Angular A Deep Dive

Last edited : 31/07/2019

Author : Paseka Monyeki

Edited : Vinod Bhavnani

Contents

Prerequisite

This document is intended for developers how want a firm rooting in Angular; the developer should be familiar with vanilla JavaScript, CSS3 and HTML5. At the time of release of this document Angular was on version 7.

1)What is TypeScript

TypeScript is a typed superset of JavaScript that compiles to plain JavaScript. This means that every valid statement of JavaScript is a valid statement of TypeScript. TypeScript is pure object oriented with classes, interfaces and statically typed like C# or Java.

2)Why TypeScript

TypeScript is a fully object oriented language unlike JavaScript, the advantage of an object oriented language is that code is structured properly and it is easier to apply concepts such as classes, interfaces, inheritance etc. which makes programming much easier.

TypeScript is strongly typed meaning that you can't change the type of a variable once it is declared.

Compilation – TypeScript gives you compile time errors while JavaScript cannot. It makes development easier and efficient.

Inheritance:

Vanilla JavaScript does not support inheritance directly unlike TypeScript. Below is an example of how easy and elegant it is to perform inheritance in TypeScript.

```
class MyClass {

     name: String;
     id: number;

    constructor(name){
      this.name = name;
      this.id = 1;
    }

    getName(){
      return this.name;
    }

  }

class ChildClass extends MyClass {

      display() : void {

        console.log("I am a child");

      }

  }
```

The equivalent JavaScript of the above TypeScript inheritance code is as follows, as can be observed it is cumbersome and has much more lines of code. This is another reason why TypeScript is preferred.

```
var __extends = (this && this.__extends) || (function () {
  var extendStatics = function (d, b) {
      extendStatics = Object.setPrototypeOf ||
          ({ __proto__: [] } instanceof Array && function (d, b) { d.__proto__ =
b; }) ||
          function (d, b) { for (var p in b) if (b.hasOwnProperty(p)) d[p] =
b[p]; };
      return extendStatics(d, b);
  };
  return function (d, b) {
      extendStatics(d, b);
```

```
      function __() { this.constructor = d; }
      d.prototype = b === null ? Object.create(b) : (__.prototype = b.prototype,
new __());
  };
})();


var MyClass = /** @class */ (function () {
  function MyClass(name) {
      this.name = name;
      this.id = 1;
  }
  MyClass.prototype.getName = function () {
      return this.name;
  };
  return MyClass;
}());


var ChildClass = /** @class */ (function (_super) {
  __extends(ChildClass, _super);
  function ChildClass() {
      return _super !== null && _super.apply(this, arguments) || this;
  }
  ChildClass.prototype.display = function () {
      console.log("I am a child");
  };
  return ChildClass;
}(MyClass));


var achild = new ChildClass("Bradley");
```

3)Basic Syntax (Types)

Variables:

In TypeScript variable definition follows the following format

```
var indentifier : type = value;
```

for example

```
var message: string = "My variable";
```

TypeScript Functions:

Functions form fundamental building blocks of any JavaScript application. In TypeScript while there are classes, namespaces and modules, functions play a vital role of describing how to do things.

1. Simple Definition
   Syntax :

```
function  function_name() {
// function body
}
```

Example :

```
getMethod(){
    console.log("This is a message");


}
```

Functions in Typescript can refer to variables outside the function body. For example

```
let c = 1000;

function addToC(a, b){
    return a + b + c;
}
```

2. Parameterized functions

Syntax :

```
function func_name( param1 [:datatype], param2 [:datatype]) {


}
```

Example:

```
logForm(value: any) {
    alert(value.firstname+" "+value.lastname+" "+
    value.city+" "+value.street+" "+value.zip);
}
```

Optional and Default Parameters

In TypeScript every parameter is assumed to be required by the function. The compiler also assumes that these parameters are the only parameters that will be passed to the function. In short, the number of arguments given to a function has to match the number of parameters the function expects.

Example :

```
//function accepting exactly two parameters
function buildName(firstName: string, lastName: string) {
  return firstName + " " + lastName;
}

// error, too few parameters
let result1 = buildName("Paseka");

// error, too many parameters
let result2 = buildName("Paseka", "Suchismita", "Vinod");

// ah, just right
let result3 = buildName("Bob", "Adams");
}
```

In JavaScript, every parameter is optional, and users may leave them off as they see fit. When they do, their value is underfined. We can get this functionality in TypeScript by adding a ? to the end of parameters we want to be optional. For example, let's say we want the last name parameter from above to be optional:

```
//function accepting exactly two parameters , one being optional

function buildName(firstName: string, lastName?: string) {

  return firstName + " " + lastName;
}

// works perfectly
let result1 = buildName("Paseka");

// error, too many parameters
let result2 = buildName("Paseka", "Suchismita", "Vinod");

// ah, just right
let result3 = buildName("Bob", "Adams");
}
```

Rest Parameters

Rest parameters are treated as a boundless number of optional parameters. When passing arguments for a rest parameter, you can use as many as you want; you can even pass none. The compiler will build an array of the arguments passed in with the name given after the ellipsis (`...`), allowing you to use it in your function.

```
//A rest function defintion , note the use of ellipsis
function buildName(firstName: string, ...restOfName: string[]) {
  return firstName + " " + restOfName.join(" ");
}

let buildNameFun: (fname: string,...rest: string[]) => string = buildName;
```

Function with return type

In TypeScript you are given a choice on whether you want to specify a return type for your method or not. The return type defines and constrains the data type of the value returned from a method.

Syntax:

```
function function_name():return_type {
    //statements
    return value;
}
```

Example:

```
//This is a method which has a return type of string
getEmail() : string  {
    return this.profileForm.get('email');
}
```

Below is a table showing all basic types of TypeScript and brief explanation

Type	Explanation	Example
Number	In TypeScript, numbers are floating point values that have a type of number. You can assign any numeric values including decimals, hexadecimals, binary, and octal literals.	`getNumber() : number{ return 7; }`
String	When you want to use textual data, string types are used and get denoted by the keyword string. Like JavaScript, TypeScript also uses double quotes (") and single quotes (') to surround the string value.	`getEmail() : string  { return this.profileForm.get('email'); }`
Boolean	When you need data that can be expressed as Boolean values i.e true , false , 0 , 1	`getValue() : boolean{ return true; }`
Void	In general this type is used in functions that do not return any value.	`printSomething() : void { console.log("Hello world"); }`

Null	You can declare a variable of type null using the null keyword and can assign only null value to it. As null is a subtype of all other types, you can assign it to a number or a boolean value.	`mynullvalue : null;`
		
		`getNull() : null{`
		`  return this.mynullvalue;`
		`}`

Undefined	You can use the undefined keyword as a data type to store the value undefined to it. As undefined is a subtype of all other types, just like null, you can assign it to a number or a boolean value.	`numericValue : undefined;`
		`getUndefined() : undefined{`
		`  return this.numericValue;`
		`}`

Any	While writing code for which you are unsure of the data type of a value, due to its dynamic content, you can use the keyword any to declare said variable. For example you want an array of mixed datatypes or receiving input from a third party service.	`dynamicVAlue : any = "Paseka Moyenki";`
		
		`getValue() : any{`
		`  return this.dynamicVAlue;`
		`}`

There are advanced return types that can be explored within TypeScript like array , tuple , never etc.

Lambda Functions

Lambda functions are used for anonymous functions i.e for function expressions. Using fat arrow (=>) we drop the need to use the 'function' keyword. Parameters are passed in the angular brackets (), and the function expression is enclosed within the curly brackets {}. See following examples of lambda function

```
//a lambda function with no parameters
() => {}
```

```
//lambda function with no parameter and no return value
var myFunction = () => {
  console.log("Hello World");
};

// calling
```

```
myFunction();
```

Lambda functions that return a value

Syntax:

```
(): return_type => {
  return value;
}
```

Example:

```
//lambda  functions  with no parameters but with return type
var myFunction = (): number => {
  return Math.random();
};

// calling
console.log("Just another number : " +myFunction());
```

Lambda functions that return a value and has parameters

Syntax:

```
//lambda functions with parameters and return type
(value1: type, value2: type, ...): type => {
  return value;
}
```

Example:

```
//lambda functions with parameters and return type
var getPoints = (username: string, points: number): string => {
    return username+" scored "+ points +" points! ";
}
//calling
console.log(getPoints('Sean Combs', 48));
```

Lambda functions inside a Class definition

Syntax:

```
class class_name {

    function_name = () => function_definition

}
```

Example:

```
//create a player class
class Player{

    //class propeties of a player
    playerCode: number;
    playerName: string;

    constructor(code: number, name: string) {
            this.playerName = name;
            this.playerCode = code;
    }

    displayPlayer = () =>{ alert(this.playerCode +' '+ this.playerName);}
  }

  //create a player object
  let myplayer = new Player(1, 'Star');
  //invoke a lambda function
  myplayer.displayPlayer();
```

Objects

Class definition syntax:

Syntax :

```
class class_name {
    //attributes definition
    //methods definition
}
```

Example:

```
class MyClass{
     name: String;
     id: number;

    constructor(name){
       this.name = name;
       this.id = 1;
    }

    getName(){
       return this.name;
    }
}
```

Object:

This is how we create an object of type “myObject”, we can then interact with the object via its methods like getName().

```
createObject(){
   let object = new MyClass("Sasha");
   object.getName();
}
```

TypeScript keywords

Keywords are words which are responsible to perform some specific task or the words which represent some specific functionality

In TypeScript, we have the following Keywords:

Reserved Words	Strict Mode Reserved Words	Contextual Keywords
Break	As	Any
Case	Implements	Boolean
Catch	Interface	Constructor
Class	Let	Declare
Const	Package	Get
Continue	Private	Module
Debugger	Protected	Require
Default	Public	Number
Delete	Static	Set
Do	Yield	string
Else	Symbol	
Enum	Type	
Export	From	
Extends	Of	
False		
Finally		
For		
Function		
If		
Import		
In		
Instanceof		

New		
Null		
Return		
Super		
Switch		
This		
Throw		
True		
Try		
Typeof		
Var		
Void		
While		
With		

This is a brief explanation of what some of the typescript keywords mean

TypeScript Keyword	Meaning
New	is for object creation
This	is used to refer to an instance variable of a class or object
Extends	Is for implementing heritance between child and parent class
Super	Is used in expressions to reference base class properties and the base class constructor.
Break	Is used to jump of out the loop
Continue	statement breaks one iteration (in the loop)

Number	is a type of a variable that specifies that variable can only a numeric value like an integer.
Instanceof	The instanceof operator allows to check whether an object belongs to a certain class
Import	Is that a module or its contents i.e classes that are defined in other modules can be used in the current module
Export	Specifies that a class can be used globally within the application

4)Difference between let and var

let : gives you the privilege to declare variables that are limited in scope to the block, statement of expression unlike var.

var : is rather a keyword which defines a variable globally regardless of block scope.

Difference between let and var

Let is for defining variables that have a limited scope meaning that they are not needed globally we just use them within the scope of the block like in a for loop, while loop, function etc.

var is used for defining variables that are used globally regardless of block scope.

RxJS (Observables, Subjects, Behavior Subjects, Replay Subjects)

RxJS (Reactive Extentions for JavaScript)

What is RxJS – It is JavaScript's implementation of reactive programming.

RxJS is a library for composing asynchronous and event-based programs by using observable sequences. It provides one core type, the Observable, satellite types (Observer, Schedulers, and Subjects).

Reactive Programming – is used to handle asynchrounous calls, asynchronous simply means that you do not have to wait for the response of a task to complete; you can execute another task while the other one is still in progress.

Why we need it – previously asynchronous calls in JavaScript were implemented with Ajax (asynchronous JavaScript and xml) the drawback with this method was that we had to have a setTimeout method for cases where we were dealing with data streams that had data that changed regularly which was a tedious job. With RxJs through observables once, we have subscribed to the data stream we will be notified of the change in data as soon as it happens.

Important concepts in Rxjs

- **Observable:** Observables are a way to provide support for passing messages between publishers and subscribers in your application.

  Observables define a function for emitting values but is not executed until a consumer of the data subscribes to it. The subscribed consumer then receives notifications until the function completed or they unsubscribe to the observable.

- **Observer:** is a collection of callbacks that knows how to listen to values delivered by the Observable.
- **Subscription:** represents the execution of an Observable, is primarily useful for cancelling the execution.

- **Operators:** are pure functions that enable a functional programming style of dealing with collections with operations like map, filter, concat, flatMap, etc.

- **Subject:** A subject in reactive programming is a special hybrid object that can act as both an observable and an observer at the same time. Data can be pushed into a subject and objects that subscribed to the subject will receive the pushed data.

- **Schedulers:** are centralized dispatchers to control concurrency, allowing us to coordinate when computation happens on e.g. setTimeout or requestAnimationFrame or others.

Observables operators

Operators	Signature	Explanation
forkJoin	forkJoin(...args, selector : function): Observable	This operator is best used when you have a group of observables and only care about the final emitted value of each
Concat	concat(observables: ...*): Observable	Concatenates multiple Observables together by sequentially emitting their values, one Observable after the other.
concatAll	*concatAll(): Observable*	Converts a higher-order Observable into a first-order Observable by concatenating the inner Observables in order.
concatMap	concatMap(project: function, resultSelector: function): Observable	Maps each value to an Observable, then flattens all of these inner Observables using concatAll
MergeMap	mergeMap(project: function: Observable, resultSelector: function: any, concurrent: number): Observable	This operator is best used when you wish to flatten an inner observable but want to manually control the number of inner subscriptions.
switchMap,	switchMap(project: function: Observable, resultSelector: function(outerValue, innerValue, outerIndex, innerIndex): any): Observable	The main difference between switchMap and other flattening operators is the cancelling effect
Pipe	public pipe(operations: ...*): Observable	Used to stitch together functional operators into a chain.
retryWhen	signature: retryWhen(receives: (errors: Observable) => Observable, the: scheduler): Observable	Returns an Observable that mirrors the source Observable with the exception of an error. If the source Observable calls error, this method will emit the Throwable that caused the error to the Observable returned from notifier

To know more about subjects and their operators go to https://www.learnrxjs.io/

Important methods when working with observables

Next() - Required. A handler for each delivered value. Called zero or more times after execution starts.

Subscribe() -For receiving data from a subject you would have to implement one or more of the following methods.

Error() - Optional. A handler for an error notification. An error halts execution of the observable instance.

Complete() -Optional. A handler for the execution-complete notification. Delayed values can continue to be delivered to the next handler after execution is complete.

Unsubscribe() - This method can be used to remove the subscription when we no longer need it.

Subject

Subjects are useful for multicasting or for when a source of data is not easily transformed into an observable. For example see the code.

1) Import Subjects from rxjs package

```
import { Subject } from 'rxjs';
```

2) Use subjects in the following manner

```
//This code does not run
const mySubject =  new Subject<number>();

mySubject.next(1);

const subscription1 = mySubject.subscribe(x => {
    console.log('From subscription 1:', x);
});

mySubject.next(2);

const subscription2 = mySubject.subscribe(x => {
    console.log('From subscription 2:', x);
});

mySubject.next(3);

subscription1.unsubscribe();

mySubject.next(4);
```

The following output will be printed on the console after running the above code

```
From subscription 1: 2
```

```
From subscription 1: 3
From subscription 2: 3
From subscription 2: 4
```

ReplaySubject

As you saw previously, late subject subscriptions will miss the data that was emitted previously. Replay subjects can help with that by keeping a buffer of previous values that will be emitted to new subscriptions.

Here's a usage example for replay subjects where a buffer of 2 previous values are kept and emitted on new subscriptions:

```
//This code does not run
const mySubject = new Rx.ReplaySubject(2);

mySubject.next(1);
mySubject.next(2);
mySubject.next(3);
mySubject.next(4);

mySubject.subscribe(x => {
  console.log('From 1st sub:', x);
});

mySubject.next(5);

mySubject.subscribe(x => {
  console.log('From 2nd sub:', x);
});
```

The following output will be printed on the console after running the above code

```
From 1st sub: 3
From 1st sub: 4
From 1st sub: 5
From 2nd sub: 4
```

```
From 2nd sub: 5
```

BehaviorSubject

One of the variants of the Subject is the BehaviorSubject. The BehaviorSubject has the characteristic that it stores the "current" value. This means that you can always directly get the last emitted value from the BehaviorSubject.

There are two ways to get this last emited value. Either you can get the value by accessing the " .value" property on the BehaviorSubject or you can subscribe to it. If you subscribe to it, the BehaviorSubject will directly emit the current value to the subscriber. Even if the subscriber subscribes much later than the value was stored. See the example below:

```
//code does not output
const mySubject = new Rx.BehaviorSubject('Hey now!');


mySubject.subscribe(x => {
 console.log('From 1st sub:', x);
});


mySubject.next(5);


mySubject.subscribe(x => {
      console.log('From 2nd sub:', x);
});
```

The following output will be printed on the console after running the above code

```
From 1st sub: Hey now!
From 1st sub: 5
From 2nd sub: 5
```

6)New Function Syntax and Types

7) Fundamentals: what is Angular

Angular is a JavaScript framework that is based on TypeScript for front end development. It is led by the Angular team at Google and by a community of individuals and corporations.

8)Why Angular

Angular pre-packaged with dependencies that it needs for development you do not have to manually install them unlike React. It is used to create single page application.

Angular	React
Angular is a fully-fledged MVC framework	Is merely a JavaScript library(Just the view)
dependency injection	No dependency injection
Real DOM	Virtual DOM
Two-way Data binding	One Way Data binding
Is based on TypeScript	Is based on vanilla JSX

Angular has benefits and features as follows:

Benefits:

- Cross platform: Angular can be used to develop web, mobile and desktop applications.
- Speed and Performance: since angular is component based, applications are more efficient since everything is not loaded all at once.

- productivity: Angular CLI (command line interface) which provides commands to start building fast, add components and tests, then instantly deploy
- reduces development time

9) Setup and Installation

For one to have project up and running you need:

- Install Visual Studio code
- Node.js (to install npm)
- Angular CLI

Visual Studio Code is the preferred code editor when developing angular applications because it is lightweight, it has an integrated terminal and other features which make developing angular applications a breeze.

What is Node.js ?

Node.js – Node.js is a JavaScript runtime environment. Sounds great, but what does that mean? How does that work?

The Node run-time environment includes everything you need to execute a program written in JavaScript.

Node.js came into existence when the original developers of JavaScript extended it from something you could only run in the browser to something you could run on your machine as a standalone application.

Now you can do much more with JavaScript than just making websites interactive.

JavaScript now has the capability to do things that other scripting languages like Python can do.

Both your browser JavaScript and Node.js run on the V8 JavaScript runtime engine. This engine takes your JavaScript code and converts it into a faster machine code. Machine code is low-level code which the computer can run without needing to first interpret it.

Why Node.js ?

Runs outside browser - Node.js makes it possible to write applications in JavaScript on the server.

Asynchronous and Event Driven – All APIs of Node.js library are asynchronous, that is, non-blocking. It essentially means a Node.js based server never waits for an API to return data. The server moves to the next API after calling it and a notification mechanism of Events of Node.js helps the server to get a response from the previous API call.

Very Fast – Being built on Google Chrome's V8 JavaScript Engine, Node.js library is very fast in code execution.

No Buffering – Node.js applications never buffer any data. These applications simply output the data in chunks.

License – Node.js is released under the MIT license

What is NPM(Node Package Manager)

npm is the world's largest Software Registry. The registry contains over 800,000 code packages. Open-source developers use npm to share software. Many organizations also use npm to manage private development.

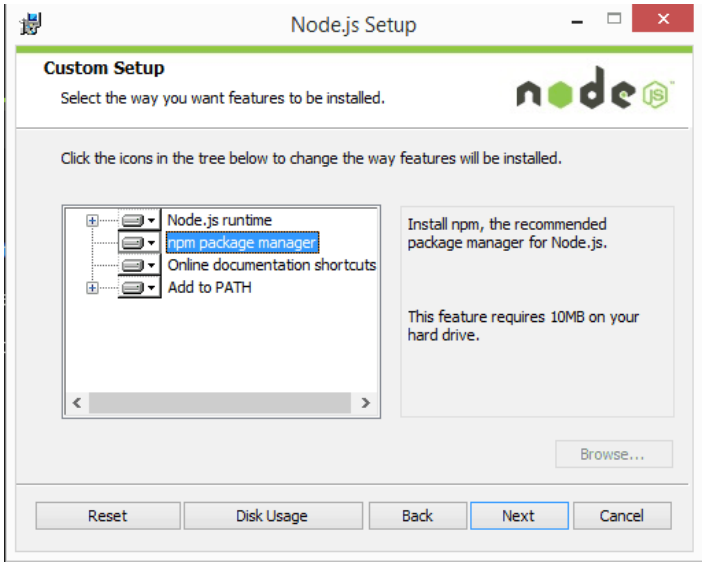
npm can install packages in local or global mode. In local mode it installs the package in a `node_modules` folder in your parent working directory.

npm can manage dependencies. npm can (in one command line) install all the dependencies of a project. Dependencies are also defined in package.json.

How to use NPM (Node Package Manager)

- It is used to install project dependencies and many more, in our case is to install project dependencies.
- Official site to download(https://nodejs.org)

Choose NPM package manager when installing



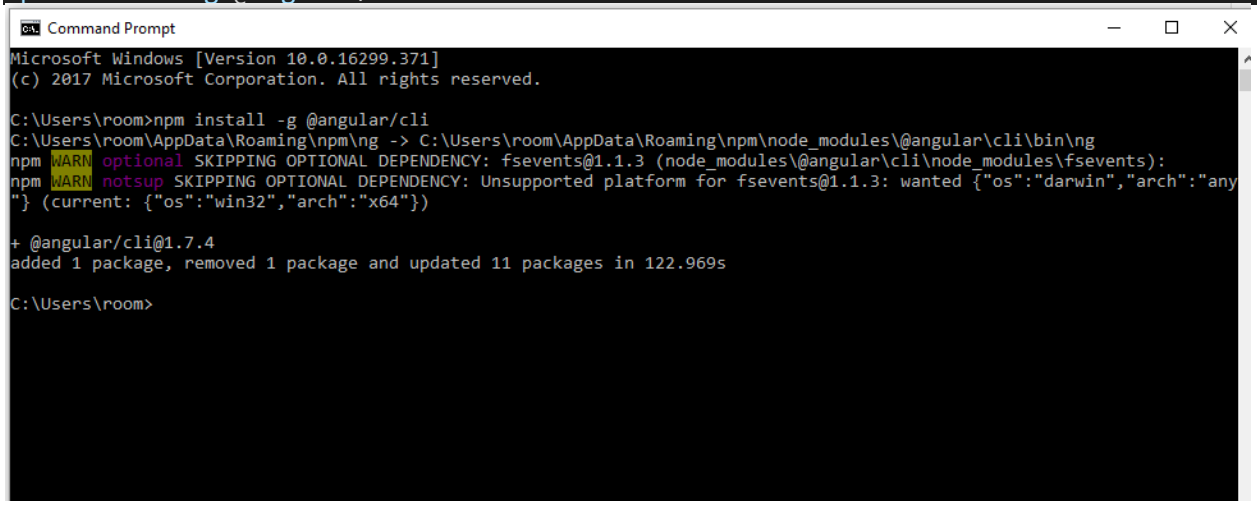
What is angular CLI

The Angular CLI is a command-line interface tool that you use to initialize, develop, structure and maintain Angular applications.

Step 1: Install the Angular CLI

Open command prompt and then enter the following command "*npm install -g @angular/cli*"

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



Commands of angular cli

`add` - Adds support for an external library to your project.

`build` - Compiles an Angular app into an output directory named dist/ at the given output path. Must be executed from within a workspace directory.

`config` - Retrieves or sets Angular configuration values in the angular.json file for the workspace.

`e2e` - Builds and serves an Angular app, then runs end-to-end tests using Protractor.

`generate` - Generates and/or modifies files based on a structure.

You can use the `ng generate` command to add features to your existing application:

- `ng generate class my-new-class`: add a class to your application
- `ng generate component my-new-component`: add a component to your application
- `ng generate directive my-new-directive`: add a directive to your application
- `ng generate enum my-new-enum`: add an enum to your application
- `ng generate module my-new-module`: add a module to your application
- `ng generate pipe my-new-pipe`: add a pipe to your application
- `ng generate service my-new-service`: add a service to your application
- `ng generate **universal** [options]` Pass this schematic to the "run" command to set up server-side rendering for an app.
- `ng generate **serviceWorker** [options]` Pass this schematic to the "run" command to create a service worker
- `ng generate **library** <name> [options]` Creates a new generic library project in the current workspace.
- `ng generate **interface** <name> <type> [options]` Creates a new generic interface definition in the given or default project.

The `generate` command and the different sub-commands also have shortcut notations, so the following commands are similar:

- `ng g cl my-new-class`: add a class to your application
- `ng g c my-new-component`: add a component to your application
- `ng g d my-new-directive`: add a directive to your application
- `ng g e my-new-enum`: add an enum to your application
- `ng g m my-new-module`: add a module to your application
- `ng g p my-new-pipe`: add a pipe to your application
- `ng g s my-new-service`: add a service to your application.

Each of the different sub-commands performs a different task and offers different options and parameters.

`help` - Lists available commands and their short descriptions.

`new` – Creates a new workspace and an initial Angular app.

`serve` - Builds and serves your app, rebuilding on file changes.

`test` - Runs unit tests in a project.

`update` - Updates your application and its dependencies.

`version`- Outputs Angular CLI version.

Step 2: Create an initial application and accessing it through the browser

In your command console type in the following commands in order

```
ng new NameofYourProject
```

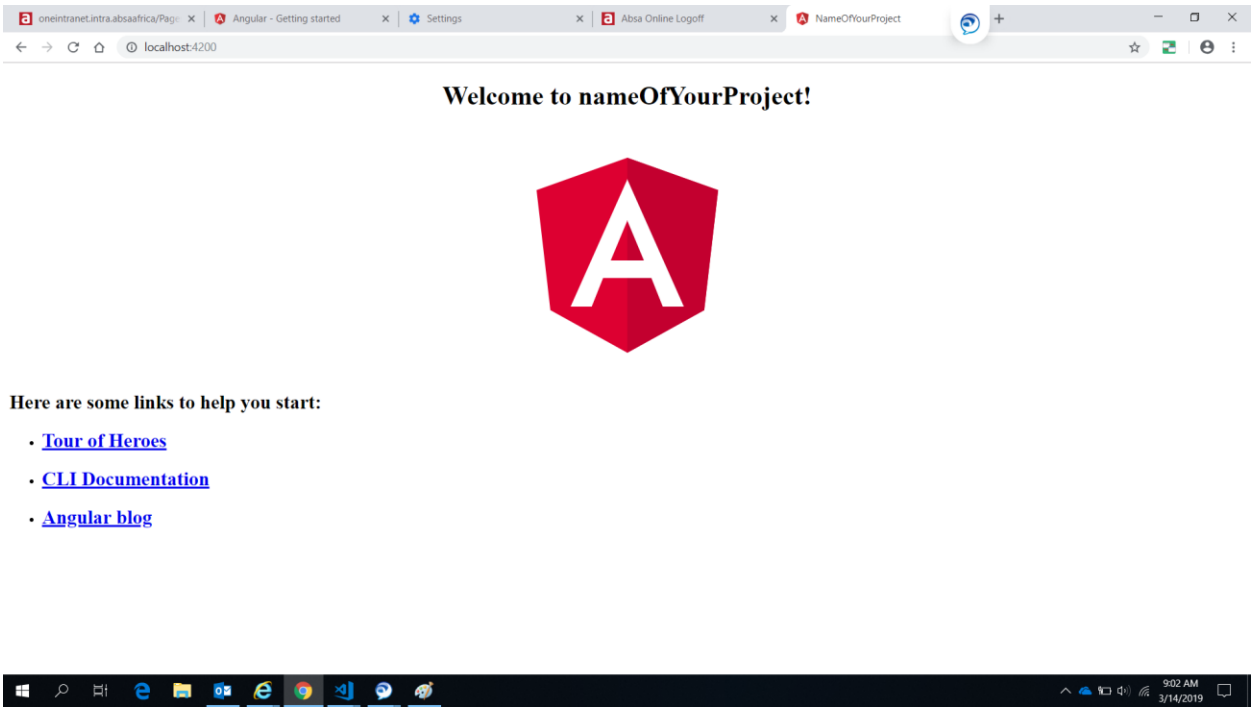
Change to directory of your project

```
cd nameOfYourProject
```

```
ng serve
```

In your browser of choice, open the following url in your address bar http://localhost:4200/

Deployed Project

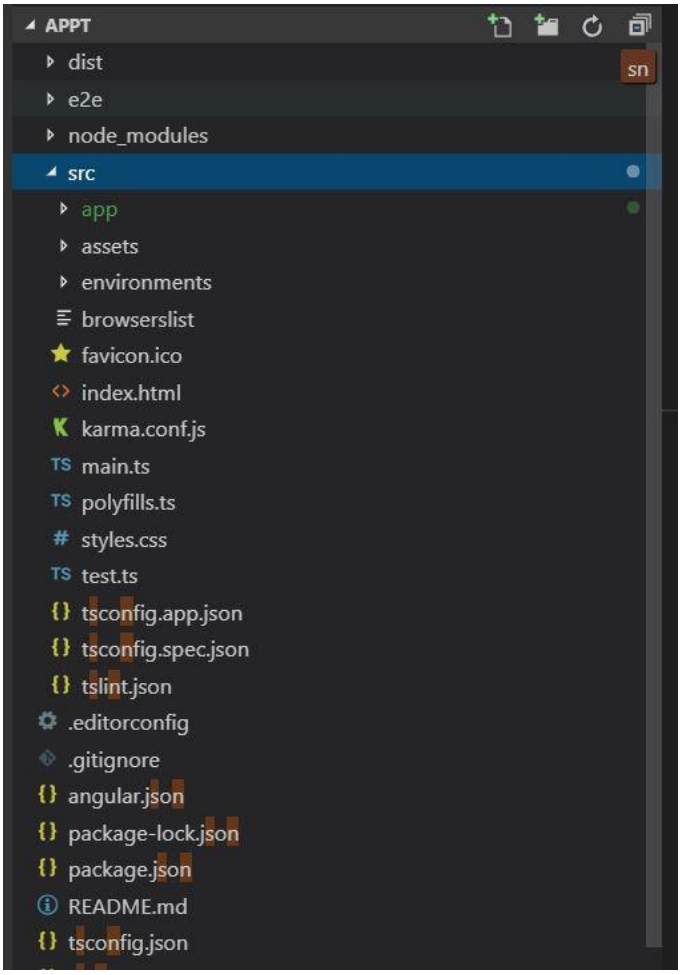


How to change contents of your homepage

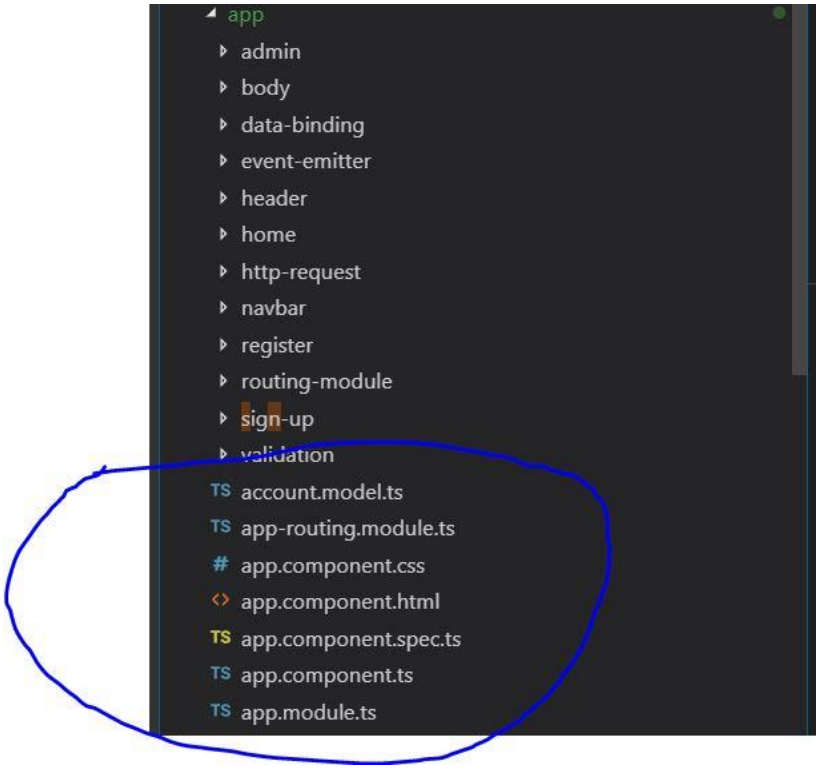
This is how the initial page will look in terms of code

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

```
  <img width="300" alt="Angular Logo"
src="
ciIHZpZXdCb3g9IjAgMCAyNTAgMjUwIj4KICAgIDxwYXRoIGZpbGw9IiNERDAwMzEiIGQ9Ik0xMjUgMzB
MMzEuOSA2My4ybDE0LjIgMTIzLjFMMTI1IDIzMGw3OC45LTQzLjcgMTQuuMi0xMjMuMXoiIC8+CiAgICA8
cGF0aCBmaWxsPSIjQzMwMDJGIiBkPSJNMTI1IDMwbjIyLjItLjFWMjMwbDc4LjktNDMuNyAxNC4yLTEyM
y4xTDEyNSAzMHoiIC8+CiAgICA8cGF0aCAgZmlsbD0iI0ZGRkZGRiIgZD0iTTEyNSA1Mi4xxTDY2LjggMT
gyLjZoMjEuN2wxMS43LTI5LjJoNDkuNGwxMS43IDI5LjJIMTgzTDEyNSA1Mi4xem0xNyA4My4zaC0zNGw
xNy00MC45IDE3IDQwLjl6IiAvPgogIDwvc3ZnPg==">
</div>
<h2>Here are some links to help you start: </h2>
<ul>
  <li>
    <h2><a target="_blank" rel="noopener" href="https://angular.io/tutorial">Tour
of Heroes</a></h2>
  </li>
  <li>
    <h2><a target="_blank" rel="noopener" href="https://angular.io/cli">CLI
Documentation</a></h2>
  </li>
  <li>
    <h2><a target="_blank" rel="noopener" href="https://blog.angular.io/">Angular
blog</a></h2>
  </li>
</ul>

<router-outlet></router-outlet>
```



There is group of files that are the major interaction point when starting development with an angular application as highlighted with blue marker these files are app-component.html, app-component.css , app-components , app-routing-module.ts , everything here is represent the structure style if the first landing page is here.



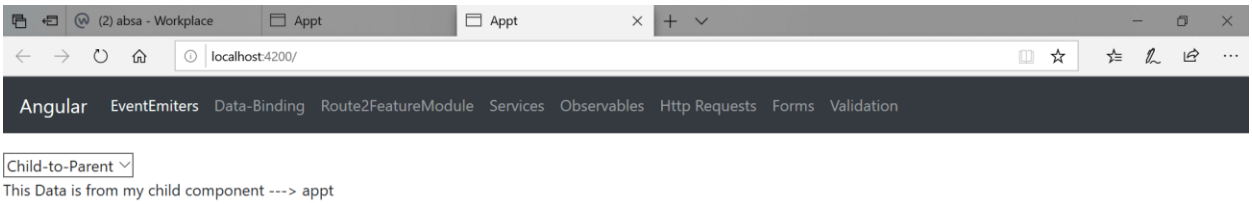
Initially The page will look like this

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>GettingStartedNg5</title>
  <base href="/">

  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body>
  <app-root></app-root>
</body>
</html>
```

We can the change app-component.html to some things like this to make customize it to our liking for example we can use the following code.

```
<app-header></app-header>
<router-outlet></router-outlet>
```

Which will now look something like this.

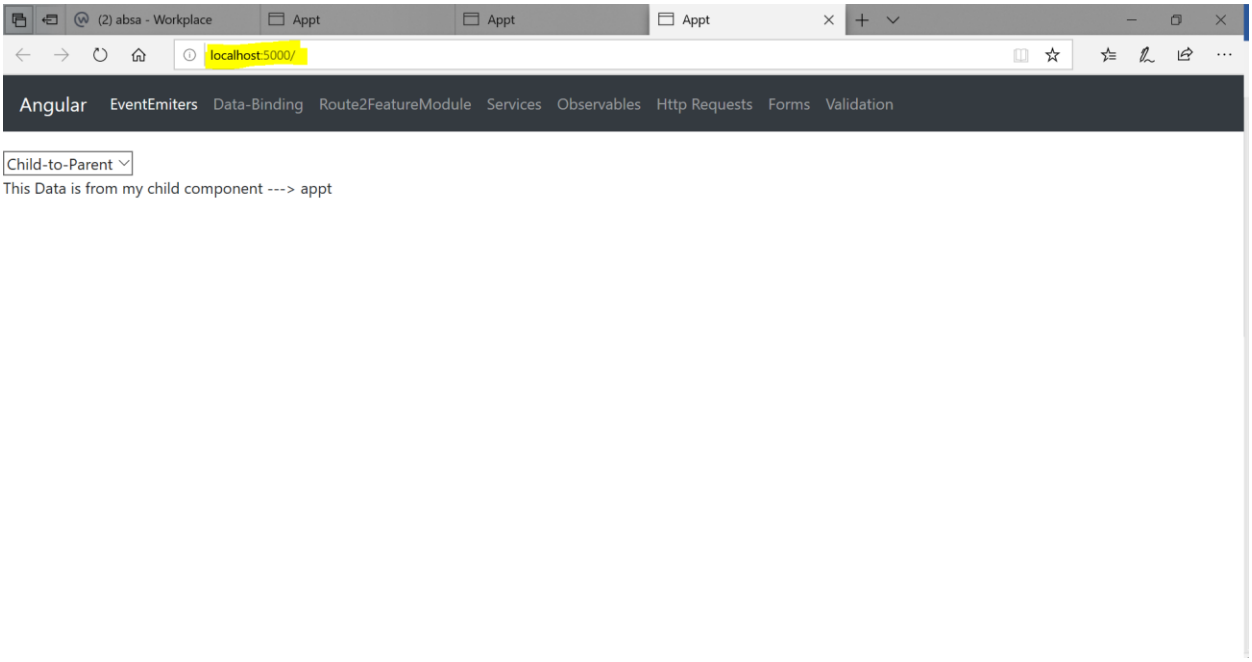


Why angular runs on port 4200 and how you can change it

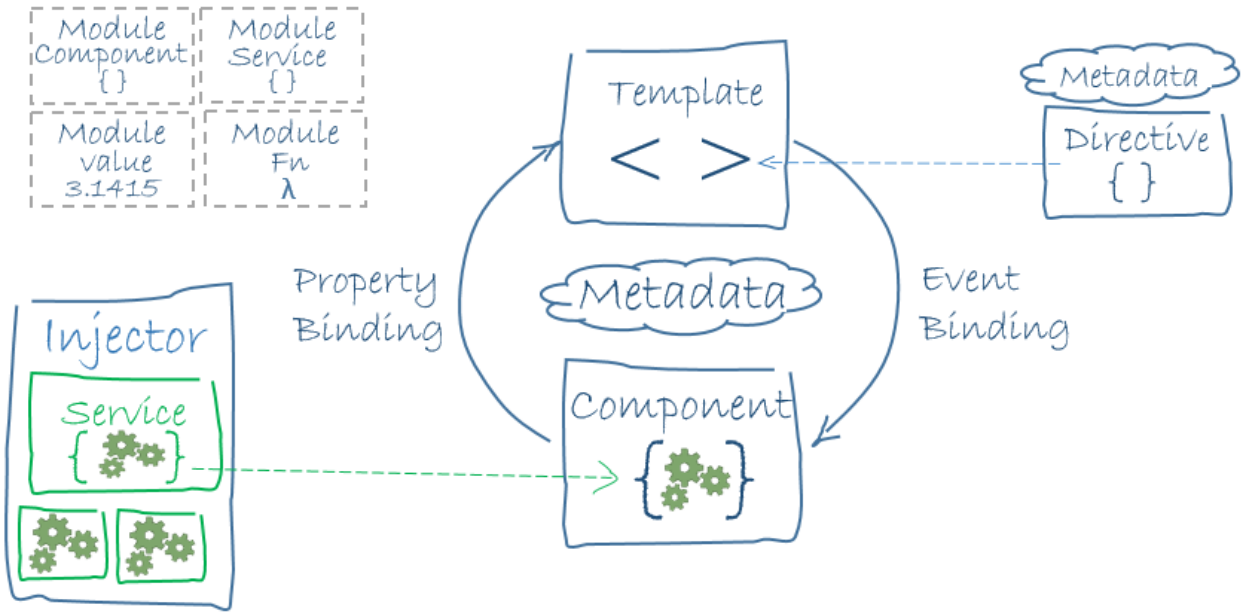
It's not really Agular's port — it's the default port chosen by Angular CLI. The number 4200 serves no particular purpose other than it's usually free, on an average computer. It's not taken by any other common software.

You change the port number by passing the `--port` flag to the serve `command`. For example, the following command will run the development server on the port 5000 instead. E.g

```
ng serve --port 5000
```



10) Architecture of Angular



The building blocks of an Angular application are *Modules* an Angular app is defined by a set of Modules. An app always has at least a *root module* that enables bootstrapping (application start up), and typically has many more *feature modules*.

- Components define *views*
- Components use *services*, which provide specific functionality not directly related to views. Services can be *injected* into components as *dependencies*.

Both components and services are simply classes, with *decorators* that mark their type and provide metadata that tells Angular how to use them.

An app's components typically define many views, arranged hierarchically. Angular provides the router service to help you define navigation paths among views.

11) Components

Components are the basic building block in an Angular application, in actual fact components are just classes nothing special about them until you mark them with the @Component decorator. The main purpose of a component is to control a view.

Components are defined using the `@component` decorator. A component has a `selector`, `template`, `style` and other properties, using which it specifies the metadata required to process the component.

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

  constructor(private linkRouter:Router) { }

  ngOnInit() {
  }


  services(){
    this.linkRouter.navigate(['services']);
  }
```

```
databinding(){
  this.linkRouter.navigate(['data-binding']);
}
```

Selector defines the tag name of your component that is how it will name be called in html.
Template is used to define the view of the component that is the html. If you want to use html from an external file as in our case use templateUrl.

Style defines the relating styling of the component that is the css If you want to use html from an external file as in our case use styleUrl.

HeaderComponent is a simple class, methods that are defined in the class are to be noted

constructor(private linkRouter:Router) is a constructor dependency injection is done here as with the linkRouter object.

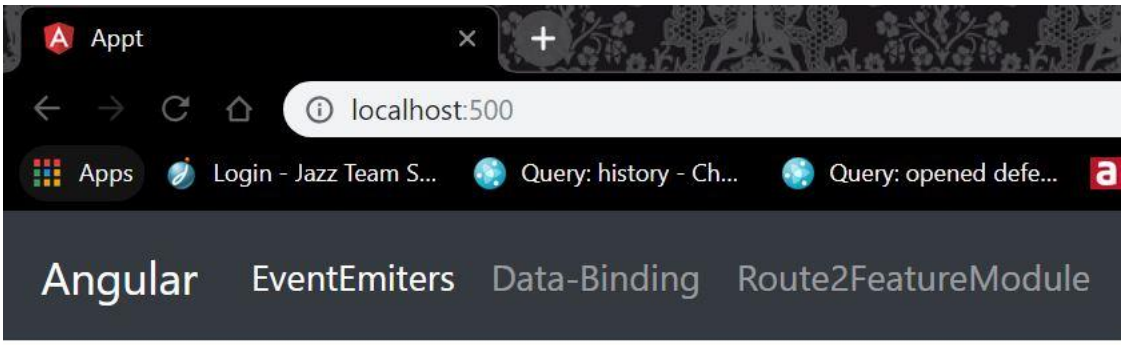
ngOnInit() all initialization and processes that need to be done right after the construction of the component are done here.

services() is a normal void method without parameters or a return type.

12) Event Emitters

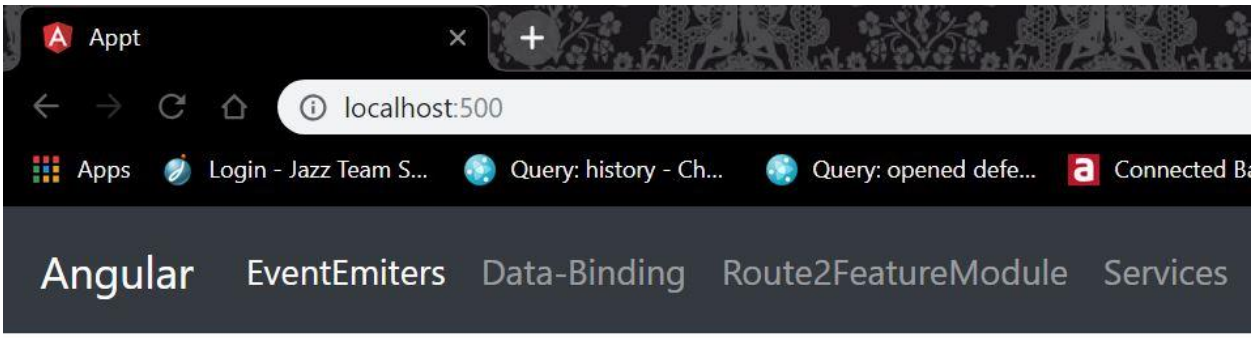
Event emitters simply provide a means for communication between child and parent components. Fundamentally Angular uses data binding for transferring data in/out your component. Property binding is used for data flowing in the component and event binding for data flowing out of your component.

Below is a complete code of event emitters in action, both the html and the TypeScript ,



Parent-to-Child

This data is from my Parent component ---> 1234 Paseka savings



Child-to-Parent

This Data is from my child component ---> hello angular

```
<select [(ngModel)]="selectedValue">
  <option *ngFor="let c of output" [ngValue]="c">{{c.name}}</option>
</select>

<div *ngIf="selectedValue.name == 'Child-to-Parent'">
  <p> This Data is from my child component ---> {{title}}</p>
</div>

<div *ngIf="selectedValue.name == 'Parent-to-Child'">
  <app-navbar  [accountDetails]="account" (outputEvent)="recieve($event)">
  </app-navbar>
</div>
```

```
output : any = [
  ];
  account : Iaccount;
  title = 'appt';
  selectedValue = null;

  constructor() {
    this.account = {
      acc_no : "1234" ,
      name : "Paseka" ,
      type : "savings"
    }

    this.output= [
      {id: 0, name: "Child-to-Parent"},
      {id: 1, name: "Parent-to-Child"}
    ];


    this.selectedValue = this.output[0];

  }

  ngOnInit() {

  }

  recieve(event){
    this.title = event;
  }
```

13)Modules

NgModule is the first basic structure you meet when coding an app with Angular, but it's also the most subtle and complex, because of **different scopes.**

Why NgModule?

The purpose of a NgModule is to declare each thing you create in Angular, and group them together .

There is two kind of main structures:

- **"declarations" is for things you'll use in your templates: mainly components** (~ views: the classes displaying data), but also directives and pipes,
- **"providers" is for services** (~ models: the classes getting and handling data).

Below is example code of an Angular module.

```
import { NgModule } from '@angular/core';
import { SomeComponent } from './some.component';
import { SomeDirective } from './some.directive';
import { SomePipe } from './some.pipe';
import { SomeService } from './shared/some.service';

@NgModule({
  declarations: [SomeComponent, SomeDirective, SomePipe],
  providers: [SomeService]
})

export class SomeModule {}
```

NgModules configure the injector and the compiler and help organize related things together.

An NgModule is a class marked by the @NgModule decorator. @NgModule takes a metadata object that describes how to compile a component's template and how to create an injector at runtime. It identifies the module's own components, directives, and pipes, making some of them public, through the exports property, so that external components can use them. @NgModule can also add service providers to the application dependency injectors

```
@NgModule({
 imports:      [ CommonModule ],
```

```
 declarations: [ CustomerComponent, NewItemDirective, OrdersPipe ],
 exports:      [ CustomerComponent, NewItemDirective, OrdersPipe,
                 CommonModule, FormsModule ]
})
```

What is a *declarable*?

Declarables are the class types—components, directives, and pipes—that you can add to a module's declarations list. They're the only classes that you can add to declarations.

Note the following:

- It imports the CommonModule because the module's component needs common directives.
- It declares and exports the utility pipe, directive, and component classes.
- It re-exports the CommonModule and FormsModule.
- By re-exporting CommonModule and FormsModule, any other module that imports this SharedModule, gets access to directives like NgIf and NgFor from CommonModule and can bind to component properties with [(ngModel)], a directive in the FormsModule.

The confusion starts with **components and services not having the same scope / visibility**:

- declarations / **components are in local scope** (private visibility),
- providers / **services are (generally) in global scope** (public visibility).

It means the **components you declared are only usable in the current module**. If you need to use them outside, in other modules, you'll have to export them:

- **if the module is imported for components, you'll need to import it in each module** needing them,

- **if the module is imported for services, you'll need to import it only once**, in the first app module

The important code that we need to know regarding modules is import, export and declarations.

14) Routing

Is simply a mechanism to navigate from one component (view) to the next, without repetition of code like in normal html? Angular routing provides services, directives and operators that manage the routing and navigation in our application.

When you use normal html use <a href and with routers you use router link.

Navigation comparison (angular vs normal html)

Angular html tag	Normal html tag
<a routerLink>	<a href>

What is Router Outlet

`RouterOutlet` is a directive that is used as `<router-outlet>`. The role of `<router-outlet>` is to mark where the router displays a view. It is placed in your root component.

RouterModule and Routes

`RouterModule` is a separate module in angular that provides required services and directives to use routing and navigation in angular application.

`Routes` defines and array of roots that map a path to a component. Paths are configured in module to make available it globally.

Navigation steps

1. Import RouterModule and Routes :

```
Import {RouterModule, Routes} from '@angular/router';
```

2. Create Array of Routes (map a URL with a component)

```
const routes: Routes = [
    { path: 'goto-mycomponent1', component:  mycomponent1},
    { path: 'goto-mycomponent2', component:  mycomponent2},
    { path: '**', component: mycomponent2}
]
```

The first line of code creates an array of routes, line 2.

a. Mapping a Route to a Component: Find the mapping.

```
{ path: 'manage-book', component: ManageBookComponent }
```

In the above mapping when we access URL **/manage-book** then `ManageBookComponent` will be displayed.

b. Configure Parameters : Find the mapping.

```
{ path: 'update-book/:id', component: UpdateBookComponent }
```

In the above path mapping we need to pass a path parameter, for example if we access the URL **update-book/100** then `UpdateBookComponent` will be displayed.

c. Redirect to a URL : Find the mapping.

```
{ path: '', redirectTo: '/manage-book ', pathMatch: 'full' }
```

If we access a URL without specifying any component path such as **"/"** then it will be redirected to URL **/manage-book** path and hence by default `ManageBookComponent` will be displayed.
d. Handling "Page Not Found" : Find the mapping.

```
{ path: '**', component: PageNotFoundComponent }
```

If we access path that has no mapping with any component, then to handle **404 Not Found** error, we use a path**(**)** that is mapped with any component to show desired message.

3. Using RouterModule.forRoot() :

Now we need to import `RouterModule.forRoot(routes)` using `imports` metadata of `@NgModule`. Here argument `routes` is our constant that we have defined above as array of `Routes` to map path with component.

```
imports: [ RouterModule.forRoot(routes) ]
```

2. Router

It is used to navigate from one component to another component. To use `Router` in any component, follow the steps.

1. Import Router : Import `Router` as follows.

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

3. ActivatedRoute and Params

`Params` is an angular router API that contains the parameter value. To get the parameter value from `Params` we need to pass key.

3. Routing with Parameters : Now suppose a URL **/update-book/100** is being accessed. To understand the fetching of parameter, find the mapping of component which we configure in module.

```
{ path: 'update-book/:id', component: UpdateBookComponent }
```

RouterOutlet

RouterOutlet is a directive that is used as <router-outlet>. The role of <router-outlet> is to mark where the router displays a view. Find the code snippet.

```
<nav [ngClass] = "'menu'">
   <a routerLink="/home" routerLinkActive="active-link">Home</a> |
   <a routerLink="/add-book" routerLinkActive="active-link">Add Book</a> |
   <a routerLink="/manage-book" routerLinkActive="active-link">Manage Book</a>
</nav>
<router-outlet></router-outlet>
```

We have created menu items in the above code using RouterOutlet. They will be shown in every view where we navigate using the route binding with routerLink.

15) Services

Services in Angular allow you to define code that's accessible and reusable throughout multiple components. A common use case for services is when you need to communicate with a backend of some sort to send and receive data.

This is command to Generate a service within a project, in this case the name our service is data

```
ng generate service data
```

Open up the new service file **/src/app/data.service.ts** and let's create the following method

Automatically you will find a class within your file as per your service name, we then create a method in the class as in our case **firstClick().** This method simply prints “clicked”.

```
@Injectable()
```

Notice that the new service imports the Angular `Injectable` symbol and annotates the class with the `@Injectable()` decorator. This marks the class as one that participates in the *dependency injection system*.

The `@Injectable()` decorator accepts a metadata object for the service, one of them being ProvidedIn metadata , which register our provider.

A provider is something that can create or deliver a service . As in our case the `providedIn` metadata value is 'root': When you provide the service at the root level,

Angular creates a single, shared instance of `DataService` and injects into any class that asks for it.

```
import { Injectable } from '@angular/core';
import { Observable, Subject } from 'rxjs';
import {HttpClient} from '@angular/common/http';


@Injectable({
  providedIn: 'root'
})

export class DataService {
  constructor() { }
  firstClick() {
    return console.log('clicked')
   }
  }
```

To use this in a component, visit **/src/app/home/home.component.ts** and update the code to the following:

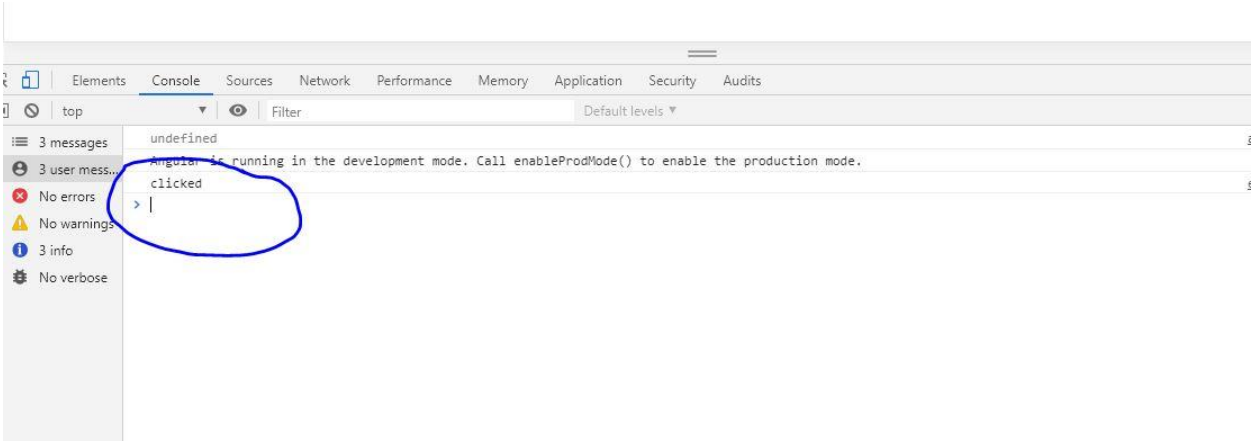
```
import { Component, OnInit } from '@angular/core';
import { DataService } from '../data.service';

@Component({
  selector: 'app-home',
  templateUrl: './home.component.html',
  styleUrls: ['./home.component.scss']
})


export class HomeComponent implements OnInit {

  constructor(private data: DataService) { }

  ngOnInit() {
  }

  firstClick() {
    this.data.firstClick();
  }
}
```

There are 3 things happening here:

- We're first importing the *DataService* at the top.
- We're creating an instance of it through dependency injection within the *constructor()* function.
- Then we call the method with *this.data.firstClick()* when the user clicks on the button.

If you try this, you will see that it works as *clicked* will be printed to the console. Awesome! This means that you now know how to create methods that are accessible from any component in your Angular 7 app.

Compiling and running this in our project we get the following output



16) Forms

Forms are one of the most efficient and widely used means to collect data from the user hence they amount to a large part of any angular application. A more formal definition of forms is:

A web form, also called an HTML form, is an online page that allows for user input. It is an interactive page that mimics a paper document or form, where users fill out particular fields.

Web forms can be rendered in modern browsers using HTML and related web-oriented languages.

In Angular, there are two types of forms namely reactive and template driven forms. Both capture user input events from the view, validate the user input, create a form model and data model to update, and provide a way to track changes. The difference is that both these forms process data differently.

Key Differences Between Reactive and Template Driven Forms

	Reactive	Template
Setup	Created in component class	Created by directives
Data Model	Structured	Unstructured
Predictability	Synchronous	Asynchronous
Validation	Functions	Directives
Scalability	Low-level Access	Abstraction on APIs
Unit Testing	Supports / it is easier	Is a challenge
Data-Binding	One-way	Two-way

Similarities of Reactive and Template Driven forms

- FormControl holds the input value and does validation on fields.
- FormGroup is simply a collection of form controls
- FormArray tracks values and status for an array of form controls.
- ControlValueAccessor creates a bridge between Angular FormControl instances and native DOM elements.

Creation of Reactive Forms

It is better to create a whole new component from scratch that will be where our form developed ,more over this will help making our form reusable in case we want to use it in multiple locations within our application.

```
//FormControl class is the basic building block of reactive forms
//FormGroup class helps to group FormControls
//FormBuilder Service provides convenient methods for generating controls
import { FormBuilder , FormGroup, Validators} from '@angular/forms';
```

FormControl represents an input in the view. The first parameter is the default value and the second is a (or an array of) validator(s) for this field.

FormGroup is composed of a map taking as value an AbstractControl(which means either a FormControl, FormGroup or FormArray). We can nest our FormGroup if we want to create a complexe value. After the map, we can also add other validators at the form level if needed.

Declare and initialize a form group

```
//Declare a form group
profileForm : FormGroup;

//Initializing our formGroup from an instance provided by formBuilder class
this.profileForm = this.formbuilder.group({

  //definition of form control with validators

   name: ['' , Validators.required],
   surname:['' , Validators.required],
   username: ['' , Validators.required],
   email :  ['' , [Validators.required , Validators.email]],
   phone :['' , Validators.required],
   password : ['' , Validators.required],
   address :['' , Validators.required],
});
```

Now that we have created the form, we'll see how to render it. We use the Angular Material Design component to have a nice view, but the important thing is to add the *formGroup* attribute indicating our public form variable *loginForm*:

```
<div class="register-form">
    <form [formGroup]="profileForm"    (ngSubmit)="onSubmit()">

      <h2 class="text-center">I seek validation =(</h2>
      <div class="form-group">
        <input type="text"  formControlName="name" placeholder="First Name">


        <div *ngIf="profileForm.get('name').touched &&
profileForm.get('name').invalid" class="alert alert-danger ">First Name is
required</div>
```

```
      </div>


      <div class="form-group">

        <input type="text"  classs="form-control" formControlName="surname"
placeholder="Last Name">
        <div *ngIf="profileForm.get('surname').touched &&
profileForm.get('surname').invalid" class="alert alert-danger ">Surname is
required</div>
       </div>

    <div>
      <input value = "submit" class="btn btn-success btn-block" type="submit">
     </div>
    </form>
```

When to use template Driven Forms

- Template-driven forms are useful for adding a simple form to an app, such as an email list signup form.
- When you don't need complex validation on your form fields;
- Minimal Coding

When to use reactive Forms

- If forms are a key part of your application, or you're already using reactive patterns for building your application, use reactive forms.
- When you need to use custom validations on your application
- When you need to use forms that can be unit tested

Validations

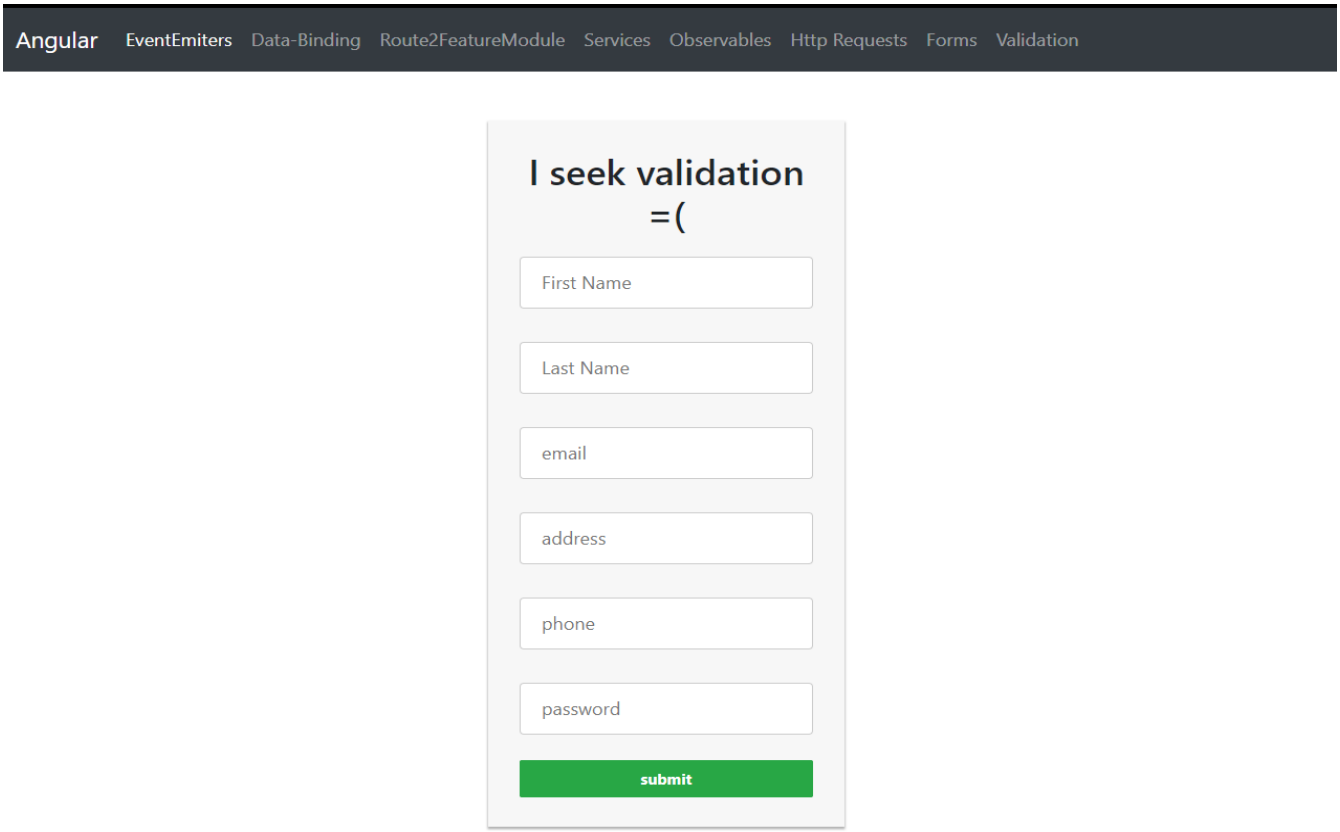
Angular packs a lot of interesting functionality for doing form validation. Validations Improve overall data quality by validating user input for accuracy and completeness. Angular provides the following functionality with regard to validation:

- highlights fields in error dynamically *as we type*
- provide inline messages while the user is typing in a field
- disable the submit button until all the needed data is available and the terms and conditions checkbox is checked

Since there are two approaches to doing Angular Forms i.e Reactive and Template driven approach, there is also two varying ways to do angular validations namely Template driven Validations and Reactive forms validations.

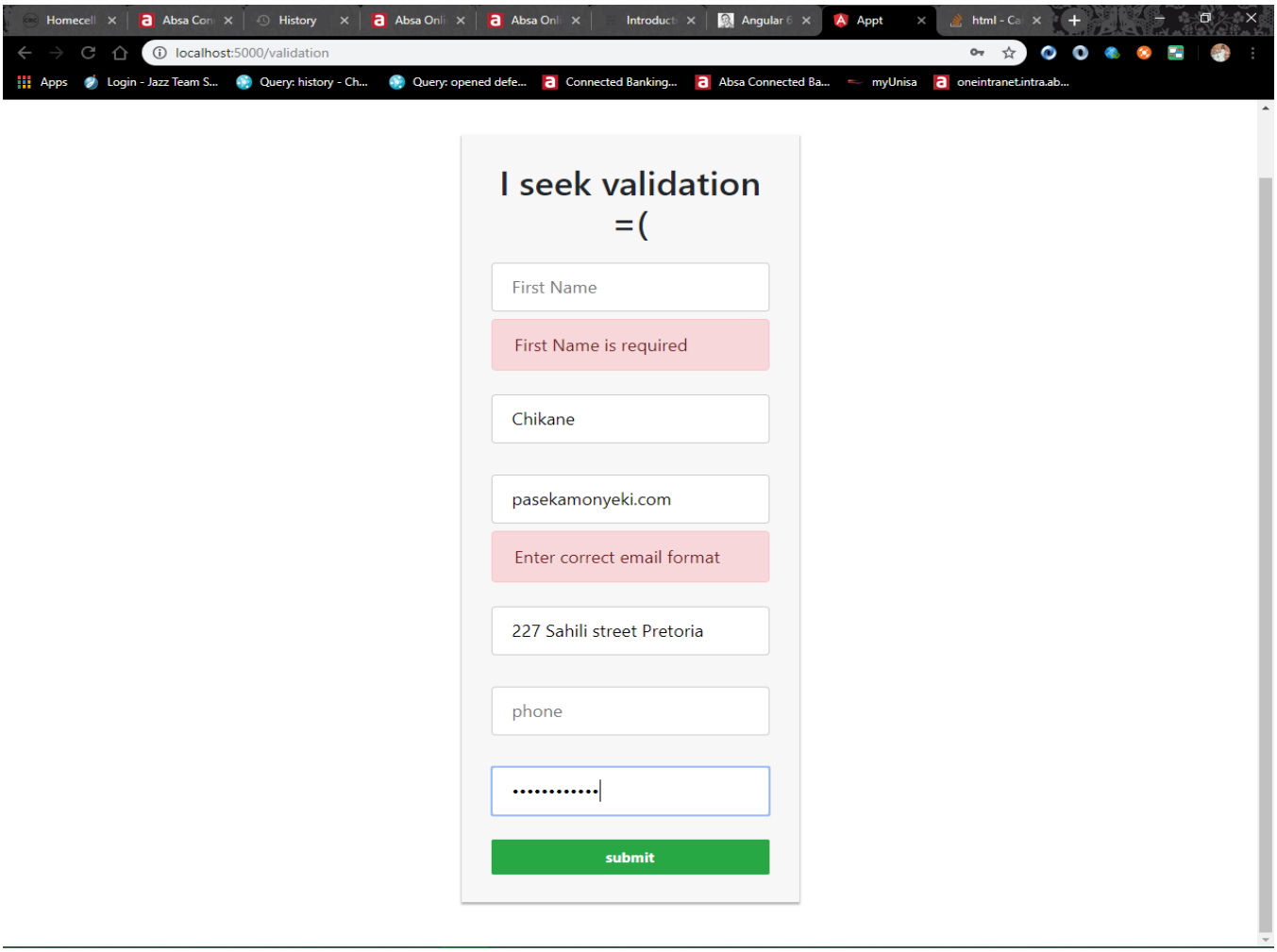
Template driven Validations

Uses only inline validation, which is a feature used within web form (html) that delivers messages to the user based on whether the information entered is correct or incorrect. Here is a simple validation form.



These are error messages being fired on the form, because of incorrect format. e.g for our validations rules we have specified that any form field must not be touched and not be filled and email field must be in the following format [a-z]@[a-z].com. as can be noted on the form the first error message is “first

is required" because we touched the field without filling it , even though the email field is filled , it is not in the correct format , phone did not present a error message because it is not touched.



The html code that produced the following form is as follows. As can be noted from the form. Important things to note are as follows the actual form is within the <form>...</forms> tag, <input> declare a form field, <div *ngif="……."><"Error message" all our validation rules are inside *ngIf="….." . e.g *ngIf="profileForm.get('surname').touched && profileForm.get('surname').invalid" specifies that whenever the field is touched and the value entered is invalid(empty text , numbers instead of text) then error message should appear.

```
<div class="register-form">
<form [formGroup]="profileForm"    (ngSubmit)="onSubmit()">

  <h2 class="text-center">I seek validation =(</h2>
  <div class="form-group">
    <input type="text"  formControlName="name" placeholder="First Name">


    <div *ngIf="profileForm.get('name').touched && profileForm.get('name').invalid" class="alert alert-danger ">First Name is required</div>

  </div>


  <div class="form-group">

    <input type="text"  classs="form-control" formControlName="surname" placeholder="Last Name">
    <div *ngIf="profileForm.get('surname').touched && profileForm.get('surname').invalid" class="alert alert-danger ">Surname is required</div>
   </div>


   <div class="form-group">
      <input type="email" classs="form-control" formControlName="email" placeholder="email">
      <div *ngIf="profileForm.get('email').invalid &&  profileForm.get('surname').touched" class="alert alert-danger "> Enter correct email format</div>
   </div>


<div class="form-group">
      <input type="text"classs="form-control" formControlName="address" placeholder="address">
      <div *ngIf="profileForm.get('address').touched && profileForm.get('address').invalid" class="alert alert-danger ">Address is required</div>
</div>

<div class="form-group">
      <input type="text"classs="form-control" formControlName="phone" placeholder="phone">
      <div *ngIf="profileForm.get('phone').touched && profileForm.get('phone').invalid" class="alert alert-danger ">Phone is required</div>
</div>

<div class="form-group">
      <input type="password" classs="form-control" formControlName="password" placeholder="password">
      <div *ngIf="profileForm.get('password').touched && profileForm.get('password').invalid" class="alert alert-danger ">password is required</div>
</div>
<div>
  <input value = "submit" class="btn btn-success btn-block" type="submit">
 </div>
</form>
```

Reactive forms validations

Validations of reactive forms are much more powerful because by design they use a much more powerful approach to handling data, which is use of a form controls and form groups (a group of form controls), form-control is where the magic of validation also happens.

1) Import the following packages from angular forms module

```
2) //FormControl class is the basic building block of reactive forms
//FormGroup class helps to group FormControls
//FormBuilder Service provides convenient methods for generating controls
import { FormBuilder , FormGroup, Validators} from '@angular/forms';
```

3) Define a new form group

```
//Declare a form group
profileForm : FormGroup;
```

4) Instantiate a new form group with form controls and relating validations

```
   this.profileForm = this.formbuilder.group({

    //definition of form control with validators

      name: ['' , Validators.required],
      surname:['' , Validators.required],
      username: ['' , Validators.required],
      email :  ['' , [Validators.required , Validators.email]],
      phone :['' , Validators.required],
      password : ['' , Validators.required],
      address :['' , Validators.required],
});
```

5) If you want to go a step further and implement custom validations this way