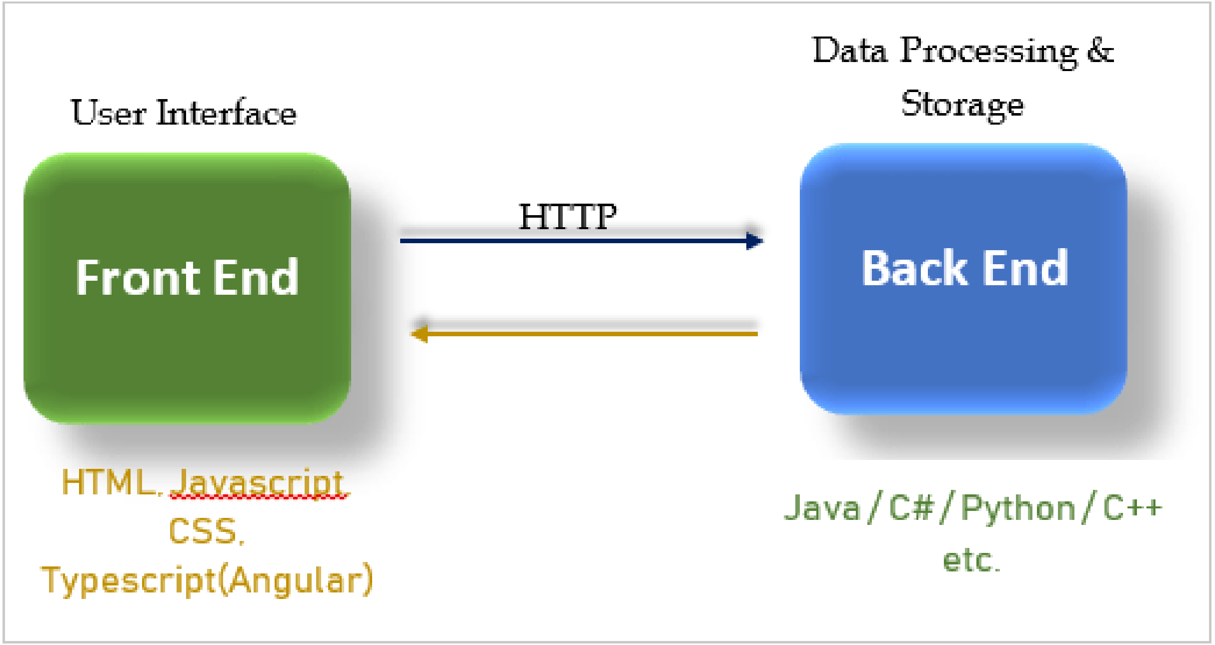
Overview

- Front end is what the user sees and interacts with backend.
- Front end runs in a web browser.
- Backend is concerned with data processing and storage.
- Backend runs over a server.


User Interface
Data Processing & Storage
Front End
HTTP
Back End
HTML, Javascript, CSS, Typescript(Angular)
Java / C# / Python / C++ etc.

What is a Front-end and what do we use it for

- It's all about presentation layer.
- A frontend which is often called a **client**, is a code which is running in a web browser.
- It's responsible for the presentation part (User Interface). Within Angular we are using HTML, CSS and Typescript to build the front-end of our application.
- The frontend is responsible for displaying data and presentation logic. This logic decides what happens when the user clicks on a button or picture or what to do when the user navigates to a different page.

What is a Backend and what do we use it for

- The backend usually consists of three parts:
  - Application (code)
  - Database(s)
  - Server (on which Application is running)

Introduction

- Angular is a framework for building client applications in HTML,CSS and JavaScript/TypeScript.
- **Why do we need Angular**

1. We can develop application using plain JS or libraries like jQuery.
2. If we develop a more complex application it may have multiple modules and thousands line of code.
3. Application **gets more complex**,
4. JavaScirpt code is getting **harder to maintain**
5. Projects get bigger their testing becomes crucial for stability

Benefits of Angular

- Increases code reusability
- Provides structure to JavaScript
- Makes an application more testable
- Gives our application a clean structure
- Modular development
- Two-way data-binding
- Great for **SPA** (Single-Page Applications)

Angular Features

1 . Cross Platform:

Angular is a cross platform language.

It supports multiple platforms.

we can develop different types of apps by using Angular.

- **Progressive web apps**

Progressive web applications are the most common apps which are built with Angular. Angular provides modern web platform capabilities to deliver high performance, offline, and zero-step installation apps.

- **Desktop**

Angular facilitates you to create desktop installed apps on different types of operating systems i.e. Windows, Mac or Linux by using the same Angular methods which we use for creating web and native apps.

Angular Features

2.Speed and Performance

- **Code generation**

  Angular makes your templates in highly optimized code for JavaScript virtual machines, it giving you all the benefits of hand-written code with the productivity of a framework.

- **Code splitting**

  Angular apps load quickly with the new Component Router, which delivers automatic code-splitting, so users only load code required to render the view they request.

3.Productivity

- **Templates**

  Quickly create UI views with simple and powerful template syntax.

Angular Features

- **Angular CLI**

  Command line tools: You can easily and quickly start building components, adding components, testing them, and then, instantly deploy them using Angular CLI.

- **IDEs**

  Get intelligent code completion, instant errors, and other feedback in popular editors and IDEs like Microsoft's VS Code.

3. Full Development

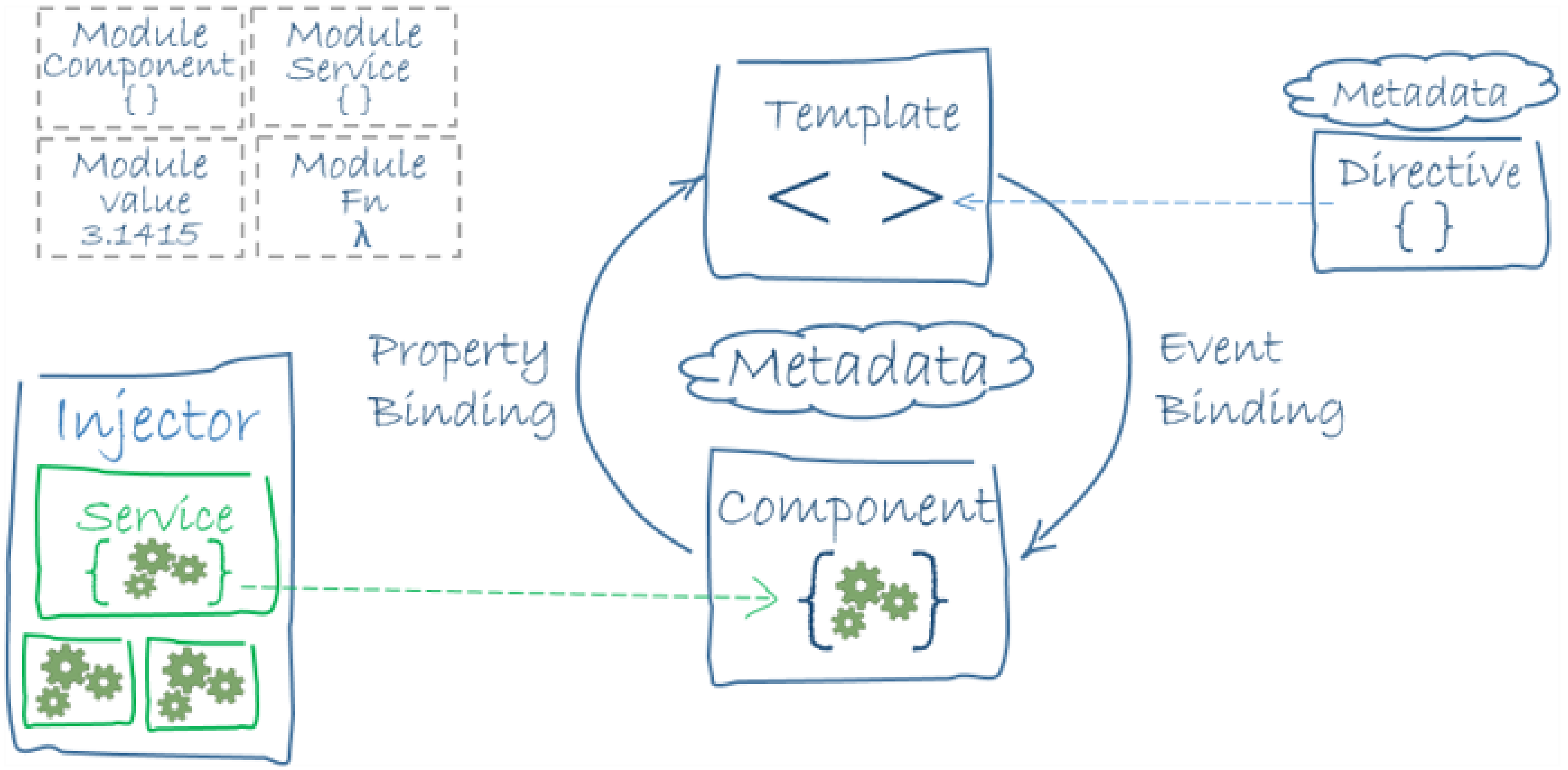
- **Testing**

  With Karma for unit tests, you can identify your mistake on the fly and Protractor makes your scenario tests run faster and in a stable manner.

Single-page application

- The most difference between a regular website and SPA is the reduced amount of page refreshes.
- SPAs have a heavier usage of AJAX- a way to communicate with back-end servers without doing a full page refresh to get data loaded into our application.
- The process of rendering pages happens mostly on the client-side.
- **Example**
- if you go through Gmail, you will notice that while opening mail from the inbox will only fetch the email and display it in place of the e-mail list. The rest of the components like sidebar, navigation bar etc. are not reloaded. It only refreshes the DOM (Document Object Model) for the required section. So, this reduces the overhead loading of the website.

Angular Architecture



Angular environment installation

- Installing Node.js and Node Package Manager (npm)
  - Download the Windows installer from the Nodes.js® web site.
  - Run the installer (the .msi file you downloaded in the previous step.)
  - Follow the prompts in the installer (Accept the license agreement, click the NEXT button a bunch of times and accept the default installation settings).
  - Restart your computer. You won't be able to run Node.js® until you restart your computer.

Angular environment installation

- Verifying installation

  how check that we have Node and NPM installed correctly

1. Open command line and type

```
node –v
npm  -v
```

2. Verify JS: Create a file called hello.js and type inside

```
console.log("Welcome to Angular");
```

3. Open terminal, navigate to folder where you have created a file

```
node hello.js
```

output

```
Welcome to Angular
```

Angular CLI installation and testing

- Angular CLI

  Angular Command Line Interface (CLI) is a tool which helps you to build angular apps.

- **Installation**

  Open terminal and run:

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

- **Verify installation**

  ```
  ng –v
  ```

- **Install Specific Version (Example: 6.1.1)**

  ```
  npm install -g @angular/cli@6.1.1
  ```

First Angular APP

- In this section, we're going to use the Angular command line interface (CLI) to generate a new Angular project.
- Syntax:

```
ng new new-app
```

- Next, point your command line to the project's root folder by running the following:

```
cd news-app
```

- **Installing Dependencies**

  To set up our dependencies, we're going to install, with just one command, all of the dependencies necessary for this tutorial.

  npm install --save @angular/material @angular/animations @angular/cdk

- **@ANGULAR/MATERIAL**

  This is the official material design package for the Angular framework.

Continue

- ng serve
- http://localhost:4200/ in a browser
- Basic Syntax :

Component	ng g component my-new-component
Directive	ng g directive my-new-directive
Pipe	ng g pipe my-new-pipe
Service	ng g service my-new-service
Class	ng g class my-new-class
Guard	ng g guard my-new-guard
Interface	ng g interface my-new-interface
Enum	ng g enum my-new-enum
Module	ng g module my-module

Visual studio Code-Project

- In order to open a project using VSC:
  - Open VSC
  - File > Open Folder
  - Navigate to project root folder
  - Click on Select Folder
- Start project
- In order to start a project we will use Node Package Manager (npm). Make sure you have npm installed in order to verify that you can open a default terminal/command line:
- Navigate to the project location using the command cd (the location on your computer will be different):
- Install dependencies using the npm install command
- Start the project using the npm start command
- Open a web browser and navigate to http://localhost:4200

VSC-Project

Add new message Hello World

- Open the src/app/app.component.html file.
- Replace the content of the file with Hello World.
- Check the result in a browser:

Project structure

1.The node_modules directory is where all of the libraries we need to build Angular are stored.
2.In the src directory, is placed the source code of your application.
3.The app/ folder contains file related to our initial App component:
4.The app.component.html contains template of the component
5.The app.component.scss contains the CSS styles for the component
6.The app.component.spec.ts contains the tests of our component
7.The app.component.ts contains a TypeScript code (logic) of the component
8.The app.module.ts contains the configuration of the App module
9.In the assets directory, you place any static assets like images or icons.
10.index.html is the main file of our app.
If we peek inside our index.html file, we can see there’s a element <app-root></app-root>, which will generate (render) our App component in the following place.
11. The styles.scss contains the global styles for our application

Project Struture

- 12.The angular.json file contains the configuration of our app.

What is TypeScript?

TypeScript is a superset of JavaScript.

TypeScript extends JavaScript with extra functionality which is not present in the current version of JavaScript supported by most browsers.

VSC

- Now you understand that TypeScript is independent of Angular, however, Angular cannot work without TypeScript. Therefore whenever you are running ng serve, or ng build, Angular CLI is calling the TypeScript compiler under the hood, so to compile the code of our application to JavaScript that can be used by browsers.

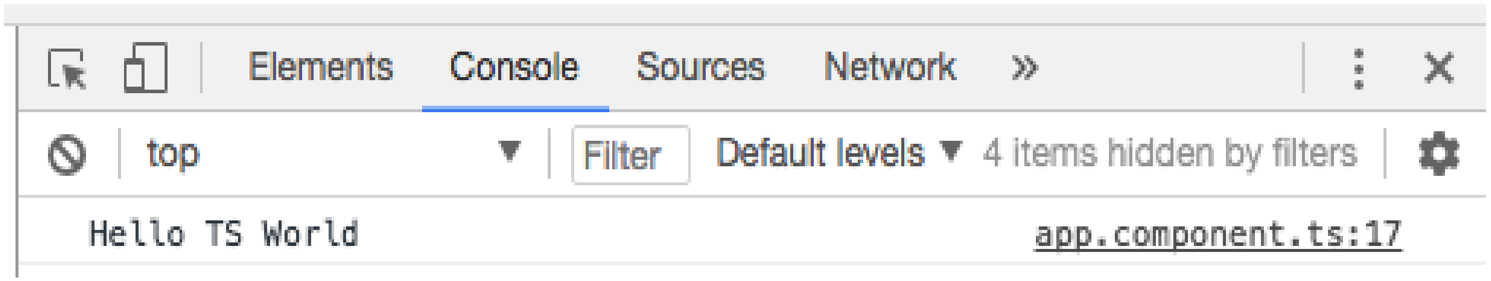
- While working with Angular you will hardly use the tsc command and stand-alone files, therefore we will switch to Visual Studio Code now.
  - Open Visual Studio Code
  - Open our recent project /my-app
  - Replace the content of /src/app/app.component.ts file

VSC

```
import { Component } from '@angular/core';
 @Component({
  selector: 'app-root',
  templateUrl: 'app.component.html',
  styleUrls: ['app.component.css']
 })
 export class AppComponent {
  constructor(){
   this.onStartup();
  }
 onStartup(){
  console.log('Hello TS World');
 }
}
```

VSC

- Save the file, then run ng serve . Open the developer console. You will see:
- As you noticed we have created a function — onStartup() — and added it to the constructor of our AppComponent. Whenever an instance of our component is created, it will call our onStartup function.



```
Hello TS World                                   app.component.ts:17
```

Typescript fundamentals

- In order to crate Angular apps, you have to know Typescript
  - Type annotations
  - Arrow functions
  - Interfaces
  - Classes
  - Constructors
  - Access modifiers
  - Properties
  - Modules

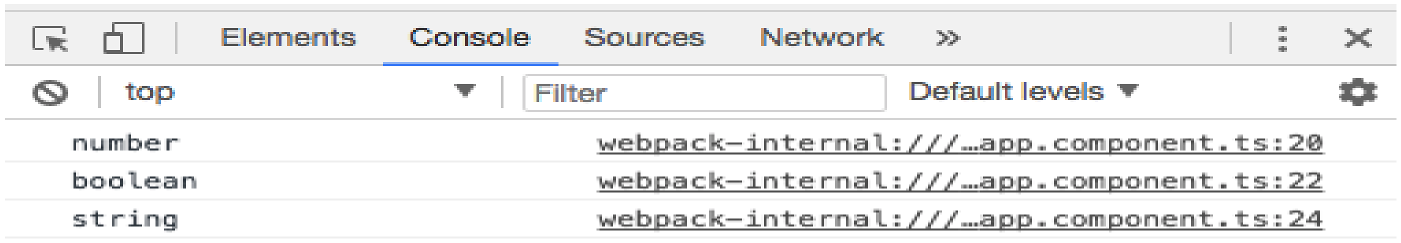
Typescript variable types

- number,
- string,
- boolean
- any

- let a: number;   //numeric type
- let b: string;   //string
- let c: boolean;   //true or false
- let d: any;   //any (dynamic)
- let e: number[] = [1,2,3];   //array of numbers
- let f: string[] = ['a','b','c']; //array of strings
- let g: any[] = [true, 1, 'a'];   //array of any

Type: any

```
let myVar:any;
myVar = 1;     //number
console.log(typeof(myVar));
myVar = true;  //boolean
console.log(typeof(myVar));
myVar = 'mdb'; //string
console.log(typeof(myVar));
```



```
number                       webpack-internal:///…app.component.ts:20
boolean                      webpack-internal:///…app.component.ts:22
string                       webpack-internal:///…app.component.ts:24
```

Typescript variable scope

we will cover a different type of variables and scopes.

Replace the content of the onStartup() function:

```
onStartup(){
 function count(){
   for (var i = 0; i < 9; i++){
     console.log(i);
   }
   console.log('Counted: ' , i);
 }
 count();}
}
```

Typescript variable scope

let Variables Scope :

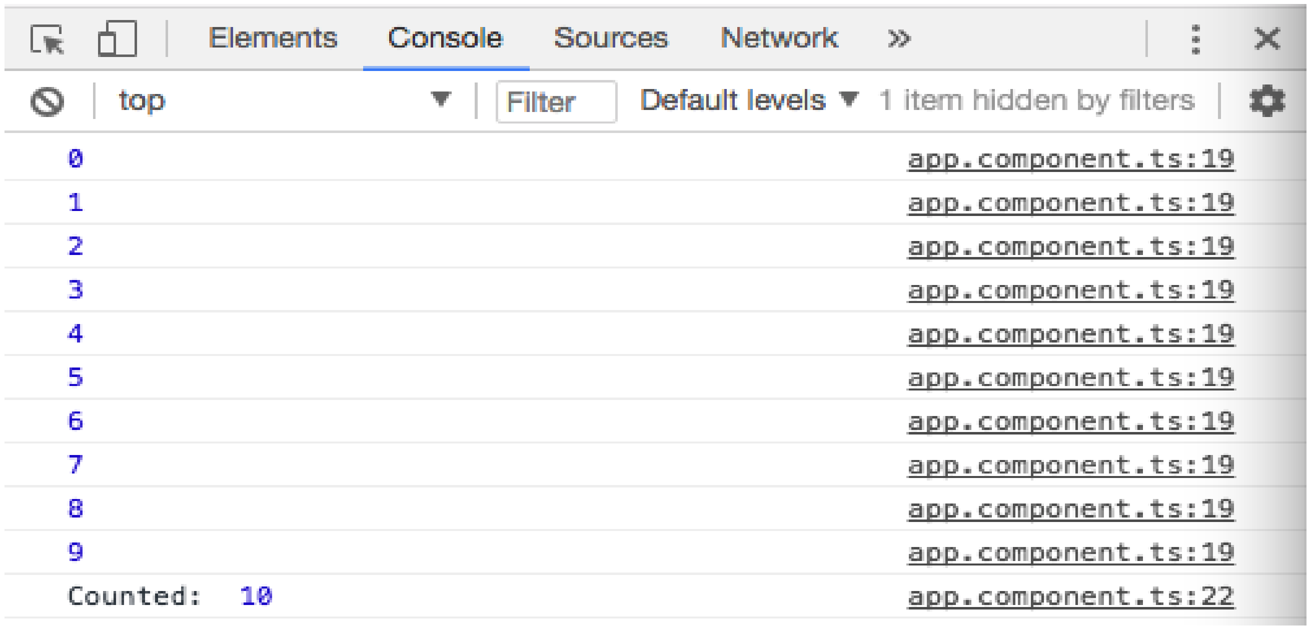
```
let num1:number = 1;
function letDeclaration() {
    let num2:number = 2;
    if (num2 > num1) {
        let num3: number = 3;
        num3++;  }
    while(num1 < num2) {
        let num4: number = 4;
        num1++;    }
    console.log(num1); //OK
    console.log(num2); //OK
    console.log(num3); //Compiler Error: Cannot find name 'num3'
    console.log(num4); //Compiler Error: Cannot find name 'num4'
}
```

Typescript variable scope

var Variables Scope:

```
var num1:number = 1;
function varDeclaration() {
   var num2:number = 2;
   if (num2 > num1) {
      var num3: number = 3;
      num3++; }
   while(num1 < num2) {
      var num4: number = 4;
      num1++; }
   console.log(num1); //2
   console.log(num2); //2
   console.log(num3); //4
   console.log(num4); //4
}varDeclaration();
```

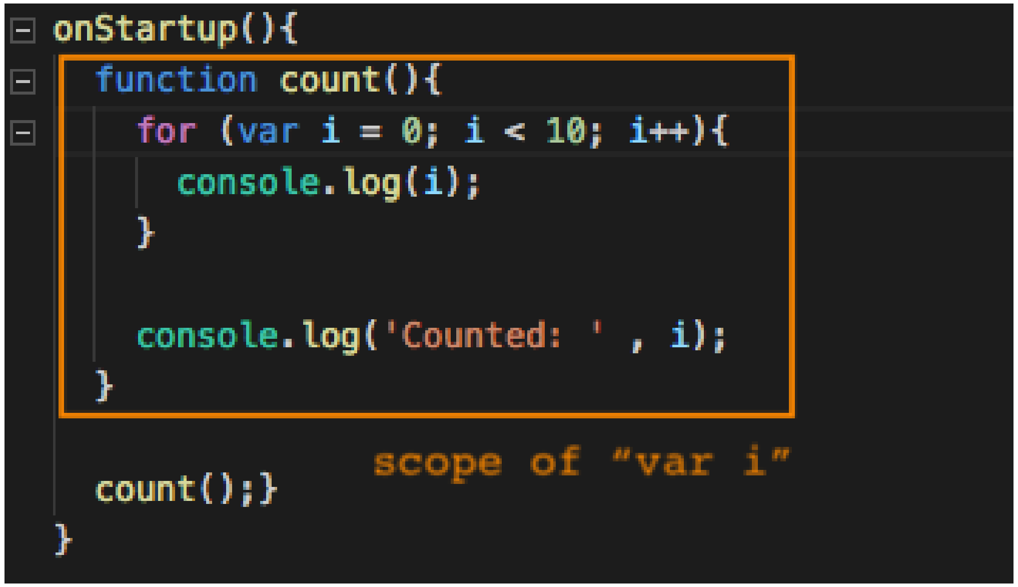
And start our application, what you will see?



```
0                                   app.component.ts:19
1                                   app.component.ts:19
2                                   app.component.ts:19
3                                   app.component.ts:19
4                                   app.component.ts:19
5                                   app.component.ts:19
6                                   app.component.ts:19
7                                   app.component.ts:19
8                                   app.component.ts:19
9                                   app.component.ts:19
Counted:  10                        app.component.ts:22
```

Scope :Var type

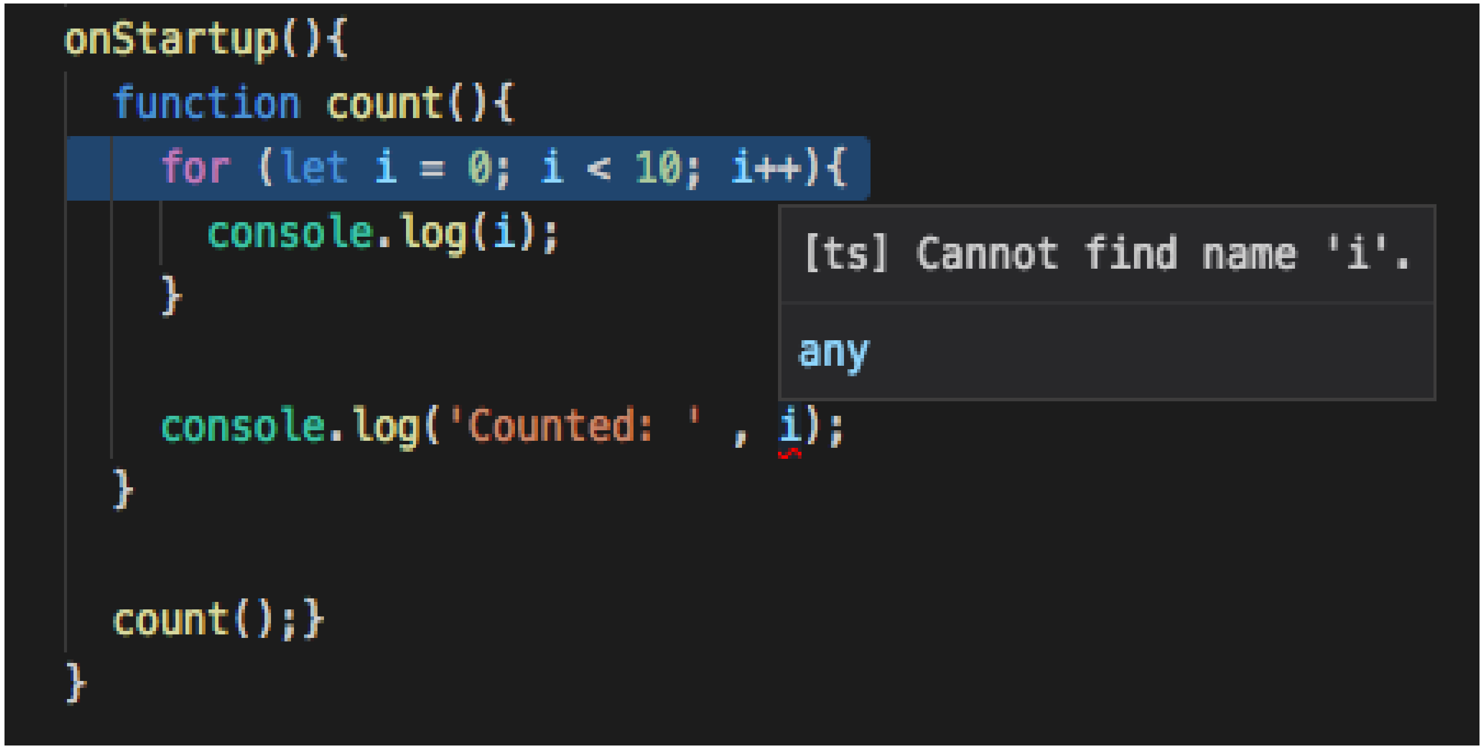
- JavaScript, all variables declared like that have scope to the nearest function.



```
onStartup(){
  function count(){
    for (var i = 0; i < 10; i++){
      console.log(i);
    }

    console.log('Counted: ' , i);
  }

  count();}
}
```

Scope : let type

- ERROR in src/app/app.component.ts(22,31): error : Cannot find name 'i'.



```
onStartup(){
  function count(){
    for (let i = 0; i < 10; i++){
      console.log(i);
    }

    console.log('Counted: ' , i);
  }

  count();}
}
```

```
[ts] Cannot find name 'i'.
any
```

Default methods for variables in typescript

- Declare the following variables in app.components.ts:

- let a: number;   //numeric type
- let b: string;   //string
- let c: boolean;   //true or false
- let d: any;   //any (dynamic)
- let e: number[] = [1,2,3];   //array of numbers
- let f: string[] = ['a','b','c']; //array of strings
- let g: any[] = [true, 1, 'a'];   //array of any

Numeric type methods:

type each of the variables, followed by dot (ie. a.) and observe the result:

Method	Description	Result
a = 5.56789;	Basic assignment	5.5679
a.toExponential()	Convert a number into an exponential notation:	#######
a.toFixed()	Converts a number to a string keeping a specified number of decimals.	6
a.toFixed(2)		5.57
a.toLocaleString('ar-EG')	Returns a string with a language sensitive representation of this number.	٥٫٥٦٨
a.toPrecision(2)	Returns a string representing the Number object to a specified precision.	5.6
	While toFixed(n) provides n length after the decimal point; toPrecision(x) provides x total length.	
a.toString()	Returns a string representing the Number object.	5.5679
a.valueOf()	The valueOf() method returns the wrapped primitive value of a Number object.	5.5679

String type methods:

Method	Description	Result
b = 'Hello World';	Basic assignment	Hello World
b.charAt(6)	The String object's charAt() method returns a new string consisting of the single UTF-16 code unit located at the specified offset into the string.	M
b.charCodeAt(6)	The charCodeAt() method returns an integer between 0 and 65535 representing the UTF-16 code unit at the given index.	77
b.concat('. It is nice to meet you.');	The concat() method combines the text of one or more strings and returns a new string.	Hello MDB Wolrd. It is nice to meet you.
b.endsWith('d')	The endsWith() method determines whether a string ends with the characters of a specified string, returning true or false as appropriate.	TRUE
b.includes('B');	The includes() method determines whether one string may be found within another string, returning true or false as appropriate.	TRUE
b.includes('b');		FALSE
a.indexOf('M')	The indexOf() method returns the index within the calling String object of the first occurrence of the specified value, starting the search at fromIndex. Returns -1 if the value is not found.	6
a.lastIndexOf('l')	The lastIndexOf() method returns the index within the calling String object of the last occurrence of the specified value, searching backwards from fromIndex. Returns -1 if the value is not found.	12

Any and boolean

- The any type doesn't offer any predefined methods. Boolean offers two basic methods — toString which returns a string of either 'true' or 'false, as well as valueOf which returns the primitive value of the Boolean object.

If statement

```
let x: number = 10, y = 20;


if (x > y)
{
    console.log('x is greater than y.');
}
else if (x < y)
{
    console.log('x is less than y.'); //This will be executed
}
else if (x == y)
{
    console.log('x is equal to y');
}
```

Ternary operator

A ternary operator is denoted by '?' and is used as a short cut for an if..else statement.

Syntax:

Boolean expression? First statement : second statement

For loop

There are three types of for loops.

- for loop
- for..of loop
- for..in loop
- for Loop
- The for loop is used to execute a block of code a given number of times, which is specified by a condition.

Syntax:

```
for (first expression; second expression; third expression ) {
   // statements to be executed repeatedly
}
```

For loop example

```
for (let i = 0; i < 3; i++) {
  console.log ("Block statement execution no." + i);
}
```

```
Block statement execution no.0
Block statement execution no.1
Block statement execution no.2
```

for...of Loop

TypeScript includes the **for...of** loop to iterate and access elements of an array, list, or tuple collection. The for...of loop returns elements from a collection e.g. array, list or tuple, and so, there is no need to use the traditional for loop shown above.

Example :

```
let arr = [10, 20, 30, 40];

for (var val of arr) {
  console.log(val); // prints values: 10, 20, 30, 40
}
```

for...in Loop

Another form of the for loop is for...in. This can be used with an array, list, or tuple. The for...in loop iterates through a list or collection and returns an index on each iteration.

```
let arr = [10, 20, 30, 40];

for (var index in arr) {
  console.log(index); // prints indexes: 0, 1, 2, 3

  console.log(arr[index]); // prints elements: 10, 20, 30, 40
}
```

Fundamentals: Type Annotations

- JavaScript is not a typed language. It means we cannot specify the type of a variable such as number, string, boolean etc. However, TypeScript is a typed language, where we can specify the type of the variables, function parameters and object properties.

- Demo:
- var message : string = "hello world"; //string
- var NumberVar: number = 100; //number
- var BooleanVar : boolean = true; //boolean
- var ArrayVar: string[] //arrays
- console.log("message ::"+message);
- console.log("number ::"+NumberVar);
- console.log("BooleanVar ::"+BooleanVar);

Fundamentals: Type Annotations

- Type Annotation in Object :

```
var employee : {
   id: number;
   name: string;
};

employee = {
  id: 100,
  name : "John"
}
```

Fundamentals:Interface

- Interface is a structure that defines the contract in your application. It defines the syntax for classes to follow. Classes that are derived from an interface must follow the structure provided by their interface.

```
interface IEmployee {
    empCode: number;
    empName: string;
    getSalary: (number) => number; // arrow function
    getManagerName(number): string;
}
```

Fundamentals:Interface

- Interface as Type :

```
interface KeyPair {
    key: number;
    value: string;
}

let kv1: KeyPair = { key:1, value:"Steve" }; // OK
let kv2: KeyPair = { key:1, val:"Steve" }; // Compiler Error: 'val' doesn't exist in
type 'KeyPair'

let kv3: KeyPair = { key:1, value:100 }; // Compiler Error:
```

Fundamentals:Interface

- **Interface as Function Type :**

```
interface KeyValueProcessor
{
    (key: number, value: string): void;
};
function addKeyValue(key:number, value:string):void {
    console.log('addKeyValue: key = ' + key + ', value = ' + value)
}
function updateKeyValue(key: number, value:string):void {
    console.log('updateKeyValue: key = '+ key + ', value = ' + value)
}
let kvp: KeyValueProcessor = addKeyValue;
kvp(1, 'Bill'); //Output: addKeyValue: key = 1, value = Bill
kvp = updateKeyValue;
kvp(2, 'Steve'); //Output: updateKeyValue: key = 2, value = Steve
```

Fundamentals:Interface

- **Type of Array :**

```
interface NumList {
   [index:number]:number
}

let numArr:  NumList = [1, 2, 3];
numArr[0];
numArr[1];

interface IStringList {
   [index:string]:string
}

let strArr : IStringList;
strArr["TS"] = "TypeScript";
strArr["JS"] = "JavaScript";
```

Fundamentals:Class

- object-oriented programming languages like Java and C#, classes are the fundamental entities used to create reusable components

- A class can include the following:

  Constructor

  Properties

  Methods

Fundamentals:Class

```
class Employee {
    empCode: number;
    empName: string;

    constructor(code: number, name: string) {
            this.empName = name;
            this.empCode = code;
    }

    getSalary() : number {
        return 10000;
    }
}
```

Fundamentals:Constructor

The constructor is a special type of method which is called when creating an object.

```
class Employee {

    empCode: number;
    empName: string;

    constructor(empcode: number, name: string ) {
        this.empCode = empcode;
        this.name = name;
    }
}
```

Creating an Object of class

```
class Employee {
   empCode: number;
   empName: string;
}

let emp = new Employee();
```

Fundamentals:Function

Functions are the primary blocks of any program.

TypeScript, functions can be of two types: named and anonymous.

Named Functions :

```
function display() {
    console.log("Hello TypeScript!");
}
display(); //Output: Hello TypeScript
function Sum(x: number, y: number) : number {
    return x + y;
}

Sum(2,3); // returns 5
```

Fundamentals:Function

- **Anonymous Function:**
- An anonymous function is one which is defined as an expression.
- This expression is stored in a variable.
- function itself does not have a name.
- These functions are invoked using the variable name that the function is stored in.

Fundamentals:Function

- **Anonymous Function Example :**

```
let greeting = function() {
    console.log("Hello TypeScript!");
};

greeting(); //Output: Hello TypeScript!
let Sum = function(x: number, y: number) : number
{
    return x + y;
}

Sum(2,3); // returns 5
```

Fundamentals:Function

- Arrow Function
- arrow notations are used for anonymous functions i.e for function expressions. They are also called lambda functions in other languages.

- Syntax:

```
(param1, param2, ..., paramN) => expression
```

Example:

```
        let sum = (x: number, y: number): number => {
        return x + y;
        }
        sum(10, 20); //returns 30
```

Fundamentals:Function

Arrow Function Example :

```
let Print = () => console.log("Hello TypeScript");

Print(); //Output: Hello TypeScript
```

Fundamentals:Function

- **With class**

```
class Employee {
   empCode: number;
   empName: string;

   constructor(code: number, name: string) {
          this.empName = name;
          this.empCode = code;
   }

   display = () => console.log(this.empCode +' ' + this.empName)
}
let emp = new Employee(1, 'Ram');
emp.display();
```

Fundamentals:Data Modifiers

- In object-oriented programming, the concept of 'Encapsulation' is used to make class members public or private i.e. a class can control the visibility of its data members. This is done using access modifiers.
- types of access modifiers in TypeScript: public, private and protected.
- **Public :**
- By default, all members of a class in TypeScript are public. All the public members can be accessed anywhere without any restrictions.

Fundamentals:Data Modifiers

- **Public access modifiers example :**

```
class Employee {
    public empCode: string;
    empName: string;
}

let emp = new Employee();
emp.empCode = 123;
emp.empName = “ram";
```

Fundamentals:Data Modifiers

- **Private :**
- The private access modifier ensures that class members are visible only to that class and are not accessible outside the containing class.

```
class Employee {
    private empCode: number;
    empName: string;
}

let emp = new Employee();
emp.empCode = 123; // Compiler Error
emp.empName = "Swati";//OK
```

Fundamentals:Data Modifiers

- Protected:
- The protected access modifier is similar to the private access modifier, except that protected members can be accessed using their deriving classes.

```
class Employee {
   public empName: string;
   protected empCode: number;

   constructor(name: string, code: number){
      this.empName = name;
      this.empCode = code;
   }
}

class SalesEmployee extends Employee{
   private department: string;
```

Fundamentals:Data Modifiers Continue..

- **Protected:**

```
class SalesEmployee extends Employee{
   private department: string;

   constructor(name: string, code: number, department: string) {
      super(name, code);
      this.department = department;
   }
}

let emp = new SalesEmployee("John Smith", 123, "Sales");
empObj.empCode; //Compiler Error
```

Fundamentals: Static

```
class Circle {
    static pi: number = 3.14;

    static calculateArea(radius:number) {
        return this.pi * radius * radius;
    }
}
Circle.pi; // returns 3.14
Circle.calculateArea(5); // returns 78.5
```

Fundamentals: Static and Non static

```
class Circle {
    static pi = 3.14;
    static calculateArea(radius:number) {
        return this.pi * radius * radius;
    }
    calculateCircumference(radius:number):number {
        return 2 * Circle.pi * radius;
    }
}
Circle.calculateArea(5); // returns 78.5
let circleObj = new Circle();
circleObj.calculateCircumference(5) // returns 31.4000000
//circleObj.calculateArea(); <-- cannot call this
```

Modules

The TypeScript code we write is in the global scope by default. If we have multiple files in a project, the variables, functions, etc. written in one file are accessible in all the other files.

example, consider the following TypeScript files: file1.ts and file2.ts

File1.ts : var greeting : string = "Hello World!";

File2.ts :

console.log(greeting); //Prints Hello World!

greeting = "Hello TypeScript"; // allowed

Modules

The above variable greeting, declared in file1.ts is accessible in file2.ts as well.

Not only it is accessible but also it is open to modifications.

Anybody can easily override variables declared in the global scope without even knowing others code.

This is a dangerous space as it can lead to conflicts/errors in the code.

TypeScript provides modules and namespaces in order to prevent the default global scope of the code and also to organize and maintain a large code base.

Modules

Modules are a way to create a local scope in the file. So, all variables, classes, functions, etc. that are declared in a module are not accessible outside the module.

A module can be created using the keyword **export** and a module can be used in another module using the keyword **import.**

Modules : Export

A module can be defined in a separate .ts file which can contain functions, variables, interfaces and classes. Use the prefix export with all the definitions you want to include in a module and want to access from other modules.

Modules : Export - Example

```
export let age : number = 20;
export class Employee {
  empCode: number;
  empName: string;
  constructor(name: string, code: number) {
    this.empName = name;
    this.empCode = code;
  }
  displayEmployee() {
    console.log ("Employee Code: " + this.empCode + ", Employee Name: " +
this.empName );
  }
}
let companyName:string = "XYZ";
```

Modules : Import

A module can be used in another module using an import statement.

Syntax:

Import { export name } from "file path without extension"

Modules : Import Example

- Importing a Single export from a Module.

```
import { Employee } from "./Employee";
let empObj = new Employee(“ramesh", 1);
empObj.displayEmployee(); //Output: Employee Code: 1, Employee Name: Ramesh
```

Modules : Import Example

- Importing the Entire Module into a Variable

```
import * as Emp from "./Employee"
console.log(Emp.age); // 20
let empObj = new Emp.Employee(“ramesh" , 2);
empObj.displayEmployee(); //Output: Employee Code: 2, Employee Name: ramesh
```

Modules : Import Example

- Renaming an Export from a Module:

```
import { Employee as Associate } from "./Employee"
let obj = new Associate(“kumar" , 3);
obj.displayEmployee();//Output: Employee Code: 3, Employee Name: kumar
```

Building Blocks of Angular

- The main building blocks of Angular are:

    Modules

    Components

    Templates

    Metadata

    Data binding

    Directives

    Services

    Dependency injection

Modules

- Angular apps are modular and to maintain modularity, we have *Angular modules* or you can say *NgModules*.
- Every Angular app contains at least one Angular module, i.e. the root module. Generally, it is named as *AppModule*.
- The *root module* can be the only module in a small application. While most of the apps have multiple modules.
- angular module is a class with @NgModule decorator.

Module

- ***Decorators*** are functions that modify JavaScript classes.
- Decorators are basically used for attaching metadata to classes
- NgModule is a decorator function that takes metadata object whose properties describe the module.
- **The properties are:**
- ***declarations:*** The classes that are related to views and it belong to this module. There are three classes of Angular that can contain view: components, directives and pipes. We will talk about them in a while.
- ***exports:*** The classes that should be accessible to the components of other modules.
- ***imports:*** Modules whose classes are needed by the component of this module.
- ***providers:*** Services present in one of the modules which is to be used in the other modules or components. Once a service is included in the providers it becomes accessible in all parts of that application
- ***bootstrap:*** The *root component* which is the main view of the application. This root module only has this property and it indicates the component that is to be bootstrapped.

Module- Example

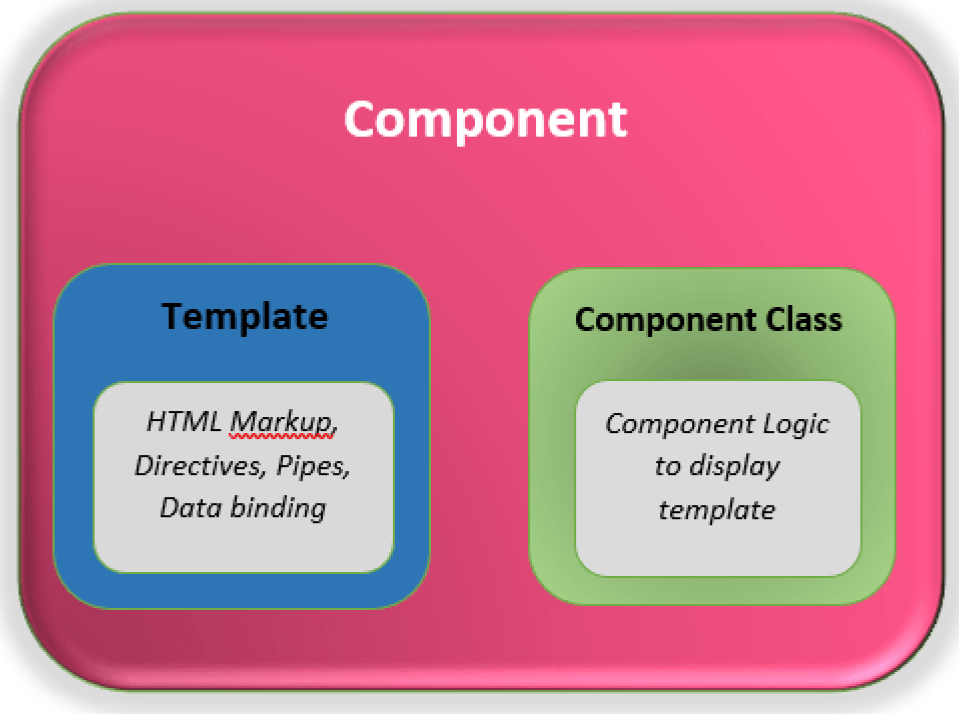
src/app/app.module.ts

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

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

export class AppModule { }
```

Component

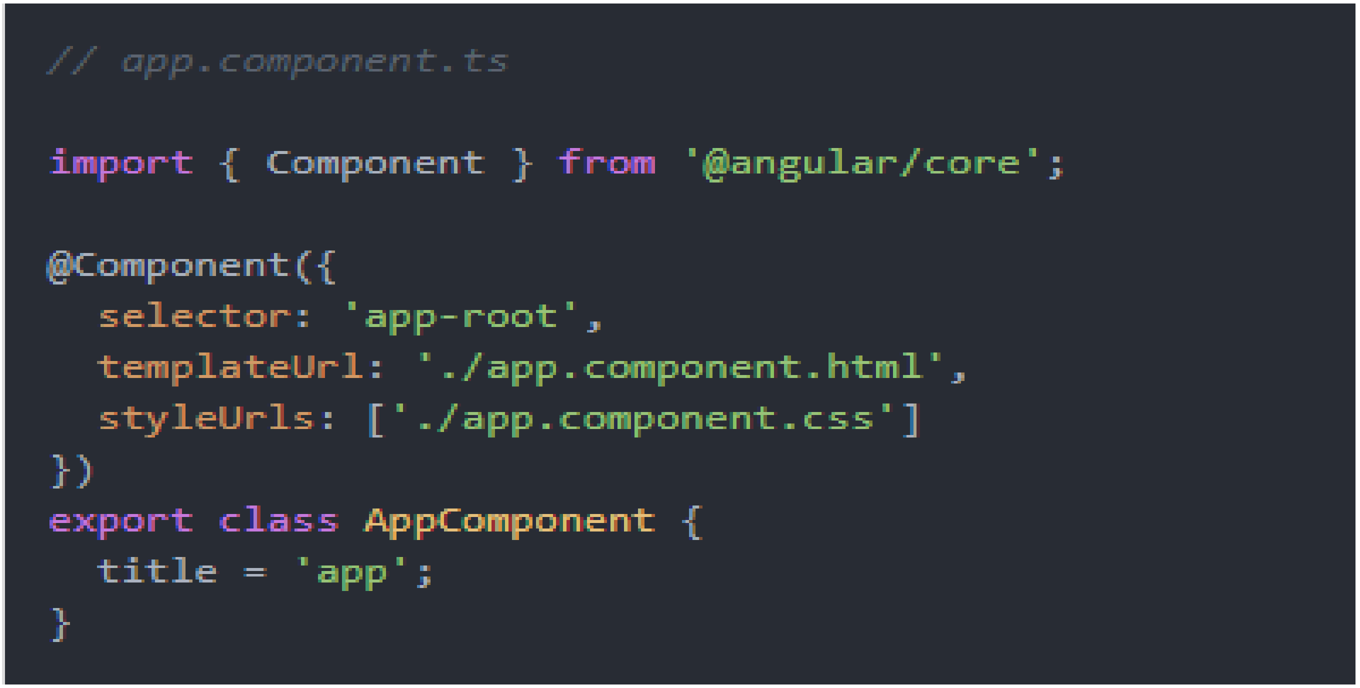
- component is the **view part** or **User Interface** of angular application. A single view (or page) on the screen could be made up of 1 component or it may be a combination of multiple components. Also, a component can be extended by other components which will be the sub-components or child components of this component.
- A component is made up of 2 parts: a class(written in typescript) which contains logic to display the view and a template that contains HTML markup. A template tells Angular how to display the view.
- The component class is annotated with @Component annotation.

Continue.,


Component
Template
HTML Markup, Directives, Pipes, Data binding
Component Class
Component Logic to display template

component

- **app.component.ts**
- defined by a decorator @Component. It contains **metadata.** Angular Application reads this and identifies this class as a component class.
- **Component = Class + Metadata + Template**



```
// app.component.ts

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

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

Component

first, we have imported a **Component** module from the angular core module.

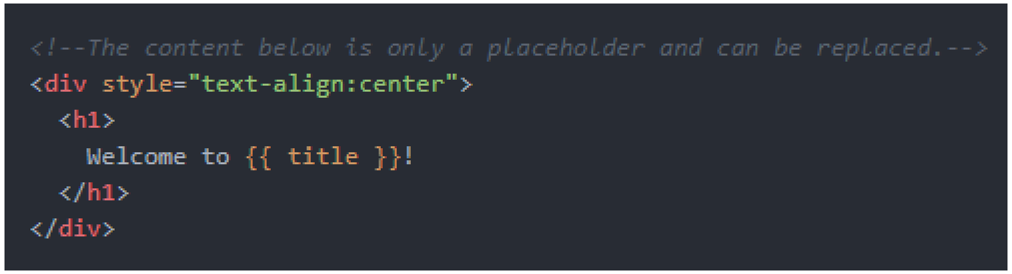
Then we have defined decorator and described the metadata.

Metadata contains three things

- **selector**
- **templateUrl**
- **styleUrls**

So it described as a which selector this component uses and what is the template URL and what is the Style URL. All the data described in the **Metadata**.

Apart from metadata, it contains typescript class. The class comprises of properties and methods. It is an essential **TypeScript** class. We can access its property in the template. In above's example **title** is the property and we can access this property in the **app.component.html** file.



```
<!--The content below is only a placeholder and can be replaced.-->
<div style="text-align:center">
  <h1>
    Welcome to {{ title }}!
  </h1>
</div>
```

Component

- **selector**
- It defines as HTML tag. In above's example, **app-root** is a selector, and we can use as a tag in the HTML file.
- **templateUrl**
- It represents a view of the component. We need to give the path of the HTML file. So, we can say it is the view of the MVC architecture. We can see Component as a Controller, and we can also define the Model as a Data of the component.
- **styleUrls**
- We give the path of the particular CSS file for a particular component.

Compontens Continue

- **we need to follow 3 steps:**

  Create the Component.

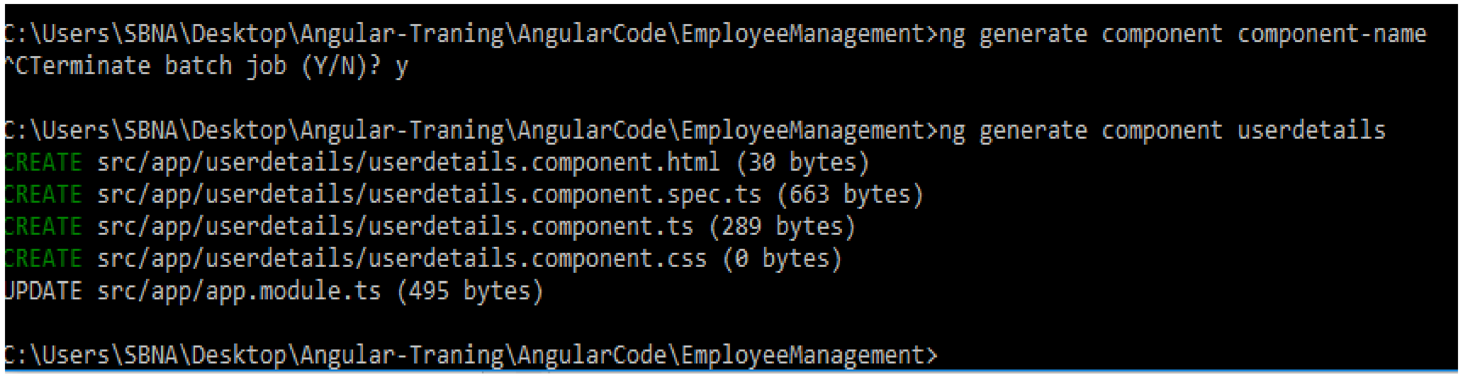
  Register the component in module.

  Add the element in HTML Markup.

Create the Component:

ng generate component component-name

Example :



```
C:\Users\SBNA\Desktop\Angular-Traning\AngularCode\EmployeeManagement>ng generate component component-name
^CTerminate batch job (Y/N)? y

C:\Users\SBNA\Desktop\Angular-Traning\AngularCode\EmployeeManagement>ng generate component userdetails
CREATE src/app/userdetails/userdetails.component.html (30 bytes)
CREATE src/app/userdetails/userdetails.component.spec.ts (663 bytes)
CREATE src/app/userdetails/userdetails.component.ts (289 bytes)
CREATE src/app/userdetails/userdetails.component.css (0 bytes)
UPDATE src/app/app.module.ts (495 bytes)

C:\Users\SBNA\Desktop\Angular-Traning\AngularCode\EmployeeManagement>
```

Register the component in module. App.module.ts

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { UserdetailsComponent } from './userdetails/userdetails.component';
@NgModule({
  declarations: [
    AppComponent,UserdetailsComponent
  ],
  imports: [BrowserModule,AppRoutingModule],
  providers: [],  bootstrap: [AppComponent]
})
export class AppModule { }
```

New Componet ts file

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

@Component({
  selector: 'app-userdetails',
  templateUrl: './userdetails.component.html',
  styleUrls: ['./userdetails.component.css']
})
export class UserdetailsComponent implements OnInit {
  constructor() { }
  ngOnInit() {
  }
}
```

Add the element in HTML Markup.

```
<p>
 userdetails works!
</p>
```

Template

Component can encapsulate data, logic, HTML markup for a view

Angular 6 uses the **<ng-template>** as the tag similar to Angular 4 instead of **<template>** which is used in Angular2. The reason Angular 4 changed **<template>** to **<ng-template>** is because there is a name conflict between the **<template>** tag and the html **<template>** standard tag.

- Introduction to the ng-template directive
- Template Input Variables
- The ng-template directive use with ngIf
- ngIf de-suggared syntax and ng-template
- ng-template template references and the TemplateRef injectable
- Configurable Components with Template Partial @Inputs
- The ng-container directive, when to use it?
- Dynamic Template with the ngTemplateOutlet custom directive
- Template outlet @Input Properties

Directives

Angular can be used to render the view dynamically. It provides special classes for this purpose called the directives.

Directives allows you to manipulate the DOM(or HTML document) at run time.

Using directives, you can add an element, remove an element, hide an element, add a class to an element or remove

- here are four types of directives in Angular,
- Components directives
- Structural directives
- Attribute directives
- Custom Directive

Components directives

- It is mainly used to specify the HTML templates. It is the most commonly-used directive in an Angular project. It is decorated with the @component decorator. This directive is a class. The component directive is used to specify the template/HTML for the Dom Layout. Its built-in is @component.

    app.component.css: contains all the CSS styles for the component

    app.component.html: contains all the HTML code used by the component to display itself

    app.component.ts: contains all the code used by the component to control its behavior

Services

- A Service is a class having certain operations for a specific purpose
- We use services in Angular to share the data among components.
- The sharing of data is also one of the responsibilities of the service.
- Service provides is application logic.
- We create a service for a certain request that interacts externally to get the data from somewhere and give that to the View.

- Demo in VSC

Drackback in Demo Code

- **public** students = [
-     {"id" : 1001, "name" :“Ramesh", "marks" : 90},
-     {"id" : 1002, "name" :“kumar", "marks" : 80},
-     {"id" : 1003, "name" : "Rahul", "marks" : 70},
-     {"id" : 1004, "name" : "Ajay", "marks" : 85},
-     {"id" : 1005, "name" :“Ravi", "marks" : 60}
-     ];
- This same data is being used in both the components, which means the component is not only responsible for displaying or working with the data, it is also taking the responsibility of generating the data,
- which actually should not be its responsibility
- These components should only be responsible to use the given data

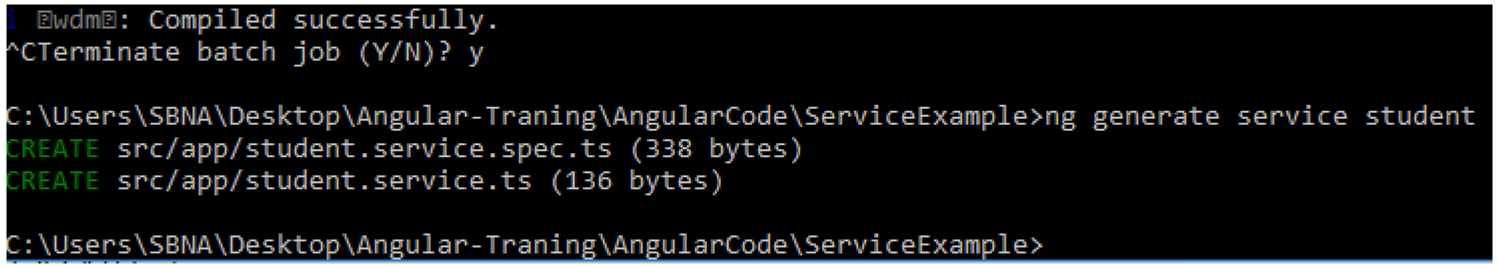
Service uses

We should have a service that will be injected into these two components and should solely be responsible to deliver the data to these components.

These components should only perform one responsibility: dealing with how to use the data given by the service.

This makes the service responsible for generating the data and components are responsible to make it display.

ng generate service student



```
i ｢wdm｣: Compiled successfully.
^CTerminate batch job (Y/N)? y

C:\Users\SBNA\Desktop\Angular-Traning\AngularCode\ServiceExample>ng generate service student
CREATE src/app/student.service.spec.ts (338 bytes)
CREATE src/app/student.service.ts (136 bytes)

C:\Users\SBNA\Desktop\Angular-Traning\AngularCode\ServiceExample>
```

Dependency Injection

Dependency Injection (DI) is a way to create objects that depend on the other objects. A Dependency Injection system supplies the dependent objects

Dependency Injection is a pattern in which a class receives its dependencies from an external source rather than creating a new one.

Why should we use Dependency Injection?

```
export class House {
    private bricks: Bricks;
    constructor() {
        this.bricks = new Bricks();
    }
}
export class Bricks {
    constructor() {}
}
```

3 fundamental problems with this code.

- This code is difficult to maintain over time
- Instances of the dependencies created by a class that needs those dependencies are local to the class and cannot share the data and the logic.
- Hard to unit test

Dependency Injection

- rewritten the above code using dependency injection (DI),
- **export class** House {
-     **private** bricks: Bricks;
-     constructor(bricks: Bricks) {
-         **this**.bricks = bricks;
-     }
- }
- **export class** Bricks {
-     constructor(size: number) {}
- }

Dependency Injection

- Dependency Injection provides benefits as mentioned below –
- create applications that are easy to write and maintain over time as the application evolves.
- Easy to share data and functionality because the Angular injector provides a Singleton, i.e., a single instance of the service.
- Easy to write and maintain unit tests as the dependencies can be mocked.

injectable service.

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

@Injectable()
export class PopcornService {

  constructor() {
    console.log("Popcorn has been injected!");
  }

  cookPopcorn(qty) {
    console.log(qty, "bags of popcorn cooked!");
  }

}
```

How you would inject our abc service it in a component:

```
import { Component } from '@angular/core';
import { PopcornService } from './popcorn.service';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
  providers: [PopcornService]
})
export class AppComponent {
  constructor(private popcorn: PopcornService) {}

  cookIt(qty) {
    this.popcorn.cookPopcorn(qty);
  }

}
```

Cont…

- The cookIt() method in the template calls the cookPopcorn() method in the injected service. Let’s make use of our cookIt() method in our template:

- <input type="number" #qty placeholder="How many bags?"> <button type="button" (click)="cookIt(qty.value)"> Cook it! </button>

Data Binding

- Data-binding means communication between the TypeScript code of your component and your template which the user sees.
- business logic in your component TypeScript code to fetch some dynamic data from the server and want to display this dynamic data to the user via template because the user sees only the template.
- we need some kind of binding between your TypeScript code and template (View).

Data-binding can be either one-way or two-way.Angular provides various types of data binding

- String Interpolation
- Property Binding
- Event Binding
- Two-way binding

String Interpolation

String Interpolation is a one-way data-binding which is used to output the data from a TypeScript code to HTML template (view). It uses the template expression in double curly braces to display the data from the component to the view.

Example :

{{ data }}

app.component.ts:

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

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

app.component.html

```
<h2>
  {{ title }}
</h2>
```

Property Binding

- Property binding is also a one-way data binding, where we bind a property of a DOM element to a field which is a property we define in our component typescript code. Behind the scene, Angular converts string interpolation into property binding.

- property binding for just displaying some value between h2 heading
- *<h2 [textContent]="title"></h2>*

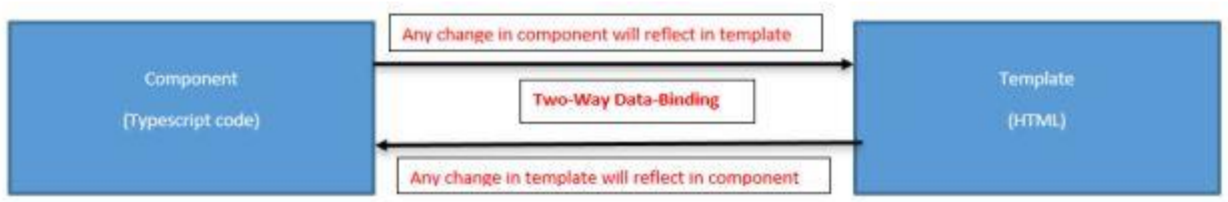
- *String Interpolation and Property binding both are one-way binding*
- **app.componnt.ts**
- **export class** AppComponent {
-   title = "Data binding using Property Binding";
-   imgUrl="https://avatars2.githubusercontent.com/u/20270535?s=40&v=4";
- }
- **app.component.html**
- <h2>{{ title }}</h2> <!-- String Interpolation -->
- <img [src]="imgUrl" /> <!-- Property Binding -->

Two-Way Data Binding

we have seen how to bind component data to view using one-way bindings. That means any change in the template(view) will not be reflected in the component typescript code.

two-way binding has a feature to update data from component to view and vice-versa.

Syntax - For two-way data binding, we combine property binding and event binding bothTo implement two-way data binding, you need to enable the ngModel directive and this depends upon FormsModule in angular/forms package. So you need to add FormsModule in imports[] array in the AppModule.


Component
(Typescript code)
Any change in component will reflect in template
Two-Way Data-Binding
Any change in template will reflect in component
Template
(HTML)

Continue.,

app.module.ts:

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

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

Continue.,

app.component.html

- <h2>Two-way Binding Demo</h2>
-    <input [(ngModel)]="fullName" /> <br/><br/>
- <p> {{fullName}} </p>

app.component.ts

- **import** { Component } from "@angular/core";
-
- @Component({
-   selector: "app-root",
-   templateUrl: "./app.component.html",
-   styleUrls: ["./app.component.css"]
- })
- **export class** AppComponent {
-   fullName: string = "Rahul";
- }

Routing

- Routing is a simple mechanism in a web application by which the requests are routed to the code that handles them
- we say navigation in an Angular application
- routing can be achieved using an Angular module called "RouterModule". It is defined inside "@angular/router".
- Demo -- (Angular-6-CRUD-Operation-master) VSC

Angular - Pipes

- Pipes are used to format the data before displaying in the View.
- Pipe is used by using **|.** This symbol is called a Pipe Operator.
- Types of pre-defined or built-in pipes in Angular 6.

  Lowercase

  Uppercase

  Titlecase

  Slice

  Json Pipe

  Number Pipe

  Percent Pipe

  Currency Pipe

  Date Pipe

Angular - Pipes

Title Example

App.component.ts:

```
import {
   Component
} from '@angular/core';
@Component({
   selector: 'app-root',
   template: `
<h2>{{name|titlecase}}</h2>
`,
})
export class AppComponent {
   name = 'Welcome to angular pipes project';
}
```

Angular - Pipes

JSONExample

App.component.ts:

```
import {  Component  } from '@angular/core';
@Component({
   selector: 'app-root',
   template: `
<h2>{{cities|json}}</h2>
`,
})
export class AppComponent {
   name = 'AngularPipes';
   public cities = {
      "city": "Jaipur",
      "country": "India"
   }
}
```

Angular - Pipes

Currenncy xample

App.component.ts:

```
import {   Component  } from '@angular/core';
@Component({
    selector: 'app-root',
    template: `
<h2>{{8.7844|number:'1.2-2'}}</h2>
<h2>{{0.23|percent}}</h2>   <h2>{{0.23|currency}}</h2>
<h2>{{0.23|currency:'GBP'}}</h2>
`,
})
export class AppComponent {
    name = 'AngularPipes';
}
```

Angular - Pipes

Date Example

App.component.ts:

```
import {  Component  } from '@angular/core';
@Component({
   selector: 'app-root',
   template: `
<h2>{{date}}</h2>
<h2>{{date|date:'short'}}</h2>  <h2>{{date|date:'shortdate'}}</h2>
`,
})
export class AppComponent {
   name = 'AngularPipes';
}
```

Authentication and Authorization

- demo