo(n:number) {
    return n * 2;
}
```

create new TS file, util.ts

```
console.log(timesTwo(9));
```

tsc util.ts

```
error. can't find timesTwo
```

have to define a dependency.

so, make timesTwo a module

```
module Utility {
    export class Useful {
        timesTwo(n:number) {
            return n * 2;
```

```
                        }
                }
        }
in util.ts:
        /// <reference path="timesTwo.ts" />
        var use = new Utility.Useful();
        console.log(use.timesTwo(9));
compile: no error.
node util.js: error: Utility is not defined.
        because we compiled only util.ts and not timesTwo.ts
        We must combine both together and compile+run
        TSC timesTwo.ts   util.ts   --out util.js
        open util.js
        node util.js
                shows 18
```

- **More Module related information**

  executed within their scope, not globally
  unless exported and then imported in another module

  Example #1: use the expClass and impClass ts files.
  Another example: use the IShape, ICircle, ITraingle & TestShape ts files.

```
// MyClass.ts
#1
export class MyClass {
        myFunction(x: string) {
                return x;
        }
}

#2:
class MyClass {
        myFunction(x: string) {
                return x;
        }
}
export{MyClass};

// open another module file: importmodule.ts
import { MyClass } from "./MyClass";      // relative path

let object = new MyClass();
console.log(object.myFunction("value"));
```

```
#3: Export as a different name.
class MyClass {
        myFunction(x: string) {
                return x;
        }
}
export{MyClass as MainClass};

// open another module file: importmodule.ts
import { MainClass } from "./MyClass";   // relative path

let object = new MainClass();
console.log(object.myFunction("value"));
```

7) enums

```
enum keyword
enum Values {
        First = 1,
        Second = 2,
        Third = 3,
        Fourth = 4
}

let first = Values.First;                          // 1
let nameOfFirst = Values[Values.First];   // "First"
```

8) Generics

```
function value(myval: number): number {
        return myval;
}

function value2(myval2: any): any {
        return myval2;
}

let val0 = value(6);                  // works
let val1 = value("test");  // compile error
let val2 = value2("test");           // works
let val3 = value2(24);               // works

function type<T>(parameter: T): T {
        return parameter;
}

function identity<T>(arg: T): T {
	//console.log(arg.length);    // Error. T doesn't have length.
	return arg;
}

var output = identity<string>("Ajay");
console.log(output);

var output2 = identity("Ajay Singala");
console.log(output2);

// Array Generic
function identityLog<T>(arg: T[]): T[] {
	console.log(arg.length);    // Array has length. No error.
	return arg;
}

var id = identityLog(["ajay", "neo", "trinity"]);
console.log(id[0]);
console.log(id[1]);
console.log(id[2]);
```

9) Namespaces:

internal modules => namespaces
external modules => modules

similar to C# to organize code
can be done in same file or separate files.

```
// namespaces.ts file
namespace Primary {
	export interface PrimaryInterface {
		isTrue(x: string): boolean;
	}
}

// importnamespaces.ts file
/// <reference path="namespaces.ts" />
namespace Primary {
	export class SecondNamespace implements PrimaryInterface {
		isTrue(x: string) {
```

```
                if(x === "true") {
                    return true;
                }
                return false;
            }
        }
    }

    var sn = new SecondNamespace();
    console.log(sn.isTrue("true"));
    console.log(sn.isTrue("nope"));
```

10) Iterators:

```
    var listItems = [5, 6, 7];
    for (var x1 in listItems) {
        console.log(x1);        // output: "0", "1", "2"
    }
    // show values.
    for (var x2 in listItems) {
        console.log(listItems[x2]);        // output: 5, 6, 7
    }
```

Angular:

Check if NodeJS + NodeJS Package Manager (npm) is installed or not

```
node -v
npm -v
```

If not installed, download+install from nodejs.org

include npm as well

To install the Angular CLI:

https://cli.angular.io

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

Install the Angular CLI:

```
    npm install @angular/cli -g
        -g is for "global"
    ng -v    // Verify it is installed
    ng           // lists tons of options to use
```

1) Create project:

```
        cd <<project folder>>
        ng new <<name>> --style=scss --routing
        For e.g.;
                ng new ng5 --style=scss --routing
                        --style:  to use css style
                                scss pronounced as "sas" file
                        --routing:      to integrate and setup routing by default in our project

        cd ng5
        ng serve        // builds the project. Note the port number where the app will be accessible.
For e.g.;
        // Keep the ng serve console running at all times as it is required to run the app
        // For any other ng tasks, open a new console and work from there
        localhost:4200
        Open browser, navigate to http://localhost:4200 to test

        Work from the src/app folder
        open a new console window
        cd src/app
        add <app-root>Loading...</app-root> to the index.html
        app.component.ts        // .ts =TypeScript
                divided in 3 sections:
                        1. imports
                        2. @Component decorator
                                define structure of component
                        3. Actual component class
        create a new component using ng cli:
                ng generate component <<compnent name>>
                ng generate component home
                        generates/updates files:
                                src/app/home/home.component.html
                                src/app/home/home.component.spec.ts
                                src/app/home/home.component.ts
                                src/app/home/home.component.scss
                                src/app/app.module.ts  // Updated
        create one more component, this time using shorthand format:
                ng g c about
                        g = generate
                        c = component
        nesting components:
                // nest home in to app component
                // HTML is aka TEMPLATE
                open app.component.html
                        add routerLink to home and about
                        remove all code except last line --> <router-outlet>
```

add the following just above <router-outlet>:

```
<ul>
    <li><a routerLink="">Home</a></li>
    <li><a routerLink="about">About</a></li>
</ul>

<app-home></app-home>                    // This will be the "selector" in home.component.ts
```

Save

Reflects in browser immediately. But link does not work. Shows nothing for Home.

Notes:

```
in about.component.ts:
    selector: 'app-about',
in home.component.ts:
    selector: 'app-home',
in app.component.ts:
    selector: 'app-root',
in index.html:
    <body>
        <app-root></app-root>
    </body>
```

in app-routing.module.ts:

```
const routes: Routes = [
  {
    path: '',
    component: HomeComponent
  },
  {
    path: 'about',
    component: AboutComponent
  }
];
```

Save. Switch to browser. Click on Home. Displays contents of home.component.html without postback

SPA: Single Page Application

Repeat for About component.

Notes:

- main.ts - compilation info

- Types of Directires:
    - Component Directives: @Component
    - Structural Directives: ngIf, ngFor, ngSwitch
    - Atrribute Directives: ngStyle, ngClass
- Properties of @Component
    - selector, template, templateUrl, styels, styleUrls, animations
- RouterModule.forRoot(routes) – check
- <router-outlet>

```
<router-outlet></router-outlet>
<router-outlet name=”sidebar”></router-outlet>

const routes: Routes = [
  { path: '', component: HomeComponent },
  { path: 'about', component: AboutComponent, outlet: ‘sidebar’}
]
```

2) Templating & Styling

```
in the html file, instead of:
        templateUrl: '...',
enter actual HTML in back-ticks as:
        template: `<h1>Hello</h1>`,

Same for styles. Instead of:
  styleUrls: ['./home.component.scss']
enter:
  styles: [`
    p { font-weight: bold }
    div { color: gray; }
  `]
```

Note: Remember to change back to templateUrl and styleUrls.

Modify the scss for:
src/style.scss
src/app/home/home.scss

3) Interpolation, Property Binding & Event Binding

Interpolation: {{ }}

displaying property values defined in the component on the html
Demo:
Define a property itemCount: number = 4 in home.c.ts

Add ({{ itemCount }}) to home.c.html as:

<p>Your bucket list ({{ itemCount }})</p>

Demo 2:

Define property btnText: string = 'Add an item' in home.c.ts

<input type="submit" class="btn" value="{{ btnText }}"> in home.c.html

Property binding: []

Ex. 1: in home.c.html, change to (w/o the {{..}}, but with [&]):

<input type="submit" class="btn" [value]="btnText">

Ex. 2: In the template:

```
<button [disabled]="buttonStatus">My Button</button>
```

In the component class:

```
buttonStatus = true
```

OR

Template: <button [disabled]="buttonStatus == 'enabled'">My Button</button>

Component: buttonStatus = 'enabled';

The above is 1-way binding, from component to html.

2-way data binding:

To retrieve and set the value of a text box, for e.g.

Have to use ngModel after importing FormsModule to make it work.

using [(ngModel)]:

import the module

add to imports array

in app.module.ts:

```
import { FormsModule } from '@angular/forms';
:
@NgModule({
  declarations: [
        :
  ],
  imports: [
        :
        FormsModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
```

in home.component.ts

goalText: string = 'My first life goal';

in home.component.html setup 2-way data binding:

```
            <input type="text" class="txt" name="item" placeholder="Life goal.."
[(ngModel)]="goalText">
```

Temporarily, also add a span after the input box:

```
             <br><span>{{ goalText }}</span>
```

On UI, when entering text, it also updates the span element.

Event binding (for e.g; button click):

Delete the <span> from previous demo.

home.component.html:

```
            <input type="submit" class="btn" [value]="btnText" (click)="addItem()">
```

home.component.ts, define an array and remove init value of itemCount.

```
            itemCount: number;
            goals = [];

            // Lifecycle hook when the component loads.
            ngOnInit() {
                        this.itemCount = this.goals.length;
            }

            addItem() {
                        this.goals.push(this.goalText);
                        this.goalText = '';
                        this.itemCount = this.goals.length;
            }
```

In the UI, this will increment the item count, but not the list of goals.
To update the list, use ngFor:
home.component.html:

```
            <div class="col">
                        <p class="life-container" *ngFor="let goal of goals">
                                    {{ goal }}
                        </p>
            </div>
```

4) Animation:

Animate each life goal item as it gets added.

go to cmd console (if error, open with Run as Administrator)
install animations library

```
            npm install @angular/animations@latest --save
```

open app.module.ts

```
            import { BrowserAnimationsModule } from '@angular/platform-browser/animations';
            :
```

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

open home.component.ts

```
NOTE: sometimes the animation fails if we don't specify {optional: true}.
// Reset user settings in preferences if required.
import { trigger, style, transition, animate, keyframes, query, stagger } from
'@angular/animations';
:
@Component({
  :
  animations: [
        trigger('goals', [
              transition('* => *', [
                    // when something enters the DOM
                    query(':enter', style({ opacity: 0}), {optional: true}),
                    // delay each subsequent DOM element by 300ms.
                    query(':enter', stagger('300ms', [
                    animate('.6s ease-in', keyframes([
                          // come from top.
                          style({opacity: 0, transform: 'translateY(-75%)', offset: 0}),
                                style({opacity: 0.5, transform: 'translateY(35px)',
offset: 0.3}), // happens in the middle of the 0.6s duration.
                                style({opacity: 1, transform: 'translateY(0)', offset:
1}),
                          ]))
                    ]), {optional: true})
              ])
        ])
  ]
})
```

apply animations to the template

home.component.html

```
// @goals is the name of the animation.
<div class="container color-light" [@goals]="goals.length">
```

for initial testing, add some values to the goals array

open home.component.ts

```
	goals = ['My first life goal', 'I want to climb a mountain', 'Go skiing'];
```
refresh browser

Remove items from list when you click on them:
home.component.html
```
	<p class="life-container"  *ngFor="let goal of goals; let i = index" (click)="removeItem(i)" >
```
home.component.ts
```
	removeItem(i) {
		this.goals.splice(i, 1);	// typical javascript.
	}
```
add one more animation that will be the reverse of previous one. Just add another ""leave" "query" to the current "animations" array.
home.component.html
```
	animations: [
	  trigger('goals', [
		transition('* => *', [
			query(':enter', style({ opacity: 0}), {optional: true}),
			query(':enter', stagger('300ms', [
				animate('.6s ease-in', keyframes([
					style({opacity: 0, transform: 'translateY(-75%)', offset: 0}),
					style({opacity: 0.5, transform: 'translateY(35px)', offset: 0.3}),
					style({opacity: 1, transform: 'translateY(0)', offset: 1}),
				]))
			]), {optional: true}),

			// This is the new one added. Reverse of ":enter".
			query(':leave', stagger('300ms', [
				animate('.6s ease-in', keyframes([
					style({opacity: 1, transform: 'translateY(0)', offset: 0}),
					style({opacity: 0.5, transform: 'translateY(35px)', offset: 0.3}),
					style({opacity: 0, transform: 'translateY(-75%)', offset: 1}),
				]))
			]), {optional: true})
		])
	  ])
  ]
})
```

5) Routing

app-routing.module.ts
```
	import { HomeComponent } from './home/home.component';
	import { AboutComponent } from './about/about.component';

	const routes: Routes = [
		{
```

```
                path: '',
                component: HomeComponent
            },
            {
                path: 'about',
                component: AboutComponent
            }
        ];

        @NgModule({
            imports: [RouterModule.forRoot(routes)],
            exports: [RouterModule]
        })
        export class AppRoutingModule { }
```

browser will show home page contents twice

app.component.html

remove <app-home></app-home>

Will work now.

Route parameters and how to retrieve them:

```
            app-routing.module.ts
                const routes: Routes = [
                    {
                        path: '',
                        component: HomeComponent
                    },
                    {
                        path: 'about/:id',
                        component: AboutComponent
                    }
                ];
                can have path: 'about/:id/:whatever' also
        app.component.html
            // use a hard-coded id for now.
            <li><a routerLink="about/48">About</a></li>

        but, how to extract the parameters from the url?
        about.component.ts
            import { ActivatedRoute } from '@angular/router';

            // DI
            export class AboutComponent implements OnInit {

              constructor(private route: ActivatedRoute) {
```

```
                        this.route.params.subscribe(res => console.log(res.id));
                    }

                    ngOnInit() {
                    }
                   }
```

Run and it displays the id in the console (F12).

normally, you would take this id, assign it to a property and call an API.

Router Outlet

<router-outlet>

https://onehungrymind.com/named-router-outlets-in-angular-2/

https://www.smashingmagazine.com/2018/11/a-complete-guide-to-routing-in-angular/

sample code:

D:\Users\AjayS\Tryouts\Angular5Tryouts\ng2-named-router-outlets-master

Component based router navigation:

about.component.ts

```
    import { Router } from '@angular/router';

    //DI
    export class AboutComponent implements OnInit {
        constructor(private route: ActivatedRoute,
            private router: Router) {

            this.route.params.subscribe(res => console.log(res.id));
        }

     ngOnInit() {
     }

     sendMeHome() {
        // The '' maps to the path in app-routing.module.ts for HomeComponent.
        this.router.navigate(['']);
    }
```

about.component.html

```
    <p>
        This is I'm all about. <a href="" (click)="sendMeHome()"><strong>Take me
back</strong></a>
    </p>
```

6) Services / Dependency Injection (DI):

generally used for making http calls or sharing data between components

Share the goals array b/w home and about component.
cmd console:
```
ng generate service data
// ng g s data
// generates src/app/data.service.ts
```
open data.service.ts
```
// have to import the Injectable module
import { Injectable } from '@angular/core';
import { BehaviorSubject } from 'rxjs;

@Injectable()
export class DataService {
	private goals = new BehaviorSubject<any>(['First', 'second']);
	goal = this.goals.asObservable();

	constructor() { }

	changeGoal(goal) {
		this.goals.next(goal);
	}
}
```

// 2 ways to import.
// #1: specify the service as a "provider" in app.component.ts (or any component individually) in
//		the @Component decorator of the class
//	For e.g.; app.component.ts:
```
//		import { MainService } from './main-service';
//		@Component({
//			selector:...
//			templateUrl:...
//			styleUrls:...
//			providers: [MainService]		// comma separated list of services required.
/		})
```
// #2: import it in the app.module.ts and list it as a provider in the "providers" section of @NgModule
//		which makes the service available globally. Then, just import that service in the component class
//		and inject it in the ctor of the component.
app.module.ts:
```
import { DataService } from './data.service';
:
@NgModule({
  :,
            providers: [DataService],                    // comma separated list of services.
            bootstrap: [AppComponent]
          })
    home.component.ts:
          import { DataService } from '../data.service';

          // DI.
          export class HomeComponent implements OnInit {
                :
                goals = [];                    // init values set in service earlier.
                constructor(private _data: DataService) { }
                :
                ngOnInit() {
                      this._data.goal.subscribe(res => this.goals = res);            //
_data.goal is asObservable
                      this.itemCount = this.goals.length;
                      this._data.changeGoal(this.goals);
                }

                addItem() {
                      :
                      this._data.changeGoal(this.goals);                    // refresh.

                }

                removeItem() {
                      :
                      this._data.changeGoal(this.goals);                    // refresh.
                }
          }
    about.component.ts:
          import { DataService } from '../data.service';
          // DI.
          export class AboutComponent implements OnInit {
                :
                goals: any;

                constructor(private route: ActivatedRoute,
                      private router: Router,
                      private _data: DataService) {

                      this.route.params.subscribe(res => console.log(res.id));

                      }
                :
```

```
                                        ngOnInit() {
                                                    :
                                                    this._data.goal.subscribe(res => this.goals = res);             //
_data.goal is asObservable
                                        }
                                        :
                              }
                    about.component.html:
                              :
                              <ul>
                                        <li *ngFor="let goal of goals">
                                                    {{ goal }}
                                        </li>
                              </ul>
```

7) Deploying

```
// generates files that are huge
ng build
// creates a "dist" folder
// use PROD flag instead
ng build --prod
// get an ~89% reduction in size.
```

Just upload the contents of the dist folder to a server.

but if uploading to a sub-folder like http://server/folder, then you have to run the ng build cmd as:

```
          ng build --prod --base-href="<<url>>"
```

or else, it will not run properly.

Another way is thru git hub pages:

```
npm install -g angular-cli-ghpages
// also need git (git-scm.com/downloads) and setup a repo
// add code to git:
```

create a github a/c, create a new repo.
No need to run git init and git add README.md

```
git add .
git commit -m "comment"
git remote add origin git@github.com:username/ng5.git
git push -u origin master

// re-run ngbuild cmd
// specify url of target with username.
ng build --prod --base-href="https://username.github.io/ng5/"
angular-cli-ghpages
```

navigate to https://username.github.io/ng5/ to see the deployed app.

More Angular topics:

Polyfills:

Angular supports most recent browsers. This includes the following specific versions:

Browser	Supported versions
Chrome	latest
Firefox	latest
Edge	2 most recent major versions
IE	11,10,9
IE Mobile	11
Safari	2 most recent major versions
iOS	2 most recent major versions
AndroidNougat (7.0)	
Marshmallow (6.0)	
Lollipop (5.0, 5.1)	
KitKat (4.4)	

Enabling polyfills:
Angular is built on the latest standards of the web platform. Targeting such a wide range of browsers is challenging because they do not support all features of modern browsers.

You compensate by loading polyfill scripts ("polyfills") for the browsers that you must support.
Enabling Polyfills:

Angular CLI users enable polyfills through the src/polyfills.ts file that the CLI created with your project.

This file incorporates the mandatory and many of the optional polyfills as JavaScript import statements.

The npm packages for the mandatory polyfills (such as zone.js) were installed automatically for you when you created your project and their corresponding import statements are ready to go. You probably won't touch these.

But if you need an optional polyfill, you'll have to install its npm package. For example, if you need the web animations polyfill, you could install it with npm, using the following command (or the yarn equivalent):

```
npm install --save web-animations-js
```

Then open the polyfills.ts file and un-comment the corresponding import statement as in the following example:

src/polyfills.ts
/**
* Required to support Web Animations `@angular/platform-browser/animations`.
* Needed for: All but Chrome, Firefox and Opera. http://caniuse.com/#feat=web-animation
**/
import 'web-animations-js'; // Run `npm install --save web-animations-js`.

If you can't find the polyfill you want in polyfills.ts, add it yourself, following the same pattern:

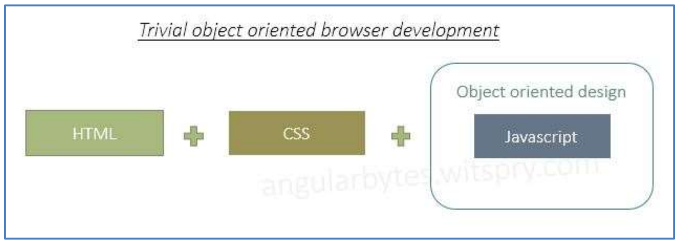
install the npm package
import the file in polyfills.ts

Code Linting

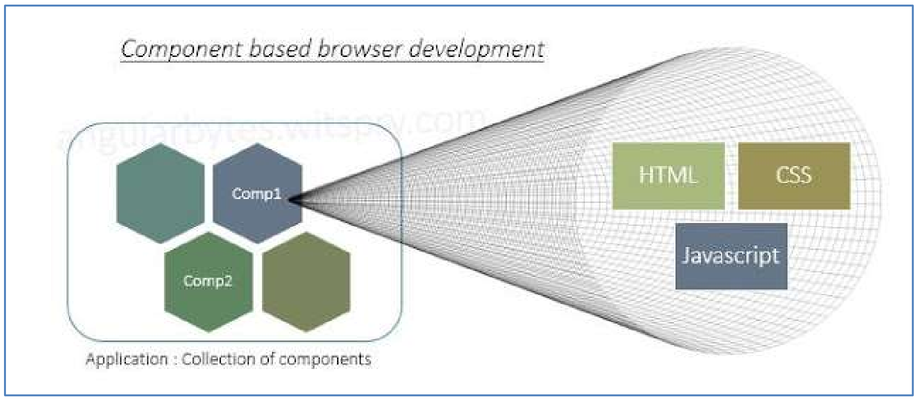
use TSLint
https://marketplace.visualstudio.com/items?itemName=eg2.tslint
npm install -g tslint

Angular Architecture

Diagram



An application is a collection of components and a component can contain several other components.


Component based browser development
angularbytes.witspry.com
Comp1
Comp2
HTML
CSS
Javascript
Application : Collection of components

A typical Angular Component (show text file):

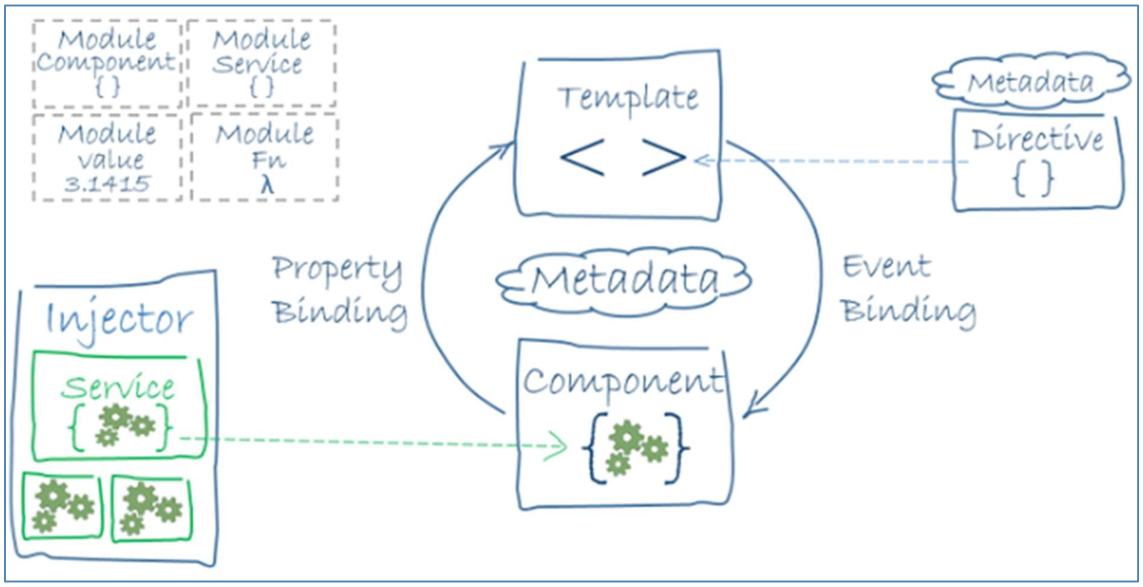
```
import { Component } from '@angular/core';
```

```
export class Player {
  id: number;
  name: string;
}

@Component({
  selector: 'my-app',
  template: `
    <h1>{{title}}</h1>
    <h2>{{player.name}} details!</h2>
    <div><label>id: </label>{{player.id}}</div>
    <div>
        <label>name: </label>
        <input [(ngModel)]="player.name" placeholder="name">
    </div>
  `
})
export class AppComponent {
  title = 'Chess Tournament';
  player: Player = {
    id: 1,
    name: 'Jeff'
  };
}
```

As you can see there is a model class *Player* having two properties id and name. Also, there is one html template where some properties have been mentioned under different brackets. There is one *AppComponent* class where properties data has been assigned to them.

Architecture diag:



A single unit of application in Angular is called Module . A module consists of different components, services and functions. Angular's own module is called *ngModule* which should be imported in the application before using Angular and its features.

Each Component class is associated with metadata. In the above example we have selector and template as a part of Angular metadata.

Angular also provides bindings to bind component class property data to the template.

Angular's provides inbuilt and custom services that can be injected in any component to make use of service functionalities.

Angular provides many kinds of data binding

We can group all bindings into four categories by the direction in which data flows. Each category has its distinctive syntax:

Interpolation

Any property in Component class can be shown as text through Interpolation written as {{}}

Property Binding

It is similar to Interpolation but it is a preferable way of doing one way binding, especially in case of styles and attribute bindings. It has a syntax []

Event Binding

It binds and event to a Component function. The events are standard browser events such as Click, DoubleClick etc.

Two-way binding

It binds a Component property to an input element. The changes gets reflected in the property at the same time. This is actually a combination of One-way binding and Event-binding. It is written as Banana-in-a-box syntax [(..)]. The outer braces shows One-way binding and the inner brace shows Event binding.

Example (show text file)

```
<input type="text" [value]="myName">
<button (click)="colorChange(textColor)" [style.color]="textColor">Change My Colo
r</button>
<input type="text" [(ngModel)]="textColor">

<table border=2>
    <!--  expression calculates colspan=2 -->
    <tr>
        <td [attr.colspan]="1 + 1 + 1">Left - Center - Right</td>
    </tr>
    <tr>
        <td>1</td>
        <td>2</td>
        <td>3</td>
    </tr>
</table>
```

ng new AngDemo
cd AngDemo
folder structure
    node_modules: all ng modules, components etc.
    src: source code
    app: app.* files
ng serve
http://localhost:4200

Modules

https://medium.com/@cyrilletuzi/understanding-angular-modules-ngmodule-and-their-scopes-81e4ed6f7407

upper left hand corner of the arch diag
app.c.ts:
    title = ‘My First App Works!”;
browser. Title updated automatically
comment all code in app.c.html
add following:
```
<h1>
    {{ title }}
    {{ 10 * 15 }}
<h1>
```

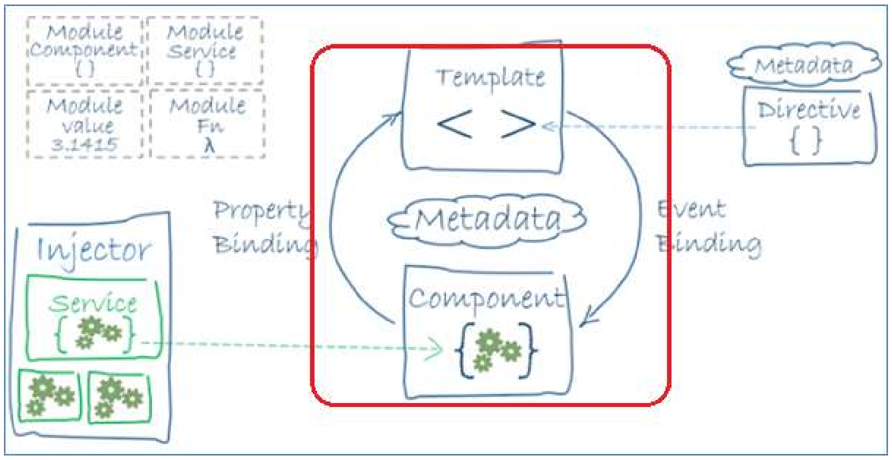
Browser: updates auto

At top of main.ts, add the following to support backward browser compatibility:
    import ‘./polyfills.ts’;

Components

- The main way we build and specify elements ad logic on the page, through both custom elements and attributes that add functionality
- Controls patch of a screen called a *view*
- Component decorator allow you to mark a class as an Angular component, to determine how components are processed, initiated and used at runtime.
- C are the most basic building blocks of a UI in Ang apps
- An Ang app is a tree of Ang components
- Ang components are a subset of directives. Unlike directives, C have a template and only one C can be initiated per element in a template.

- C must belong to an ngModule in-order to be used by another C or app.



- Show app.c.ts
- Explain @Component, selector, template, export etc.

Classes and Components

Create a new component:
ng g component my-component
(ng g c my-component)

ng g class my-class
show the my-component component, and html files

Now, how do we utilize it in the app?
Open app.module.ts
    MyComponent is auto-scafolded
Open my-class, add a property:

```
export class MyClass {
    name:string
}
```

Open app.m.ts
NOTE: Typically, you do not import classes in app.module.ts, but this is just to demo DI.

```
    import { MyClass } from './my-class';
```

```
providers: [MyClass],
```

list as provider so that we can inject in our app.c.ts

app.c.ts:

```
import { MyClass } from './my-class';
```

and

```
export class AppComponent {
  title = 'My first App works!';

  constructor(private myclass: MyClass) {

  }

  test() {
    this.myclass.name = "Worked!";
  }
}
```

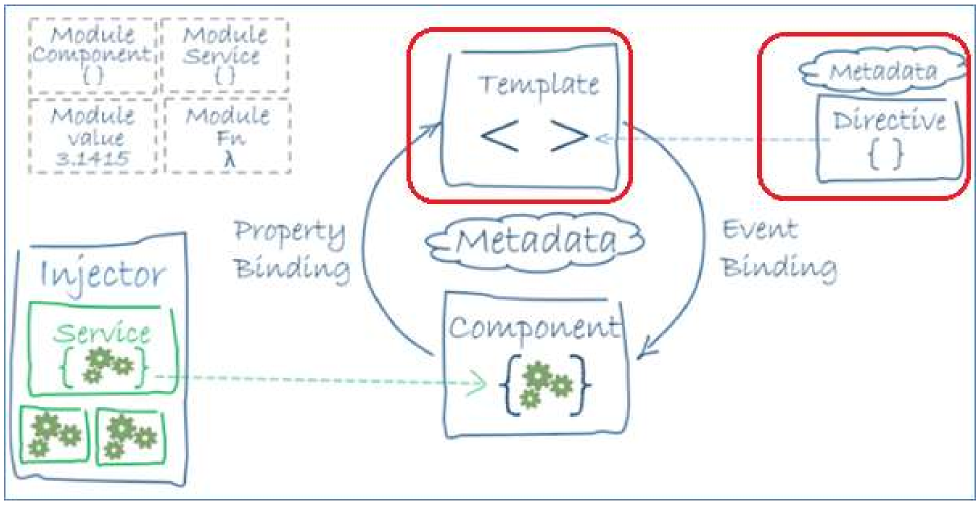
Open app.c.html, add:

```
<button (click)="test">Test Submit</button>
<br>
<h1>{{ myclass.name }}</h1>
```

Browser: click button

Templates

Template is a form of html that tells Angular how to render the component.
Templates are just the UI aspects of our component



2 ways to render template:

1. In the “template” section of @Component decorator
2. Using an html and providing “templateUrl” in the @Component decorator

What if I want to change the startup template of the app?

Open app.m.ts

Change the “bootstrap” value: bootstrap: [MyComponentComponent]
Browser: nothing happens
Have to change the tag in index.html. Open index.html
<app-root> ties to app.component.ts
Replace with selector from MyComponent:

```
<app-my-component>Loading...</app-my-component>
```

In my-component.c.html, add at top:

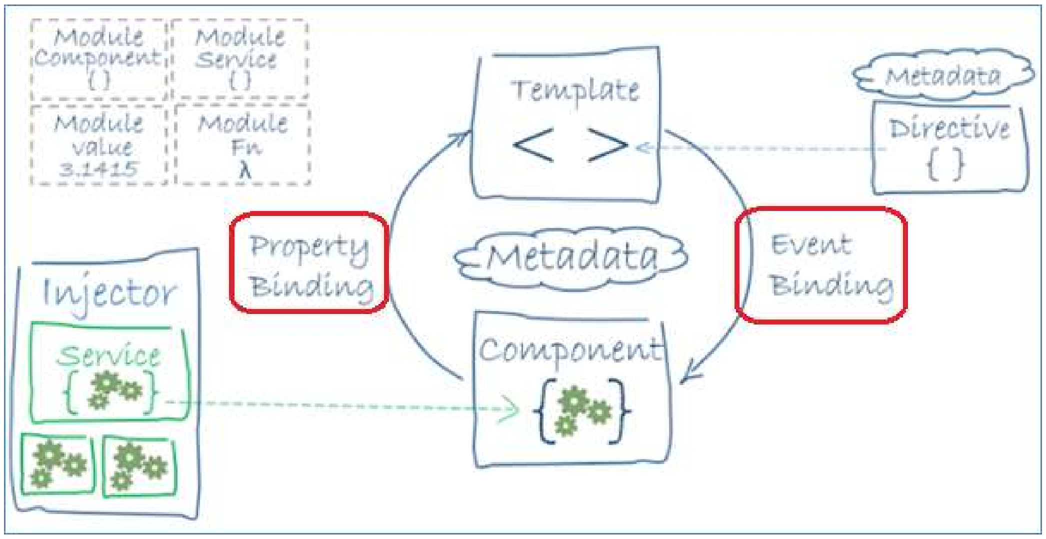
```
<h1>Test</h1>
```

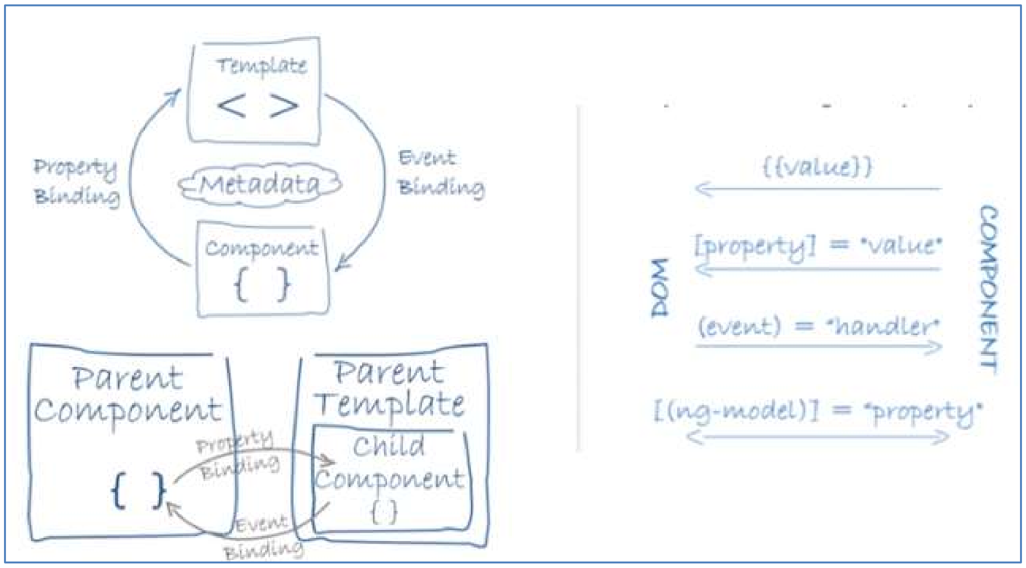
Also, use template in the my-comp…c.ts file and test output:

```
template: `<h1>Test</h1><p>my-component worked!</p>`,
```

Data Binding

It is a mechanism for co-ordinating parts of a template with parts of a component.





Change startup back to AppComponent

Open app.m.ts

Change the “bootstrap” value: bootstrap: [AppComponent]

Open index.html and change to use <app-root>

```
<app-root>Loading...</app-root>
```

In app.module.ts:

```
import { FormsModule } from '@angular/forms';
```

and

```
  imports: [
    BrowserModule,
    FormsModule
  ],
```

In app.c.ts:

```
import { FormsModule } from '@angular/forms';
```

In app.c.html, add the following after <h1>: (*3 ways to do binding. 2nd and 3rd are 2-way data-binding*)

```
<input [value]="myclass.name" type="text">
<input [value]="myclass.name" (input)="myclass.name = $event.target.value"
type="text">
<input [(ngModel)]="myclass.name" type="text">
```

This ties to the property in MyClass.

Browser test.

Angular input definition

User actions raise DOM events. Bind events to actions.

User input OnKey demo:

Open app.c.html and add these:

```
<br>
<input (keyup)="onKey($event)" type="text">
<p>{{ values }}
```

Open app.c.ts:

```
  title = 'My first App works!';
  values = '';
  onKey(event: KeyboardEvent) {
    this.values += (<HTMLInputElement>event.target).value;
  }
```

Better to use the [(ngModel)] technique for 2-way binding.

Forms

Co-ordinates a set of data-bound user controls, tracks changes, validates input and presents errors.

- Forms: Class creation

  ng g class Form-Class

```
export class FormImplement {
    constructor(public name: string, public age: number, public married?:
string) {

    }
}
```

- Forms: Forms module

  Import the FormsModule in the app.m.ts file

```
import { FormsModule } from '@angular/forms';
```

and

```
  imports: [
    BrowserModule,
    FormsModule
  ],
```

- Forms: Forms template

Open app.c.ts:

```
import {FormImplement} from './form-class';
```

Add a property:

```
model = new FormImplement("Ajay Singala", 26, "No");
```

Open app.c.html and add at end a form:

```
<form>
  <label for="name">Name: </label>
  <input type="text" [(ngModel)]="model.name" id="name" name="name" required>
  <label for="age">Age: </label>
  <input type="text" [(ngModel)]="model.age" id="age" name="age" required>
  <label for="married">Married: </label>
  <input type="text" [(ngModel)]="model.married" id="married" name="married">
</form>
```

- Forms: Forms Validation

Open app.c.html:

```
<form>
  <label for="name">Name: </label>
  <input type="text" [(ngModel)]="model.name" #name="ngModel" id="name"
name="name" required>
  <label for="age">Age: </label>
  <input type="text" [(ngModel)]="model.age" #age="ngModel" id="age"
name="age" required>
  <label for="married">Married: </label>
  <input type="text" [(ngModel)]="model.married" #married="ngModel"
id="married" name="married">

  <div *ngIf="!age.touched">
    Untouched
  </div>
</form>
```

Can use the following:

- touched, untouched: controls visited or not
- dirty, pristine: if the value has changed
- valid, invalid

age shows untouched
click in age text box and nagivate out. Untouched disappears
Add following and test:

```
  <div *ngIf="age.valid">
    Valid
  </div>
```

Change to:

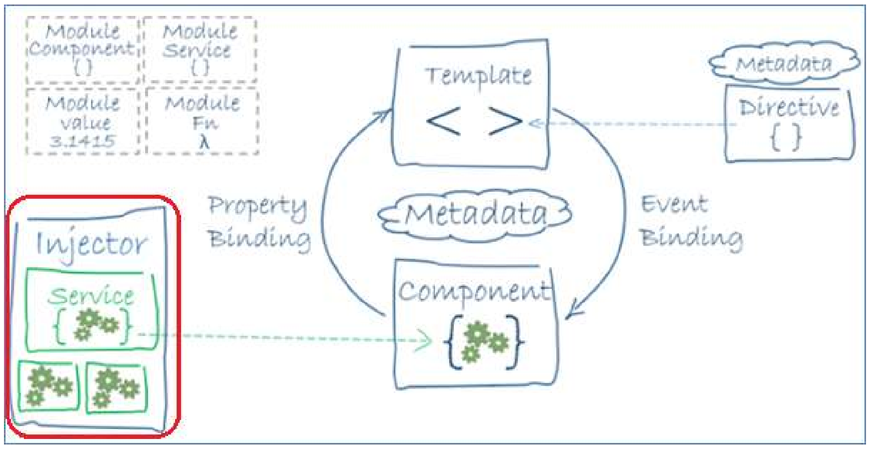
```
  <div *ngIf="!age.valid">
    Invalid
  </div>
```

Add following and test:

```
  <div *ngIf="age.dirty">
    Is Dirty
  </div>
```

DI

DI is a design pattern that passes an object as dependencies in different components across the app



Inject a service in to a component and use it.
To use a service as DI, make it injectable by importing the Injectable module and using the @Injectable decorator.
Create a new class: ng g class MainService

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

@Injectable()

export class MainService {
    getData() {
        return "My First DI!!";
    }
}
```

2 ways to use (inject) a service:

1. do it directly in the component and inject it as:

```
@Component {
   Providers: [service]
}
```

OR

2. do it in the app.module.ts, which is the preferred way, so it can be reused in any component and you don't have to include it in each component separately.

In app.m.ts:

```
import { MainService } from './main-service';
:
@NgModule({
  declarations: [
      :
  ],
  imports: [
      :
  ],
  providers: [MyClass, MainService],
  bootstrap: [AppComponent]
})
```

Then, import it in the component, app.c.ts:

```
import { MainService } from './main-service';
```

and inject it in the ctor:

```
export class AppComponent {
  constructor(private myclass: MyClass, private mainService: MainService) {
    console.log(mainService.getData());
  }
      :
}
```

Refresh browser, Displays in console (inspect element)

If you remove the MainService from the “providers” section in app.m.ts, it will show errors on the console window as DI error.
You can inject it in the component as:

```
  providers: [MainService],
```

and it will work. Not recommended. Apply in app.module.ts.

Services – Http Client

https://coursetro.com/posts/code/171/Angular-7-Tutorial---Learn-Angular-7-by-Example -> Angular 7 HTTP Client

~~Using http services for CRUD operations.~~
~~Create a json file to send/receive data, assets/http-service.json:~~

```
{
    "name":"John Smith",
    "age":"35",
    "occupation":"Programmer",
    "status":"Married",
    "salary":"75000"
}
```

~~Move the json file to the “assets” folder~~
~~Create a new class: ng g class HolderClass and define ctor with the mapped fields:~~

```
export class HolderClass {
    constructor(public name: string,
            public age: number,
            public occupation: string,
            public status: string,
            public income: string) {
    }
}
```

~~In app.module.ts:~~

```
import { HttpModule, Response } from '@angular/http';
```

~~and~~

```
  imports: [
    :
    , HttpModule
  ],
```

~~Then, In app.c.ts:~~
~~Import Http and Response modules:~~

```
import { Http, Response } from '@angular/http';
```

~~Also import our HolderClass:~~

```
import { HolderClass } from './holder-class';
```

~~To allow mapping the http service and subscribe it to the holder class, import:~~

```
import { Observable } from 'rxjs/Observable';
import 'rxjs/Rx';
```

~~Inject http in to the ctor:~~

```
~~  constructor(private myclass: MyClass, private mainService: MainService,~~
~~    private http: Http) {~~
~~    console.log(mainService.getData());~~
~~  }~~
```

~~then, in the ctor, add this statement:~~

```
~~    var data = http.get('assets/http-service.json').subscribe(res =>~~
~~console.log(res.json() as HolderClass));~~
```

~~define a var "data", use http to get the data from the json asset file, after getting it, subscribe to the response, and output to the console the json of the response and set that as the HolderClass.~~
~~It will map each of the properties in the json to the HolderClass.~~
~~Refresh browser, Displays json object in console (inspect element)~~

Promise vs Observable

https://angular.io/guide/observables
https://stackoverflow.com/questions/37364973/promise-vs-observable
https://dzone.com/articles/angular-observables-and-promises-how-to-use-them
http://chariotsolutions.com/blog/post/angular2-observables-http-separating-services-components/

A Promise handles a single event when an Async op completes or fails. Cannot be cancelled.
An Observable is like a stream and allows you to pass zero or more events where a callback is called for each individual event (which is cancellable).
P: work with a single value at a time
O: Work with multiple values at a time
Pick O over P, as some of the features of P are already embedded in O.
Demo to subscribe to a json result and map it to an array object of HolderClass
Convert the json file in to an array (using [and]):

```
[
    {
        "name": "John Smith",
        "age": "35",
        "occupation": "Programmer",
        "status": "Married",
        "salary": "75000"
    }
]
```

Open app.c.ts
Declare an array and add a map() method:

```
  //result:Array<HolderClass>;
  result: any;

  map() {
    this.http.get('assets/http-service.json')
      .subscribe(res => this.result = res);
  }
```

In app.c.html, just before </form>:

```
  <button (click)="map()">Map</button>
  <ul *ngFor="let item of result">
    <li>{{ item.name }}</li>
  </ul>
```

Refresh browser, click on Map. Displays one record. Add one more rec to the json and try again.
Add more fields to display in html and retry:

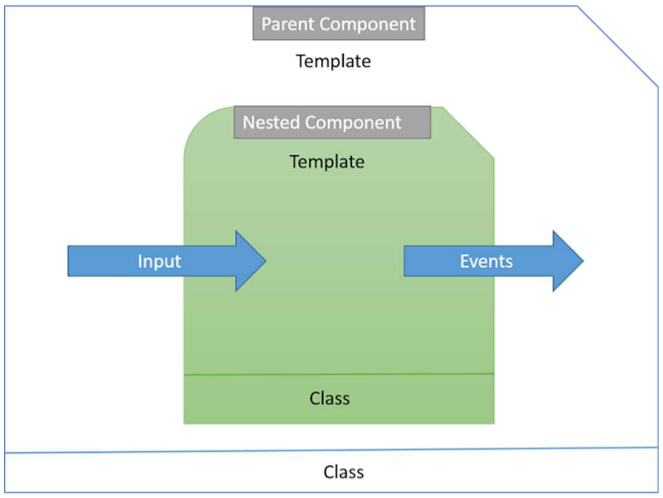
```
      <li>{{ item.name }}</li>
      <li>{{ item.age }}</li>
```

Nested / Child components

https://codecraft.tv/courses/angular/quickstart/nesting-components-and-inputs/

In Angular, a comp can have child comps and those child comps can have their own further child comps. Seamless support for nested comps.
Questions:

1. how to use nested comps in its parent comp?
2. how can we pass data from parent to its child comp?
3. how can we get the data back from child to parent comp?
4. how does child respond to parent events?
5. How does parent respond to child events?



Build Parent and Child comps:

Use the AngDemo project to show the “my friends” components.
Parent: myprofile (myfriends comp not required)
Child: myfriend

Directives

https://codecraft.tv/courses/angular/built-in-directives/overview/
https://codecraft.tv/courses/angular/built-in-directives/ngfor/
https://codecraft.tv/courses/angular/built-in-directives/ngif-and-ngswitch/

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

https://blog.angularindepth.com/angular-mastery-ngclass-ngstyle-e972dd580889

(for demo, use Directives Component in AngDemo project)
3 types:

1. Directives: components without a view / template
2. Components: directives with a view / template
3. Structural directives: change the DOM layout by adding and removing DOM elements
4. Attribute directives: change the appearance or behaviour of an element, component or other directive

Everything you can do with a directive you can also do with a component. But not everything you can do with a component you can do with a directive.

Components are the most common of the 4. Already seen in demos.
Structural Directives: change the structure of a view
For e,g,; ngFor and ngIf
Attribute Directives: used as attributes of elements
For e.g.; ngModel (already seen in demos)
Other example are ngStyle and ngClass directive can change several element styles at the same time

Custom Directives:

ng g directive Highlight
(demo available in AngDemo)

https://codecraft.tv/courses/angular/custom-directives/creating-a-custom-directive/
https://angular.io/guide/attribute-directives
https://alligator.io/angular/building-custom-directives-angular/

Pipes

https://codecraft.tv/courses/angular/pipes/built-in-pipes/

Some values benefit from a bit of editing. You may notice that you desire many of the same transformations repeatedly, both within and across many applications. You can almost think of them as styles. In fact, you might like to apply them in your HTML templates as you do styles.

Introducing Angular pipes, a way to write display-value transformations that you can declare in your HTML.

For example, in most use cases, users prefer to see a date in a simple format like April 15, 1988 rather than the raw string format Fri Apr 15 1988 00:00:00 GMT-0700 (Pacific Daylight Time)

A pipe takes in data as input and transforms it to a desired output.

Built-in Pipes

Demo: AngDemo/src/app/PipesDemo/Birthday component

Async Pipes

Async Promise Demo:
Demo: AngDemo/src/app/PipesDemo/Async-Pipe component

Async Observables Demo:
Demo: AngDemo/src/app/PipesDemo/async-observable-pipe component

Custom Pipes

2 demos:

- A “default” pipe (angdemo/pipedemo/default-pipe.ts)
- A “Exopnential Power Booster” pipe (angdemo/pipedemo/power-booster)

Lifecycle Hooks

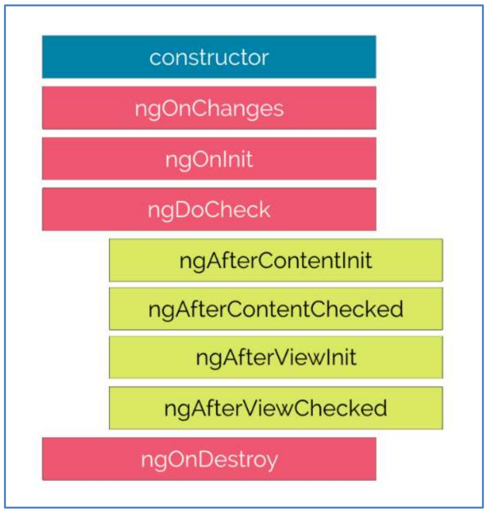
- OnInit, OnChanges, DoCheck etc.

Hooks for components

Hook	Purpose and Timing
constructor	This is invoked when Angular creates a component or directive by calling new on the class.
ngOnChanges()	Respond when Angular (re)sets data-bound input properties. The method receives a SimpleChanges object of current and previous property values. Called before ngOnInit() and whenever one or more data-bound input properties change.
ngOnInit()	Initialize the directive/component after Angular first displays the data-bound properties and sets the directive/component's input properties. Called once, after the first ngOnChanges().
ngDoCheck()	Detect and act upon changes that Angular can't or won't detect on its own. Called during every change detection run, immediately after ngOnChanges()and ngOnInit().
ngAfterContentInit()	Respond after Angular projects external content into the component's view / the view that a directive is in. Called once after the first ngDoCheck().
ngAfterContentChecked()	Respond after Angular checks the content projected into the directive/component. Called after the ngAfterContentInit() and every subsequent ngDoCheck().
ngAfterViewInit()	Respond after Angular initializes the component's views and child views / the view that a directive is in. Called once after the first ngAfterContentChecked().
ngAfterViewChecked()	Respond after Angular checks the component's views and child views / the view that a directive is in. Called after the ngAfterViewInit and every subsequent ngAfterContentChecked().
ngOnDestroy()	Cleanup just before Angular destroys the directive/component. Unsubscribe Observables and detach event handlers to avoid memory leaks. Called just before Angular destroys the directive/component.

`ngDoCheck` and `ngOnChanges` should not be implemented together on the same component.

Hooks are executed in this order:



Hooks for child components

These hooks are only called for components and not directives.

ngAfterContentInit
Invoked *after* Angular performs any content projection into the components view

ngAfterContentChecked
Invoked each time the content of the given component has been checked by the change detection mechanism of Angular.

ngAfterViewInit
Invoked when the component's view has been fully initialized.

ngAfterViewChecked
Invoked each time the view of the given component has been checked by the change detection mechanism of Angular.

Demo: AngDemo/src/app/Hooks

Communication Between Components

ViewChild, ViewChildren, ContentChild, ContentChildren

The view children of a given component are the elements used within its template, its view.

We can get a reference to these view children in our component class by using the `@ViewChild` decorator. https://www.techiediaries.com/angular-dom-queries-viewchild/

Demo at: ViewChildDemo\Jokes

Demo at: ngfirst/Child and ngfirst/app

@Input(), @Output() and EventEmitter

Demo at: ngfirst/Child, ngfirst/personParent and ngfirst/personChild

- https://levelup.gitconnected.com/angular-7-share-component-data-with-other-components-1b91d6f0b93f
- https://dzone.com/articles/understanding-output-and-eventemitter-in-angular
- https://www.codementor.io/yomateo/angular7-101-input-output-os4et83m5

Model / Template-Driven Forms

https://coursetro.com/posts/code/171/Angular-7-Tutorial---Learn-Angular-7-by-Example -> Angular 7 Forms

Template Driven Forms:

- https://jasonwatmore.com/post/2019/06/15/angular-8-template-driven-forms-validation-example
- https://codecraft.tv/courses/angular/forms/template-driven/

Whether we are template driven or model driven we need some basic form HTML to begin with.

Important: In model driven forms, contrary to what you might think, the HTML for our form isn't automatically created for us. We *still* need to write the HTML that represents our form and then explicitly link the HTML form elements to code on our component.

We added the novalidate attribute to the form element, by default browsers perform their own validation and show their own error popups. Since we want to handle the form validation ourselves we can switch off this behaviour by adding novalidate to the form element.

We are using the markup and styles from the twitter bootstrap UI framework to structure our form.

No Validations:

Demo: \ModelDrivenForms\ModelForms\model-form component

With Validations:

Demo: \ModelDrivenForms\ModelForms\model-form-v component

Advanced Routing

Using Angular Route Guard For securing routes:

- https://codeburst.io/using-angular-route-guard-for-securing-routes-eabf5b86b4d1
- https://medium.com/@ryanchenkie_40935/angular-authentication-using-route-guards-bf7a4ca13ae3

Lazy Loaded Module Example in Angular 8|7 with loadChildren & Dynamic Imports:

- https://www.techiediaries.com/angular-lazy-load-module-example/

Lazy Loading Routes in Angular:

- https://alligator.io/angular/lazy-loading/

Unit Testing

- https://codecraft.tv/courses/angular/unit-testing/jasmine-and-karma/
- https://dev.to/mustapha/angular-unit-testing-101-with-examples-6mc

Terminologies

Automated testing

It's the practice of writing code to test our code, and then run those tests. There are 3 types of tests: unit tests, integration tests, and end-to-end (e2e) tests.

Unit test

A unit test or UT is the procedure to check the proper functioning of a specific part of a software or a portion of a program.

Karma

Karma is a test runner. It will automatically create a browser instance, run our tests, then gives us the results. The big advantage is that it allows us to test our code in different browsers without any manual change in our part.

The pattern that Karma uses to identify test files is `<filename>.spec.ts`. This is a general convention that other languages use. If for some reason you want to change it, you can do so in the `test.ts` file.

- Karma is a tool which lets us spawn browsers and run jasmine tests inside of them all from the command line
- The results of the tests are also displayed on the command line
- Karma can also watch your development files for changes and re-run the tests automatically
- When creating Angular projects using the Angular CLI, it defaults to creating and running unit tests using Jasmine and Karma

- Whenever we create files using the CLI as well as creating the main code file it also creates simple jasmine spec file named the same as the main code file but ending in .spec.ts, like:

```
customer.component.ts
customer.component.spec.ts
```

Jasmine

Jasmine is a popular Javascript testing framework. It comes with test doubles by using spies (we'll define what is a spy later), and assertions built into it out of the box.

Jasmine provides a lot of useful functions to write tests. The three main APIs are:

1. `Describe()`: It's a suite of tests
2. `it()`: Declaration of a single test
3. `expect()`: Expect something to be true for example

Mock

Mock objects are *fake* (simulated) objects that mimic the behavior of real objects in controlled ways.

Fixture

A fixture is a wrapper around an instance of a component. With a fixture, we can have access to a component instance as well as its template.

Spy

Spies are useful for verifying the behavior of our components depending on outside inputs, without having to define those outside inputs. They're most useful when testing components that have services as a dependency.

Built-in Matchers

expect(array).toContain(member);	expect(mixed).toEqual(mixed);
expect(fn).toThrow(string);	expect(mixed).toMatch(pattern);
expect(fn).toThrowError(string);	expect(number).toBeCloseTo(number, decimalPlaces);
expect(instance).toBe(instance);	expect(number).toBeGreaterThan(number);
expect(mixed).toBeDefined();	expect(number).toBeLessThan(number);
expect(mixed).toBeFalsy();	expect(number).toBeNaN();
expect(mixed).toBeNull();	expect(spy).toHaveBeenCalled();
expect(mixed).toBeTruthy();	expect(spy).toHaveBeenCalledTimes(number);
expect(mixed).toBeTruthy();	expect(spy).toHaveBeenCalledWith(...arguments);
expect(mixed).toBeUndefined();	

Samples #1

```
import { CustomerComponent } from './customer.component';
import { CustomerModel } from '../customer-model';

describe('Hello there!', () => {
  it('greet hello', () => {
    let custComponent = new CustomerComponent();
    expect(custComponent.greet())
      .toEqual('Hello there!');
  })
});

describe('Get First Customer', () => {
  it('Get First Customer', () => {
    let custComponent = new CustomerComponent();
    custComponent.ngOnInit();
    expect(custComponent.customers[0].firstName)
      .toEqual('first');
  })
});

describe('Create a new Customer', () => {
  it('Create a new Customer', () => {
    let custComponent = new CustomerComponent();
    let aCustomer = new CustomerModel("Test Firstname", "Test Lastname");
    custComponent.create(aCustomer);
    let count = custComponent.customers.length;
    expect(custComponent.customers[count - 1].firstName)
      .toEqual('Test Firstname');
  })
});
```

Running the tests

1. Execute: `ng test`
   a. On the command prompt, in the same folder from where you run ***`ng serve`***, execute **`ng test`**
2. This will compile and run all tests in your Angular app and open the results in a browser
   a. May take some time the first time
   b. Keep the command window running

   c. Just like ng server, ng test will constantly monitor changes to your code and re-execute tests and refresh the browser with the new results
3. In the browser, click on “Spec List” to view collapsed list of the test results
   a. Displays all the tests
      i. Green means passed
      ii. Red means failed
   b. Click on a specific test to see details of that test
4. In the browser, click on “Failures” to view all failed tests with details
5. On the command window, <CTRL+C> to break or close browser and <CTRL+C>

Test Suites – Setup and Teardown

- Executing multiple tests as a ***test suite***
- Setup for each test and clean up at the end
  - **beforeAll**
    - Called once, before all the specs in describe test suite are run.
  - **afterAll**
    - Called once after all the specs in a test suite are finished.
  - **beforeEach**
    - Called before each test specification, it function, has been run.
  - **afterEach**
    - Called after each test specification has been run.

```
describe('Run Tests Together', () => {
  let custComponent: CustomerComponent;
  beforeEach(() => {
    custComponent = new CustomerComponent();
  });
  afterEach(() => {
    custComponent = null;
  });

  it('2gether: greet hello', () => {
    let custComponent = new CustomerComponent();
    expect(custComponent.greet())
      .toEqual('Hello there!');
  });

  it('2gether: Get First Customer', () => {
    let custComponent = new CustomerComponent();
    custComponent.ngOnInit();
    expect(custComponent.customers[0].firstName)
```

```
      .toEqual('first');
  });
});
```

Samples #2

Skeleton of a test

Using the three Jasmine APIs mentioned above, a skeleton of a unit test should look like this:

```
describe('TestSuitName', () => {
  // suite of tests here
  it('should do some stuff', () => {
    // this is the body of the test
  });
});
```

When testing, there's a pattern that became almost a standard across the developer community, called AAA (Arrange-Act-Assert). AAA suggests that you should divide your test method into three sections: arrange, act and assert. Each one of them only responsible for the part in which they are named after.

So the arrange section you only have code required to set up that specific test. Here objects would be created, mocks setup (if you are using one) and potentially expectations would be set. Then there is the Act, which should be the invocation of the method being tested. And on Assert you would simply check whether the expectations were met.

Following this pattern does make the code quite well structured and easy to understand. In general lines, it would look like this:

```
  it('should truncate a string if its too long (>20)', () => {
    // Arrange
    const pipe = new TroncaturePipe();

    // Act
    const ret = pipe.transform('1234567890123456789012345');
    // Assert
    expect(ret.length).toBeLessThanOrEqual(20);
  });
```

Configuration & instantiation

In order to access methods of the component we want to test, we first need to instantiate it. Jasmine comes with an API called `beforeAll()` which is called once before all the tests. The thing is if we instantiate our component inside this function our tests won't be isolated because the component properties could be changed by each test, and therefore, a first test could influence the behavior of a second test.
To solve that problem, Jasmine has another API called `beforeEach()`, which is very useful as it lets our tests to be run from the same starting point and thus to be run in isolation.
So, using this API, our test should look something like this:

```
describe('componentName', () => {
  // suite of tests here
  beforeEach(() => {
    TestBed.configureTestingModule({
      declarations: [myComponent],
    });
    fixture = TestBed.createComponent(myComponent);
    component = fixture.componentInstance;
  });
  it('should do some stuff', () => {
    // this is the body of the test
    // test stuff here
    expect(myComponent.methodOfMyComponent()).not.toBe(true);
  });
});
```

All of a sudden we have a lot of new unknown APIs. Let's have a closer look at what we have here.

Angular comes with an API for testing `testBed` that has a method `configureTestingModule()` for configuring a test module where we can import other Angular modules, components, pipes, directives, or services.

Once our testing module configured we can then instantiate for example the component we want to test.

Components

An Angular component combines an HTML template and a TypeScript class. So, to test a component we need to create the component's host element in the browser DOM.

To do that we use a `TestBed` method called `createComponent()`. This method will create a fixture containing our component instance and its HTML reference. With this fixture, we can access the raw component by calling its property `componentInstance` and its HTML reference by using `nativeElement`.

With that, an Angular component test should look like this:

```
describe('HeaderComponent', () => {
  let component: HeaderComponent;
  let element: HTMLElement;
  let fixture: ComponentFixture<HeaderComponent>;

  // * We use beforeEach so our tests are run in isolation
  beforeEach(() => {
    TestBed.configureTestingModule({
      // * here we configure our testing module with all the declarations,
      // * imports, and providers necessary to this component
      imports: [CommonModule],
      providers: [],
      declarations: [HeaderComponent],
    }).compileComponents();

    fixture = TestBed.createComponent(HeaderComponent);
    component = fixture.componentInstance; // The component instantiation
    element = fixture.nativeElement; // The HTML reference
  });
  it('should create', () => {
    expect(component).toBeTruthy();
  });

  it('should create', () => {
    // * arrange
    const title = 'Hey there, i hope you are enjoying this article';
    const titleElement = element.querySelector('.header-title');
    // * act
    component.title = title;
    fixture.detectChanges();
    // * assert
    expect(titleElement.textContent).toContain(title);
  });
});
```

After setting the title in our test, we need to call detectChanges() so the template is updated with the new title we just set (because binding happens when Angular performs change detection).

Pipes

Because a pipe is a class that has one method, transform, (that manipulates the input value into a transformed output value), it's easier to test without any Angular testing utilities.

Bellow an example of what a pipe test should look like:

```
describe('TroncaturePipe', () => {
  it('create an instance', () => {
    const pipe = new TroncaturePipe(); // * pipe instantiation
    expect(pipe).toBeTruthy();
  });

  it('truncate a string if its too long (>20)', () => {
    // * arrange
    const pipe = new TroncaturePipe();
    // * act
    const ret = pipe.transform('1234567891234567894566661 23');
    // * asser
    expect(ret.length).toBe(20);
  });
});
```

Directives

An attribute directive modifies the behavior of an element. So you could unit test it like a pipe where you only test its methods, or you could test it with a host component where you can check if it correctly changed its behavior.

Here is an example of testing a directive with a host component:

```
// * Host component:
@Component({
  template: `<div [appPadding]="2">Test</div>`,
})
```

```
class HostComponent {}
@NgModule({
  declarations: [HostComponent, PaddingDirective],
  exports: [HostComponent],
})
class HostModule {}

// * Test suite:
describe('PaddingDirective', () => {
  let component: HostComponent;
  let element: HTMLElement;
  let fixture: ComponentFixture<HostComponent>;

  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [CommonModule, HostModule], // * we import the host module
    }).compileComponents();

    fixture = TestBed.createComponent(HostComponent);
    component = fixture.componentInstance;
    element = fixture.nativeElement;

    fixture.detectChanges(); // * so the directive gets appilied
  });

  it('should create a host instance', () => {
    expect(component).toBeTruthy();
  });

  it('should add padding', () => {
    // * arrange
    const el = element.querySelector('div');
    // * assert
    expect(el.style.padding).toBe('2rem'); // * we check if the directive
worked correctly
  });
});
```

Services

Like pipes, services are often easier to test. We could instantiate them with the `new` keyword. That's fine for basic services, but if your service has dependencies, it's better to use the `TestBed.configureTestingModule` API like this:

```
describe('LocalService', () => {
  let service: LocalService;

  beforeEach(() => {
    TestBed.configureTestingModule({
      providers: [LocalService],
    });

    service = TestBed.get(LocalService); // * inject service instance
  });

  it('should be created', () => {
    expect(service).toBeTruthy();
  });
  it('should set the local', () => {
    // * act
    service.setLocal('fr');
    // * assert
    expect(service.getLocal()).toBe('fr');
  });
});
```

To get the service instance, we could inject it inside a test by calling `TestBed.get()` (with the service class as the argument).

Well, with that you should be ready to write tests for your Angular applications. That being said, to tackle some of the common difficulties you might encounter while writing Angular tests, I added small cheatsheets you can find in the next section :)

Cheatsheets

Dealing with HTTP requests

To avoid making HTTP requests at each test, one method consists of providing a *fake* service that mocks the real one (the one that communicates via HTTP requests).
Once the fake service is implemented we provide it to the `TestBed.configureTestingModule()` like this:

```
class FakeApiService {
  // Implement the methods you want to overload here
  getData() {
    return of({ items: [] }); // * mocks the return of the real method
  }
}
//...
TestBed.configureTestingModule({
  imports: [],
  declarations: [myComponent],
  providers: [
    {
      provide: RealApiService,
      useClass: FakeApiService,
    },
  ],
});
//...
```

Dealing with the Angular router

To deal with the Router you could either add the `RouterTestingModule` in the imports of your testing module or you could mock it using the technique we saw in the test above.

Using spies

Spies are an easy way to check if a function was called or to provide a custom return value.
Here is an example of how to use them:

```
it('should do something', () => {
  // arrange
  const service = TestBed.get(dataService);
  const spyOnMethod = spyOn(service, 'saveData').and.callThrough();
  // act
  component.onSave();
  // assert
  expect(spyOnMethod).toHaveBeenCalled();
});
```

Dealing with asynchronous code

Dealing with promises

```
it('should do something async', async () => {
  //  * arrange
  const ob = { id: 1 };
  component.selected = ob;
  //  * act
  const selected = await component.getSelectedAsync(); // get the promise
value
  //  * assert
  expect(selected.id).toBe(ob.id);
});
```

Dealing with observables

```
it('should do something async', (done) => {
  //  * arrange
  const ob = { id: 1 };
  component.selected = ob;
  //  * act
  const selected$ = component.getSelectedObs(); // get an Observable
  //  * assert
  selected$.subscribe(selected => {
    expect(selected.id).toBe(ob.id);
    done(); // let Jasmine know that you are done testing
  });
});
```

Dealing with timeouts

```
const TIMEOUT_DELAY = 250;
//...
it('should do something async', (done) => {
  //  * arrange
  const ob = { id: 1 };
  //  * act
  component.setSelectedAfterATimeout(ob);
  // * assert
  setTimeout(() => {
    expect(component.selected.id).toBe(ob.id);
    done(); // let Jasmine know that you are done testing
  }, TIMEOUT_DELAY);
});
```

Angular JWT Authentication Example & Tutorial

- Url: https://dzone.com/articles/authenticate-your-angular-app-with-jwt
- Repo: https://github.com/AjaySingala/ngAuthWithNodeJSAndJWT.git

Other refs:

- https://jasonwatmore.com/post/2018/11/16/angular-7-jwt-authentication-example-tutorial

- https://www.codeproject.com/Articles/2259378/Using-Azure-AD-for-login-to-an-Angular-application
- https://medium.com/@sambowenhughes/configuring-your-angular-6-application-to-use-microsoft-b2c-authentication-99a9ff1403b3
- https://medium.com/@ryanchenkie_40935/angular-authentication-using-the-http-client-and-http-interceptors-2f9d1540eb8
- https://www.digital-moves.eu/2018/07/19/authentication-with-azure-ad-angular-6-client-web-api/
- https://stackoverflow.com/questions/51787312/passing-token-through-angular
- https://www.npmjs.com/package/microsoft-adal-angular6
- https://www.briankeating.net/post/Azure-AD-Angular7-net-Core-22-ADAL
- https://github.com/Azure-Samples/active-directory-angularjs-singlepageapp-dotnet-webapi
- https://github.com/Azure-Samples/active-directory-angularjs-singlepageapp