Developing Rich Internet Applications using Angular

seed®

beyond the obvious

Developing Rich Internet Applications using Angular

N·S·D·C
National
Skill Development
Corporation
Transforming the skill landscape
Training Partner

seed®
beyond the obvious

Compiled by SEED Infotech Ltd.

Index

Introduction to Single Page Application Framework - Angular4

This chapter includes

1) Introduction to SPA
2) Introduction to Typescript
3) Introduction to Angular4

Part-I Angular, Typescript

What are Single Page Applications (SPA)

SPA is essentially an evolution of the MPA+AJAX design pattern, where the only shell page is generated on the server and

All UI is rendered by browser JavaScript code.

SPA requests the markup and data separately, and renders pages directly in browser.

By using JavaScript frameworks like AngularJS /Angular and other framework such as KnockoutJS.

Popular JavaScript Frameworks for Building SPAs

4) Angular
5) React
6) Ember
7) Aurelia
8) Vue.js
9) Cycle.js
10) Backbone

Introduction to Angular as SPA Framework

Angular is a framework for building client applications in HTML and either JavaScript or a language like TypeScript that compiles to JavaScript. It is a framework that makes it easy to build applications with the web. It combines declarative templates, dependency injection, end to end tooling, and integrated best practices to solve development challenges. It is typically used by developers to build RIA applications for web, mobile, or the desktop.

Angular is a framework for building Rich internet Application or SPA applications. The framework consists of several libraries, some of them core and some optional. Angular ships as a collection of JavaScript modules. Each Angular library name begins with the @angular prefix. When required they are imported using JavaScript import statements. For instance

- To create components, import Component module to work with decorator from the @angular/core library as below.

  import { Component } from '@angular/core';
- @Component decorator, which identifies the class to which it is applied as a component class.

What is Node.js?

Node.js

Node.js is an open source server framework and it is free. It runs on various platforms (Windows,

Linux, Unix, Mac OS X, etc.). It uses JavaScript on the server. It can generate dynamic page content.

NPM- Node package Manager

NPM is a package manager for Node.js packages, or modules..

The NPM program is by default installed on the computer when Node.js is installed.

A package in Node.js contains all the files for a module.

Visual Studio Code for Windows

Download the Visual Studio Code installer for Windows.

(https://go.microsoft.com/fwlink/?LinkID=534107) Once it is downloaded, run the installer (VSCodeSetup-version.exe). By default, VS Code is installed under C:\Program Files (x86)\Microsoft VS Code for a 64-bit machine. Visual Studio Code is a source code editor developed by Microsoft for Windows, Linux and macOS. VS Code has built-in debugging support for the Node.js runtime and can debug JavaScript, TypeScript, and any other language that gets transpiled to JavaScript.

AngularCLI

Angular CLI is a Command Line Interface (CLI) to automate development tasks. It is a tool to initialize, develop, scaffold and maintain Angular applications. It allows to:

- create a new Angular application
- helps to preview application during development.
- run unit tests
- run application’s end-to-end (E2E) tests
- build application for deployment to production.

Scaffold	Usage
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

Introduction to Typescript

TypeScript is a free and open-source programming language developed and maintained by Microsoft. It is a strict syntactical superset of JavaScript, and adds optional static typing to the language. Anders Hejlsberg, lead architect of C# and creator of Delphi and Turbo Pascal, has worked on the development of TypeScript. TypeScript may be used to develop JavaScript applications for client-side or server-side (Node.js) execution.

TypeScript is designed for development of large applications and compiles to JavaScript. As TypeScript is a superset of JavaScript, existing JavaScript programs are also valid TypeScript programs.

By definition, "TypeScript is JavaScript for application-scale development."

TypeScript is a strongly typed, object oriented, compiled language. It was designed by **Anders Hejlsberg** (designer of C#) at Microsoft. TypeScript is both a language and a set of tools. TypeScript is a typed superset of JavaScript compiled to JavaScript. In other words, TypeScript is JavaScript plus some additional features.

TypeScript is a strict superset of ECMAScript 2015, which is itself a superset of ECMAScript 5, commonly referred to as JavaScript.

Programming in TypeScript

Boolean

The let keyword is used to define variable . The most basic datatype is the simple true/false value, which JavaScript and TypeScript call a boolean value.

let isDone: boolean = false;

Number

As in JavaScript, all numbers in TypeScript are floating point values. These floating point numbers get the type number. In addition to hexadecimal and decimal literals, TypeScript also supports binary and octal literals introduced in ECMAScript 2015.

- let decimal: number = 6;

- let hex: number = 0xf00d;
- let binary: number = 0b1010;
- let octal: number = 0o744;

String

Another fundamental part of creating programs in JavaScript for webpages and servers alike is working with textual data. As in other languages, we use the type string to refer to these textual datatypes. Just like JavaScript, TypeScript also uses double quotes (") or single quotes (') to surround string data.

let color: string = "blue";

color = 'red';

You can also use template strings, which can span multiple lines and have embedded expressions. These strings are surrounded by the backtick/backquote (`) character, and embedded expressions are of the form ${ expr }.

let fullName: string = `Bob Bobbington`;

let age: number = 37;

let sentence: string = `Hello, my name is ${ fullName }.

I'll be ${ age + 1 } years old next month.`;

This is equivalent to declaring sentence like so:

let sentence: string = "Hello, my name is " + fullName + ".\n\n" +

"I'll be " + (age + 1) + " years old next month.";

Array

TypeScript, like JavaScript, allows you to work with arrays of values. Array types can be written in one of two ways. In the first, you use the type of the elements followed by [] to denote an array of that element type:

let list: number[] = [1, 2, 3];

The second way uses a generic array type, Array<elemType>:

let list: Array<number> = [1, 2, 3];

Enums in TypeScript

num Color {Red, Green, Blue}

let c: Color = Color.Green;

By default, enums begin numbering their members starting at 0. You can change this by manually setting the value of one of its members. For example, we can start the previous example at 1 instead of 0:

enum Color {Red = 1, Green, Blue}

let c: Color = Color.Green;

Or, even manually set all the values in the enum:

enum Color {Red = 1, Green = 2, Blue = 4}

let c: Color = Color.Green;

Use of Any

We may need to describe the type of variables that we do not know when we are writing an application. These values may come from dynamic content, e.g. from the user or a 3rd party library. In these cases, we want to opt-out of type-checking and let the values pass through compile-time checks. To do so, we label these with the any type:

let notSure: any = 4;

notSure = "maybe a string instead";

notSure = false; // okay, definitely a Boolean

var declarations

Declaring a variable in JavaScript has always traditionally been done with the var keyword.

var a = 10;

As you might’ve figured out, we just declared a variable named a with the value 10.

We can also declare a variable inside of a function.

Defining Functions

```
function f() {
    var message = "Hello, world!";

    return message;
}
// Named function
function add(x, y) {
    return x + y;
}
// Anonymous function
let myAdd = function(x, y) { return x + y; };
```

Typing the function

Let’s add types to our simple examples from earlier:

```
function add(x: number, y: number): number {
    return x + y;
}
```

let myAdd = function(x: number, y: number): number { return x + y; };

We can add types to each of the parameters and then to the function itself to add a return type. TypeScript can figure the return type out by looking at the return statements, so we can also optionally leave this off in many cases.

Inferring the types

In playing with the example, you may notice that the TypeScript compiler can figure out the type if you have types on one side of the equation but not the other:

// myAdd has the full function type

let myAdd = function(x: number, y: number): number { return x + y; };

Optional and Default Parameters

In TypeScript, every parameter is assumed to be required by the function. This doesn’t mean that it can’t be given null or undefined, but rather, when the function is called the compiler will check that the user has provided a value for each parameter. 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.

```
function buildName(firstName: string, lastName:
string) {
    return firstName + " " + lastName;
}
```

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

let result2 = buildName("Bob", "Adams", "Sr."); // error, too many parameters

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

In JavaScript, every parameter is optional, and users may leave them off as they see fit. When they do, their value is undefined. 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 buildName(firstName: string,
lastName?: string) {
    if (lastName)
        return firstName + " " + lastName;
    else
        return firstName;
}
```

let result1 = buildName("Bob"); // works correctly now

let result2 = buildName("Bob", "Adams", "Sr."); // error, too many parameters

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

Any optional parameters must follow required parameters. Had we wanted to make the first name optional rather than the last name, we would need to change the order of parameters in the function, putting the first name last in the list.

In TypeScript, we can also set a value that a parameter will be assigned if the user does not provide one, or if the user passes undefined in its place. These are called default-initialized parameters. Let’s take the previous example and default the last name to "Smith".

```
function buildName(firstName: string, lastName
= "Smith") {
    return firstName + " " + lastName;
}
```

let result1 = buildName("Bob"); // works correctly now, returns "Bob Smith"

let result2 = buildName("Bob", undefined); // still works, also returns "Bob Smith"

let result3 = buildName("Bob", "Adams", "Sr."); // error, too many parameters

let result4 = buildName("Bob", "Adams"); // ah, just right

Optional and Default Parameters

In TypeScript, every parameter is assumed to be required by the function. This doesn’t mean that it can’t be given null or undefined, but rather, when the function is called the compiler will check that the user has provided a value for each parameter. 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.

```
function buildName(firstName: string, lastName:
string) {
    return firstName + " " + lastName;
}
```

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

let result2 = buildName("Bob", "Adams", "Sr."); // error, too many parameters

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

In JavaScript, every parameter is optional, and users may leave them off as they see fit. When they do, their value is undefined. 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 buildName(firstName: string,
lastName?: string) {
    if (lastName)
        return firstName + " " + lastName;
    else
        return firstName;
}
```

let result1 = buildName("Bob"); // works correctly now

let result2 = buildName("Bob", "Adams", "Sr."); // error, too many parameters

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

Any optional parameters must follow required parameters. Had we wanted to make the first name optional rather than the last name, we would need to change the order of parameters in the function, putting the first name last in the list.

In TypeScript, we can also set a value that a parameter will be assigned if the user does not provide one, or if the user passes undefined in its place. These are called default-initialized parameters. Let’s take the previous example and default the last name to "Smith".

```
function buildName(firstName: string, lastName
= "Smith") {
    return firstName + " " + lastName;
}
```

let result1 = buildName("Bob"); // works correctly now, returns "Bob Smith"

let result2 = buildName("Bob", undefined); // still works, also returns "Bob Smith"

let result3 = buildName("Bob", "Adams", "Sr."); // error, too many parameters

let result4 = buildName("Bob", "Adams"); // ah, just right

Unlike plain optional parameters, default-initialized parameters don’t need to occur after required parameters. If a default-initialized parameter comes before a required parameter, users need to explicitly pass undefined to get the default initialized value. For example, we could write our last example with only a default initializer on firstName:

```
function buildName(firstName = "Will",
lastName: string) {
    return firstName + " " + lastName;
}
```

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

let result2 = buildName("Bob", "Adams", "Sr."); // error, too many parameters

let result3 = buildName("Bob", "Adams"); // okay and returns "Bob Adams"

let result4 = buildName(undefined, "Adams"); // okay and returns "Will Adams"

TypeScript Programming Features- Control Structures

```
// Nested if else
var num:number = 4;
if (num >0 )
 {
    console.log("Ohh its positive number");
} else if (num < 0 )
{
   console.log("Ohh its negative number"); }
else
 {
     console.log("Ohh its actually zero");
 }
```

//output - Ohh its positive number

//Logical Operators And-&& Or- || & Not - !

TypeScript Programming Features- Loops

```
var num:number = 5;
while(num >=1) {
   num-=1;
   console.log("Num is  "+ num);
   num--;
}
//output is  4 2 0
var n:number = 5
do {
    n++;
   console.log(n);
} while (n <= 10);

var num:number = 4;
var s:number;
var factorial = 1;
for( s = num;s>=1;s--) {
   factorial *= s;
}
console.log(" Factorial -" + factorial)
//Output - Factorial -24
```

Iterators

An object is deemed iterable if it has an implementation for the Symbol.iterator property. Some built-in types like Array, Map, Set, String, Int32Array, Uint32Array, etc. have their Symbol.iterator property already implemented. Symbol.iterator function on an object is responsible for returning the list of values to iterate on.

a) for..of statements

for..of loops over an iterable object, invoking the Symbol.iterator property on the object. Here is a simple for..of loop on an array:

```
let someArray = [1, "string", false];
for (let entry of someArray) {
    console.log(entry); // 1, "string", 
false
}
```

b) for..of vs. for..in statements

Both for..of and for..in statements iterate over lists; the values iterated on are different though, for..in returns a list of keys on the object being iterated, whereas for..of returns a list of values of the numeric properties of the object being iterated.

Here is an example that demonstrates this distinction:

```
let list = [4, 5, 6];
for (let i in list) {
   console.log(i); // "0", "1", "2",
}
for (let i of list) {
   console.log(i); // "4", "5", "6"
}
```

Another distinction is that for..in operates on any object; it serves as a way to inspect properties on this object. for..of on the other hand, is mainly interested in values of iterable objects. Built-in objects like Map and Set implement Symbol.iterator property allowing access to stored values.

```
let pets = new Set(["Cat", "Dog", "Hamster"]);
pets["species"] = "mammals";
for (let pet in pets) {
   console.log(pet); // "species"
}
for (let pet of pets) {
    console.log(pet); // "Cat", "Dog",
"Hamster"
}
```

Classes in TypeScript

```
class Greeter {
    greeting: string;
    constructor(message: string) {
        this.greeting = message;
    }
    greet() {
        return "Hello, " + this.greeting;
    }
}
let greeter = new Greeter("world");
```

Inheritance

In TypeScript, we can use common object-oriented patterns. One of the most fundamental patterns in class-based programming is being able to extend existing classes to create new ones using inheritance.

Let's take a look at an example:

```
class Animal {
    move(distanceInMeters: number = 0) {
        console.log(`Animal moved
${distanceInMeters}m.`);
    }
}
class Dog extends Animal {
    bark() {
        console.log('Woof! Woof!');    }
}
const dog = new Dog();
dog.bark();
dog.move(10);
dog.bark();
```

This example show the most basic inheritance feature: classes inherit properties and methods from base classes. Here, Dog is a derived class that derives from the Animal base class using the extends keyword. Derived classes are often called subclasses, and base classes are often called superclasses.

Because Dog extends the functionality from Animal, we were able to create an instance of Dog that could both bark() and move().

Let’s now look at a more complex example.

```
class Animal {
    name: string;
    constructor(theName: string) { this.name =
theName; }
    move(distanceInMeters: number = 0) {
        console.log(`${this.name} moved
${distanceInMeters}m.`);
    }
}

class Snake extends Animal {
    constructor(name: string) { super(name); }
    move(distanceInMeters = 5) {
        console.log("Slithering...");
        super.move(distanceInMeters);
    }
}

class Horse extends Animal {
    constructor(name: string) { super(name); }
    move(distanceInMeters = 45) {
        console.log("Galloping...");
        super.move(distanceInMeters);
    }
}
let sam = new Snake("Sammy the Python");
let tom: Animal = new Horse("Tommy the
Palomino");
sam.move();
tom.move(34);
```

This example covers a few other features we didn't previously mention. Again, we see the extends keywords used to create two new subclasses of Animal: Horse and Snake.

One difference from the prior example is that each derived class that contains a constructor function must call super() which will execute the constructor of the base class. What's more, before we ever access a property on this in a constructor body, we have to call super(). This is an important rule that TypeScript will enforce.

The example also shows how to override methods in the base class with methods that are specialized for the subclass. Here both Snake and Horse create a move method that overrides the move from Animal, giving it functionality specific to each class. Note that even though tom is declared as an Animal, since its value is a Horse, calling tom.move(34) will call the overriding method in Horse:

```
Slithering...
Sammy the Python moved 5m.
Galloping...
Tommy the Palomino moved 34m.
```

Public, private, and protected modifiers

Public by default

In our examples, we've been able to freely access the members that we declared throughout our programs. If you're familiar with classes in other languages, you may have noticed in the above examples we haven't had to use the word public to accomplish this; for instance, C# requires that each member be explicitly labeled public to be visible. In TypeScript, each member is public by default.

You may still mark a member public explicitly. We could have written the Animal class from the previous section in the following way:

a) Public :

```
class Animal {
    public name: string;
    public constructor(theName: string) {
this.name = theName; }
    public move(distanceInMeters: number) {
        console.log(`${this.name} moved
${distanceInMeters}m.`);
    }
}
```

b) Understanding private

When a member is marked private, it cannot be accessed from outside of its containing class. For example:

```
class Animal {
    private name: string;
    constructor(theName: string) { this.name
= theName; }
}
new Animal("Cat").name; // Error: 'name' is
private;
```

TypeScript is a structural type system. When we compare two different types, regardless of where they came from, if the types of all members are compatible, then we say the types themselves are compatible.

However, when comparing types that have private and protected members, we treat these types differently. For two types to be considered compatible, if one of them has a private member, then the other must have a private member that originated in the same declaration. The same applies to protected members.Let’s look at an example to better see how this plays out in practice:

```
class Animal {
    private name: string;
    constructor(theName: string) { this.name
= theName; }
}
class Rhino extends Animal {
    constructor() { super("Rhino"); }
}
class Employee {
    private name: string;
    constructor(theName: string) { this.name
= theName; }
}
let animal = new Animal("Goat");
let rhino = new Rhino();
let employee = new Employee("Bob");

animal = rhino;
animal = employee; // Error: 'Animal' and
'Employee' are not compatible
```

In this example, we have an Animal and a Rhino, with Rhino being a subclass of Animal. We also have a new class Employee that looks identical to Animal in terms of shape. We create some instances of these classes and then try to assign them to each other to see what will happen. Because Animal and Rhino share the private side of their shape from the same declaration of private name: string in Animal, they are compatible. However, this is not the case for Employee. When we try to assign from an Employee to Animal we get an error that these types are not compatible. Even though Employee also has a private member called name, it’s not the one we declared in Animal.

a) Understanding protected

The protected modifier acts much like the private modifier with the exception that members declared protected can also be accessed by instances of deriving classes. For example,

```
class Person {
    protected name: string;
    constructor(name: string) { this.name =
name; }
}

class Employee extends Person {
    private department: string;
    constructor(name: string, department:
string) {
        super(name);
        this.department = department;
    }

    public getElevatorPitch() {
        return `Hello, my name is
${this.name} and I work in
${this.department}.`;
    }
}
```

```
let howard = new Employee("Howard", "Sales");
console.log(howard.getElevatorPitch());
console.log(howard.name); // error
```

Notice that while we can’t use name from outside of Person, we can still use it from within an instance method of Employee because Employee derives from Person.

A constructor may also be marked protected. This means that the class cannot be instantiated outside of its containing class, but can be extended. For example,

```
class Person {
    protected name: string;
    protected constructor(theName: string) {
this.name = theName; }
}

// Employee can extend Person
class Employee extends Person {
    private department: string;

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

    public getElevatorPitch() {
        return `Hello, my name is
${this.name} and I work in
${this.department}.`;
    }
}

let howard = new Employee("Howard", "Sales");
let john = new Person("John"); // Error: The
'Person' constructor is protected
Readonly modifier
You can make properties readonly by using the
readonly keyword. Readonly properties must be
initialized at their declaration or in the
constructor.
class Octopus {
    readonly name: string;
    readonly numberOfLegs: number = 8;
    constructor (theName: string) {
        this.name = theName;
    }
}
let dad = new Octopus("Man with the 8 strong
legs");
dad.name = "Man with the 3-piece suit"; //
error! name is readonly.
```

Accessors

TypeScript supports getters/setters as a way of intercepting accesses to a member of an object. This gives you a way of having finer-grained control over how a member is accessed on each object.

Let’s convert a simple class to use get and set. First, let’s start with an example without getters and setters.

```
class Employee {
    fullName: string;
}
let employee = new Employee();
employee.fullName = "Bob Smith";
```

```
if (employee.fullName) {
    console.log(employee.fullName);
}
let passcode = "secret passcode";

class Employee {
    private _fullName: string;

    get fullName(): string {
        return this._fullName;
    }

    set fullName(newName: string) {
        if (passcode && passcode == "secret
passcode") {
            this._fullName = newName;
        }
        else {
            console.log("Error: Unauthorized
update of employee!");
        }
    }
}
let employee = new Employee();
employee.fullName = "Bob Smith";
if (employee.fullName) {
    console.log(employee.fullName);
}
```

Interfaces

```
class Point {
    x: number;
    y: number;
}
interface Point3d extends Point {
    z: number;
}
let point3d: Point3d = {x: 1, y: 2, z: 3};
```

```
Generics
function identity<T>(arg: T): T {
    return arg;
}
```

We've now added a type variable T to the identity function. This T allows us to capture the type the user provides (e.g. number), so that we can use that information later. Here, we use T again as the return type. On inspection, we can now see the same type is used for the argument and the return type. This allows us to traffic that type information in one side of the function and out the other.

Once we've written the generic identity function, we can call it in one of two ways. The first way is to pass all of the arguments, including the type argument, to the function:

```
let output = identity<string>("myString");  //
type of output will be 'string'
class GenericNumber<T> {
    zeroValue: T;
    add: (x: T, y: T) => T;
}
```

```
let myGenericNumber = new
GenericNumber<number>();
myGenericNumber.zeroValue = 0;
myGenericNumber.add = function(x, y) { return x
+ y; };
This is a pretty literal use of the
GenericNumber class, but you may have noticed
that nothing is restricting it to only use the
number type. We could have instead used string
or even more complex objects.
let stringNumeric = new GenericNumber<string>();
stringNumeric.zeroValue = "";
stringNumeric.add = function(x, y) { return x +
y;
};alert(stringNumeric.add(stringNumeric.zeroValu
e, "test"));
```

Part- III : What is Angular4?

Angular4 or Angular is a framework for building client applications in HTML and either JavaScript or a language like TypeScript that compiles to JavaScript.

Angular is a TypeScript-based open-source front-end web application platform led by the Angular Team at Google and by a community of individuals and corporations. Angular is a complete rewrite from the same team that built AngularJS.

Differences between Angular and AngularJS

- Angular does not have a concept of "scope" or controllers, instead it uses a hierarchy of components as its main architectural concept
- Angular has a different expression syntax, focusing on "[ ]" for property binding, and "( )" for event binding
- Modularity – much core functionality has moved to modules
- Angular recommends the use of Microsoft's TypeScript language, which introduces the following features:

✦ Class-based Object Oriented Programming

✦ Generics

- TypeScript is a superset of ECMAScript 6 (ES6), and is backwards compatible with ECMAScript 5 (i.e.: JavaScript). Angular also includes ES6:

✦ Lambdas

✦ Iterators

✦ For/Of loops

- Replacing controllers and $scope with components and directives – a component is a directive with a template
- Support asynchronous programming using RxJS

Angular4 Application - Development Environment

- Working with Angular application requires

✦ Node.js – Node.js is a JavaScript runtime built on Chrome's V8 . It runs JS on the server. JavaScript engine.

❑ Download it from https://nodejs.org.

❑ It helps to run JavaScript outside the browser. NodeJS uses the chrome JavaScript

- Npm
  - ❑ Tool that packages Angular application. By default installed when Node.js is installed.
- TypeScript
  - ❑ A language for coding Angular application.
- AngularCLI – A command line to develop, manage , debug, run and test Angular applications
- IDE – Visual Studio Code is a code editor.
- Download it https://go.microsoft.com/fwlink/?LinkID=534107 and
- install it.

A quick look at features of Angular4

The newly introduced features of Angularr in the release4.0 are as below.

1) Typescript support
   - Angular v4.0 is compatible with newer versions TypeScript 2.1 and TypeScript 2.2.
   - This helps with better type checking and also enhanced IDE features for Visual Studio Code.
2) View engine with less code
   - The view engine is introduced in Angular 4 where the produced code of components can be reduced up to 60%.
   - The bundles are reduced to thousands of KBs.
3) Templates
   - template is now ng-template
4) ngIf with else
   - It’s now also possible to use an else syntax in the templates:

```
<div *ngIf="races.length > 0; else empty">
<h2> Now the condition is true</h2>
</div>
<ng-template  #empty>
<h2> Now the condition is false </h2>
</ng-template>
```

5) Pipes

- Angular 4 introduced a new titlecase pipe.
- It changes the first letter of each word into uppercase:
- Example :
    - <p>{{ ‘SARA TEENS' | titlecase }}</p>
    - <!-- will display ‘Sara Teens' -->

Note : These features will be covered in details in the further chapters

Angular Components and Modules

What are modules?

Every angular applications comprises of components and modules. In Angular, a module offers a way to group components, directives, pipes and services that are related. The NgModule - a class decorated with @NgModule - is a fundamental feature of Angular. Every Angular app has at least one NgModule class, the root module, conventionally named AppModule. By convention, the root module class is called AppModule . It exists in a file named app.module.ts. To define modules NgModule decorator must be used.The @NgModule decorator defines the metadata for the module.This is typically a class that has been decorated with @NgModule.

While the root module may be the only module in a small application, most apps have many more feature modules, each a cohesive block of code dedicated to an application domain, a workflow, or a closely related set of capabilities.An NgModule, whether a root or feature, is a class with an @NgModule decorator.

What is the use of NgModule decorator?

NgModule is a decorator function that takes a single metadata object whose properties describe the module. The most important properties are:

1) declarations - the view classes that belong to this module. Angular has three kinds of view classes: components, directives, and pipes.
2) exports - the subset of declarations that should be visible and usable in the component templates of other modules.
3) imports - other modules whose exported classes are needed by component templates declared in this module.
4) providers - creators of services that this module contributes to the global collection of services; they become accessible in all parts of the app.
5) bootstrap - the main application view, called the root component, that hosts all other app views. Only the root

module should set this bootstrap property.

The example code for the root module is as shown below

src/app/app.module.ts

```
content_copyimport { NgModule }      from
'@angular/core';
import { BrowserModule } from
'@angular/platform-browser';
@NgModule({
  imports:      [ BrowserModule ],
  providers:    [ Logger ],
  declarations: [ AppComponent ],
  exports:      [ AppComponent ],
  bootstrap:    [ AppComponent ]
})
export class AppModule { }
```

Launch an application by bootstrapping its root module. During development you're likely to bootstrap the AppModule in a main.ts file like this one.

```
src/main.ts

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

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

if (environment.production) {
  enableProdMode();
}

platformBrowserDynamic().bootstrapModule(AppMod
ule);
```

Few built-in Modules

1) NgModule
2) FormsModule
3) HttpModule

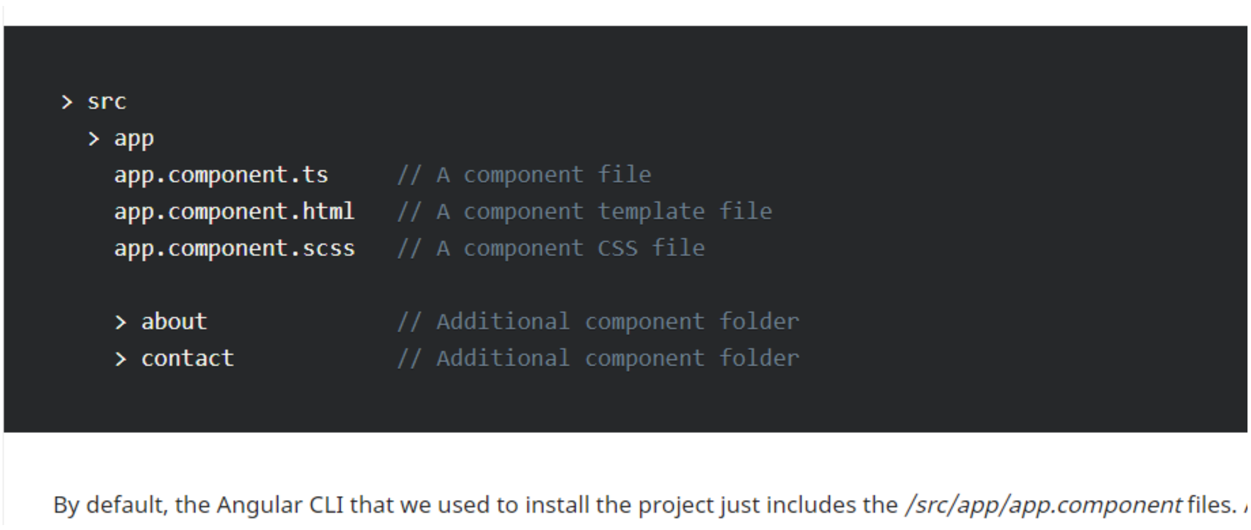
What are Angular Components?

Angular components are the basic building blocks of your app. Each component defines:

Any necessary imports needed by the component,a component decorator, which includes properties that allow to define the template, CSS styling, animations, etc. It is a class, which is where your component logic is stored.

Angular 4 components are simply classes that are designated as a component with the help of a component decorator. Every component has a defined template which can communicate with the code defined in the component class.

Angular components reside within the /src/app folder:



```
> src
  > app
    app.component.ts      // A component file
    app.component.html    // A component template file
    app.component.scss    // A component CSS file

    > about               // Additional component folder
    > contact             // Additional component folder
```

By default, the Angular CLI that we used to install the project just includes the */src/app/app.component* files.

This is what the app.component.ts file will look like by default:

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

// Component Decorator
```

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

// Component Class
export class AppComponent {
  title = 'app works!';
}
```

The Component Decorator

selector: This is the name of the tag that the component is applied to. For instance: <app-root>Loading...</app-root> within index.html.

templateUrl & styleUrls: These define the HTML template and stylesheets associated with this component. You can also use template and styles properties to define inline HTML and CSS.

The @Component decorator takes a required configuration object with the information Angular needs to create and present the component and its view.

Here are a few of the most useful @Component configuration options:

- **selector**: CSS selector that tells Angular to create and insert an instance of this component where it finds a <hero-list> tag in parent HTML. For example, if an app's HTML contains <hero-list></hero-list>, then Angular inserts an instance of the HeroListComponent view between those tags.
- **templateUrl**: relative address of this component's HTML template.
- **styleUrl**: relative address of this component's css file .
- **providers**: array of dependency injection providers for services that the component requires.

Thus components are the basic building blocks of your application. Every component has an associated template, stylesheet, and a class with logic.

Lets look at complete example of first HelloWorld application

App.component.ts

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

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

app.component.html includes

<h1>{{title}}</h1> <br/>

app.component.css includes

```
h1 {
  color: #369;
  font-family: Arial, Helvetica, sans-serif;
  font-size: 250%;
}
```

App.module.ts includes

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

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

import { **AppComponent** } from './app.component';

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

index.html includes

```
<!doctype html>
<html>
<head>
  <meta charset="utf-8">
  <title>SessionComponents</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>Loading...</app-root>

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

Defining and using types

Create Product.ts

```
export class Product {
  constructor(
    public ProductId: number,
    public ProductName: string) { }
}
```

```
product.component.ts
import { Component, OnInit } from
'@angular/core';
import {Product} from './Product';
@Component({
selector: 'app-product',
templateUrl:'./product.component.html',
styleUrls: ['./product.component.css']
})
export class ProductsComponent implements
OnInit {
obj:Product=null;

  ngOnInit() {
     this.obj=new Product(1,"Fridge")
    }
}
```

product.component.html

```
<ul>
      <li>{{obj.ProductId}}--
>{{obj.ProductName}}</li>
 </ul>
```

Add some css for styling <ul> & <li> tags in product.component.css file

Include the component in app.module.ts

In the app.component.html file add

```
<app-product></<app-product>
```

Launch browser go to localhost:4200 and check whether the component gets rendered as expected .

Component Communication

In Angular a component can share data and information with another component by passing data or events. A component can be used inside another component, thus creating a component hierarchy. The component being used inside another component is known as the child component and the enclosing component is known as the parent component. Let us consider the components created in the listing below. We have created a component called **AppChildComponent**, which will be used inside another component.

```
import { Component } from '@angular/core';
@Component({
    selector: 'appchild',
    template: `<h2>Hi {greetMessage}</h2>`
})
export class AppChildComponent {
    greetMessage: string = "I am Child";
}
```

We have also created another component called **AppComponent**. Inside **AppComponent**, we are using AppChildComponent:

```
import {Component } from '@angular/core';
import {AppChildComponent} from
'./appchild.component';
@Component({
   selector: 'my-app',
   template: `<h1>Hello {message}</h1> <br/>
   <appchild></appchild>`,
})
export class AppComponent {
   message : string = "I am Parent";
}
```

In the above listings, **AppComonent** is using **AppChildComponent**, hence **AppComponent** is the parent component and **AppChildComponent** is the child component.

Passing Data From Parent Component to Child Component

Let us start with passing data from the parent component to the child component. This can be done using the input property. **@Input** decorator or input properties are used to pass data from parent to child component. To do this, we'll need to modify child **AppChildComponent** as shown in the listing below:

```
import { Component,Input,OnInit } from
'@angular/core';
@Component({
    selector: 'appchild',
    template: `<h2>Hi {{greetMessage}}</h2>`
})
export class AppChildComponent  implements
OnInit{
    @Input() greetMessage: string ;
        constructor(){
        }
        ngOnInit(){
        }
    }
```

As you notice, we have modified the greetMessage property with the **@Input()** decorator. Also, we have implemented **onInit**, which will be used in demos later. So essentially, in the child component, we have decorated the greetMessage property with the **@Input()** decorator so that value of greetMessage property can be set from the parent component.

Next, let us modify the parent component **AppComponent** to pass data to the child component.

```
import {Component } from '@angular/core';
import {AppChildComponent} from
'./appchild.component';
@Component({
    selector: 'my-app',
    template: `<h1>Hello {{message}}</h1> <br/>
  <appchild
[greetMessage]="childmessage"></appchild>
  `,
})
export class AppComponent  {

        message : string = "I am Parent";
        childmessage : string = "I am passed
from Parent to child component"
    }
```

From the parent component, we are setting the value of the child component's property greetMessage. To pass a value to the child component, we need to pass the child component property inside a square bracket and set its value to any property of parent component. We are passing the value of childmessage property from the parent component to the greetMessage property of the child component.

Intercept Input From Parent Component in Child Component

We may have a requirement to intercept data passed from the parent component inside the child component. This can be done using getter and setter on the input property.

That's all folks. Happy learning!!!!

Introduction to Angular Directives and pipes

Overview of Directives in Angular

There are three kinds of directives in Angular:

1) **Components -** directives with a template.
2) **Structural directives -** change the DOM layout by adding and removing DOM elements.
3) **Attribute directives -** change the appearance or behavior of an element, component, or another directive.

Now we will discuss each one of the above directive in detail.

What are structural Directives?

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

As with other directives, you apply a structural directive to a host element. The directive then does whatever it's supposed to do with that host element and its descendants.

Structural directives are easy to recognize. An asterisk (*) precedes the directive attribute name as in this example.

```
<div *ngIf="hero"
class="name">{{hero.name}}</div>
No brackets. No parentheses. Just *ngIf set to
a string.
```

Three of the common, built-in structural directives—NgIf, NgFor, and NgSwitch...—are described in the Template Syntax guide and seen in samples throughout the Angular documentation. Here's an example of them in a template:

4) **Using NgIf**

NgIf is the simplest structural directive and the easiest to understand. It takes a boolean expression and makes an entire chunk of the DOM appear or disappear.

The ngif element is used to add elements to the HTML code if it evaluates to true, else it will not add the elements to the HTML code.

```
<p *ngIf="true">
  Expression is true and ngIf is true.
  This paragraph is in the DOM.
</p>
<p *ngIf="false">
  Expression is false and ngIf is false.
  This paragraph is not in the DOM.
</p>
```

Let’s now take a look at an example of how we can use the *ngif directive.

- **Step 1 -** First add a property to the class named appStatus. This will be of type Boolean. Let’s keep this value as true.

```
import { Component } from '@angular/core';
@Component ({
   selector: 'my-app',
   templateUrl: 'app/app.component.html'
})
export class AppComponent {
   appTitle: string = 'Welcome';
   appStatus: boolean = true;
}
```

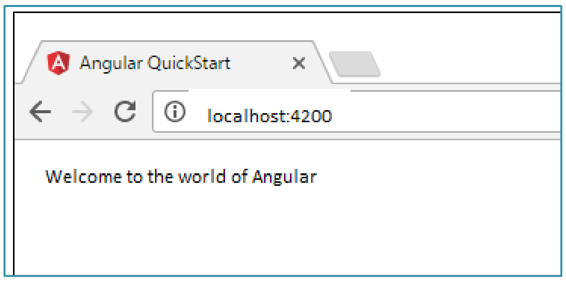
- **Step 2 -** Now in the app.component.html file, add the following code.

  <div *ngIf = 'appStatus'>{{appTitle}} World of Angular </div>

  In the above code, we now have the *ngIf directive. In the directive we are evaluating the value of the appStatus property. Since the value of the property should evaluate to true, it means the div tag should be displayed in the browser.

  Once we add the above code, we will get the following output in the browser.

Output



Welcome to the world of Angular

5) Using ngFor

The ngFor element is used to elements based on the condition of the For loop.

Syntax

*ngFor = 'let variable of variablelist'

The variable is a temporary variable to display the values in the variablelist.

Let's now take a look at an example of how we can use the *ngFor directive.

- **Step 1 −** First add a property to the class named appList. This will be of the type which can be used to define any type of arrays.

```
import { Component } from '@angular/core';
 @Component ({
   selector: 'my-app',
   templateUrl: 'app/app.component.html'
})
export class AppComponent {
   appTitle: string = 'Welcome';
   mobileList: any[] = [ {
      "ID": "1",
      "Name" : "Moto G4 plus"
   },
   {
      "ID": "2",
      "Name" : "Mi-Y1"
   } ];
}
```

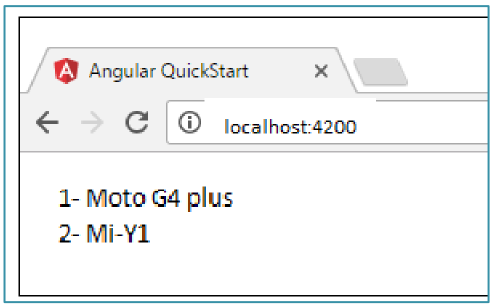
Hence, we are defining the appList as an array which has 2 elements. Each element has 2 sub properties as ID and Name.

- Step 2 − In the app.component.html, define the following code.

```
<div *ngFor = 'let list of mobileList'>
   <ul>
      <li>{{list.ID}} - {{list.Name}}</li>
   </ul>
</div>
```

In the above code, we are now using the ngFor directive to iterate through the appList array. We then define a list where each list item is the ID and name parameter of the array.

Once we add the above code, we will get the following output in the browse.



```
1- Moto G4 plus
2- Mi-Y1
```

b) ngFor with index

The index property of the NgForOf directive context returns the zero-based index of the item in each iteration. You can capture the index in a template input variable and use it in the template.

The next example captures the index in a variable named i and displays it with the hero name like this.

```
<div *ngFor="let hero of heroes; let
i=index">{{i + 1}} - {{hero.name}}</div>
```

Angular 4 NgIf-Then-Else

Angular 4 has introduced NgIf with then and else. If condition in NgIf is false and it is necessary to show a template then we use else and its binding point is <ng-template>. Usually then is the inlined template of NgIf but we can change it and make non-inlined using a binding and binding point will be <ng-template>.

c) Using NgIf with Else

In our application else is used when we want to display something for false condition. The else block is used as follows.

```
<div *ngIf="condition; else
elseBlock">...</div>
<ng-template #elseBlock>...</ng-template>
```

d) Using NgIf with Then and Else

then template is the inline template of NgIf and when it is bound to different value then it displays <ng-template> body having template reference value as then value. NgIf with then and else is used as follows.

```
<div *ngIf="condition; then thenBlock else
elseBlock"></div>
<ng-template #thenBlock>...</ng-template>
<ng-template #elseBlock>...</ng-template>
```

For else block we need to use <ng-template> element. It is referred by a template reference variable. NgIf will use that template reference variable to display else block when condition is false. The <ng-template> element can be written anywhere in the HTML template but it will be readable if we write it just after host element of NgIf. In the false condition, the host element will be replaced by the body of <ng-template> element. Now find the example of NgIf with else.

ngif-else.component.html

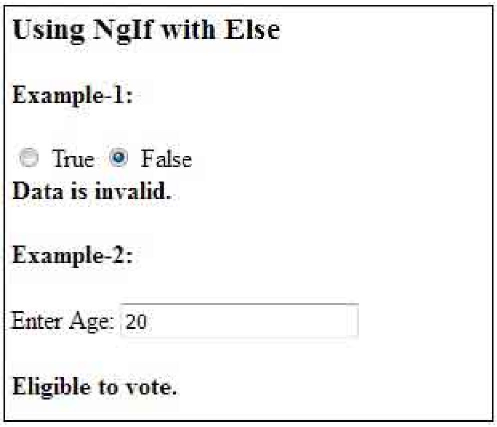
```
<h3>Using NgIf with Else</h3>
<b>Example-1:</b><br/><br/>
<div>
  <input type="radio" name="rad2" (click)=
"changeValue(true)"> True
  <input type="radio" name="rad2" (click)=
"changeValue(false)"> False
</div>
<div *ngIf="isValid; else elseBlock">
   <b>Data is valid.</b>
</div>
```

```
<ng-template #elseBlock>
  <div>
   <b>Data is invalid.</b>
  </div>
</ng-template>

<br/><b>Example-2:</b><br/><br/>
<div>Enter Age:   <input type="text"
[(ngModel)] = 'age'> </div> <br/>
<div *ngIf="age < 18; else elseAgeBlock">
   <b>Not eligible to vote.</b>
</div>
<ng-template #elseAgeBlock>
   <b>Eligible to vote.</b>
</ng-template>
```

In the above code, we are using two examples of NgIf directive. When condition is true, the host element with its descendants will be added in DOM layout and if condition is false then <ng-template> code block will run in the place of host element. Now Find the component used in our example.

```
ngif-else.component.ts
import { Component } from '@angular/core';

@Component({
     selector: 'app-ngif-else',
     templateUrl: './ngif-else.component.html'
})
export class NgIfElseComponent {
     isValid: boolean = true;
     age:number = 12;
     changeValue(valid: boolean) {
  this.isValid = valid;
     }
}
```

Output as below.



Using NgIf with Else

Example-1:

True [✓ False]

Data is invalid.

Example-2:

Enter Age: 20

Eligible to vote.

e) NgForOf with HTML Elements

NgForOf is a repeater directive — a way to present a list of items. You define a block of HTML that defines how a single item should be displayed.

provides several exported values that can be aliased to local variables.

index: Index of current item.

even: True for an even index.

odd: True for an odd index.

first: True for first item.

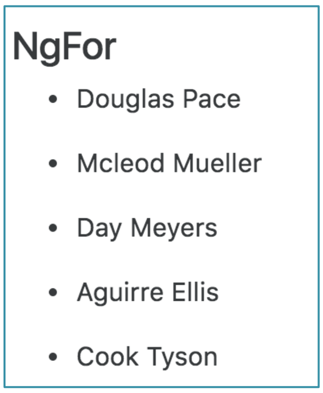
last: True for last item.

NgForOf directive is used with HTML elements as following.

<li *ngFor="let item of items; index as i; even as isEven; odd as isOdd; first as isFirst; last as isLast; trackBy: trackByFn">

```
  ------
</li>
@Component({
  selector: 'ngfor-example',
  template: `
 <ul>
  <li *ngFor="let person of people">
    {{ person.name }}
  </li>
 </ul>
```

```
  `
})
class NgForExampleComponent {
  people: any[] = [
    {
      "name": "Douglas  Pace"
    },
    {
      "name": "Mcleod  Mueller"
    },
    {
      "name": "Day  Meyers"
    },
    {
      "name": "Aguirre  Ellis"
    },
    {
      "name": "Cook  Tyson"
    }
  ];
}
```


NgFor

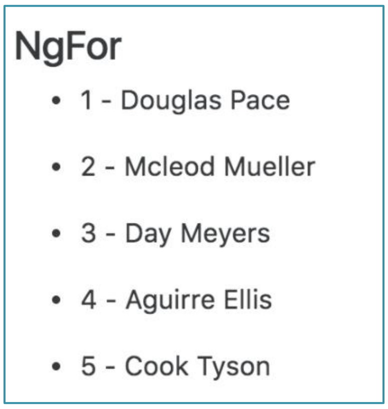
- Douglas Pace
- Mcleod Mueller
- Day Meyers
- Aguirre Ellis
- Cook Tyson

Sometimes we also want to get the index of the item in the array we are iterating over.

We can do this by adding another variable to our ngFor expression and making it equal to index, like so:

```
<ul>
```

```
  <li *ngFor="let person of people; let i = index">
    {{ i + 1 }} - {{ person.name }}
  </li>
</ul>
```



NgFor

- 1 - Douglas Pace
- 2 - Mcleod Mueller
- 3 - Day Meyers
- 4 - Aguirre Ellis
- 5 - Cook Tyson

Few more examples on : index, even and odd

Here we will use NgForOf with index, even and odd local variables using <div> element.

```
<div *ngFor="let person of allPersons; index as i; even as isEven; odd as isOdd">
  <font color="blue" *ngIf="isEven">{{i + 1}}: {{person.personId}} - {{person.name}} - {{person.age}} </font>
  <font color="red" *ngIf="isOdd">{{i + 1}}: {{person.personId}} - {{person.name}} - {{person.age}} </font>
</div>
```

The same can be achieved using <ng-template> as following.

```
<ng-template ngFor let-person [ngForOf]="allPersons" let-i="index" let-isEven="even" let-isOdd="odd">
 <div>
```

```
   <font color="blue" *ngIf="isEven">{{i + 1}}:
{{person.personId}} - {{person.name}} -
{{person.age}} </font>
   <font color="red" *ngIf="isOdd">{{i + 1}}:
{{person.personId}} - {{person.name}} -
{{person.age}} </font>
   <input [(ngModel)]="person.name"> <button
(click)="remove(person.personId)">Remove</button
>
 </div>
</ng-template>
```

first and last

Find the use of first and last local variables of NgForOf using <div> element.

```
<div *ngFor="let person of allPersons; index as
i; first as isFirst; last as isLast">
   <font color="blue" *ngIf="isFirst; else
elseBlock1">{{i + 1}}: {{person.personId}} -
{{person.name}} - {{person.age}} </font>
   <ng-template #elseBlock1>
      <font color="red" *ngIf="isLast; else
elseBlock2">{{i + 1}}: {{person.personId}} -
{{person.name}} - {{person.age}} </font>
      <ng-template #elseBlock2>
     {{i + 1}}: {{person.personId}} -
{{person.name}} - {{person.age}}
      </ng-template>
   </ng-template>
</div>
```

We can do same using <ng-template> as following.

```
<ng-template ngFor let-person
[ngForOf]="allPersons" let-i="index" let-
isFirst="first" let-isLast="last">
 <div>
   <font color="blue" *ngIf="isFirst; else
elseBlock1">{{i + 1}}: {{person.personId}} -
{{person.name}} - {{person.age}} </font>
```

```
   <ng-template #elseBlock1>
      <font color="red" *ngIf="isLast; else
elseBlock2">{{i + 1}}: {{person.personId}} -
{{person.name}} - {{person.age}} </font>
      <ng-template #elseBlock2>
         {{i + 1}}: {{person.personId}} -
{{person.name}} - {{person.age}}
      </ng-template>
   </ng-template>
 </div>
</ng-template>
```

f) **ngFor with trackBy** is used for change propagation. When the contents of the iterator changes, NgForOf makes the corresponding changes to DOM as given below.
   1) If an item is added then new instance of template is added in DOM.
   2) If an item is removed then its respective template is removed from DOM.
   3) If items are reordered, respective templates are reordered in DOM.
   4) If no change for an item, its respective template in DOM will be unchanged.

Angular uses object identity to maintain the entire above task of change propagation. If we want that Angular should not use object identity for change propagation but use user provided identity, then we can achieve it using trackBy . It assigns a function that accepts index and item as function arguments. We can use trackBy with HTML elements as following.

```
import { Component } from '@angular/core';
 @Component({
    selector: 'list-stu',
    templateUrl: './stuList.component.html',
})
export class StudentListComponent {
    employees: any[];
    constructor() {
        this.stulist = [
            {
                code: 's101', name: 'Tom',
gender: 'Male',
                dateOfBirth: '25/6/1988'
            },
            {
                code: 's102', name: 'Alex',
gender: 'Male',
                 dateOfBirth: '9/6/1982'
            },
            {
                code: 's103', name: 'Mike',
gender: 'Male',
                 dateOfBirth: '12/8/1979'
            },
      ];
    }
    trackBystuCode(index: number, stu: any):
string {
    return stu.code;
    }
}
```

employeeList.component.html

```
<table border="1">
    <thead>
        <tr>
            <th>Index</th>
            <th>Code</th>
            <th>Name</th>
            <th>Gender</th>
            <th>Date of Birth</th>
            <th>Isfirst</th>
            <th>IsLast</th>
            <th>IsEven</th>
            <th>IsOdd</th>
        </tr>
    </thead>
    <tbody>
       <tr *ngFor='let stu of stulist;let
i=index;let isFirst = first;
       let isLast = last;let isEven = even; let
isOdd = odd;  trackBy:trackBystuCode'>
        <td>{{i}}</td>
        <td>{{stu.code}}</td>
            <td>{{ stu.name}}</td>
            <td>{{ stu.gender}}</td>
            <td>{{ stu.dateOfBirth}}</td>
            <td>{{isFirst}}</td>
             <td>{{isLast}}</td>
              <td>{{isEven}}</td>
               <td>{{isOdd}}</td>
        </tr>

    </tbody>
</table>
<br />
```

6) **NgSwitch** is an angular directive that displays one element from a possible set of elements based on some condition. NgSwitch uses ngSwitchCase and ngSwitchDefault. NgSwitch uses ngSwitchCase keyword to define a set of possible element trees and ngSwitchDefault is used to define default value when expression condition does not match to any element tree defined by ngSwitchCase. NgSwitch is used as property binding such as [ngSwitch] with bracket [ ]. To define possible set of elements, we need to add asterisk (*) as prefix to the switch case keywords as *ngSwitchCase and *ngSwitchDefault. Whenever NgSwitch finds a match evaluated by expression then the respective element defined by ngSwitchCase is added to DOM and if no match is found then the element defined by ngSwitchDefault is added to DOM.

   Find the code snippet for NgSwitch with bracket [ ].

```
<ul [ngSwitch]="person">
  <li *ngSwitchCase="Sara">Hello Sara</li>
  <li *ngSwitchCase="Williams">Hello Williams
</li>
  <li *ngSwitchCase="Shawn">Hello Shawn </li>
  <li *ngSwitchDefault>Hey Ola</li>
</ul>
```

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-ngswitch',
  template: `<h4>NgSwitch</h4>
<ul *ngFor="let person of people"
    [ngSwitch]="person.country">

  <li *ngSwitchCase="'UK'"
      class="text-success">
    {{ person.name }} ({{ person.country }})
  </li>
  <li *ngSwitchCase="'USA'"
      class="text-primary">
    {{ person.name }} ({{ person.country }})
  </li>
  <li *ngSwitchCase="'HK'"
      class="text-danger">
    {{ person.name }} ({{ person.country }})
  </li>
  <li *ngSwitchDefault
      class="text-warning">
    {{ person.name }} ({{ person.country }})
  </li>
</ul>`
})
export class SwitchDemoComponent {
  people: any[] = [
    {
      "name": "Douglas  Pace",
      "age": 35,
      "country": 'MARS'
    },
    {
      "name": "Mcleod  Mueller",
      "age": 32,
      "country": 'USA'
    },
    {
      "name": "Day  Meyers",
      "age": 21,
      "country": 'HK'
    },
    {
      "name": "Aguirre  Ellis",
      "age": 34,
      "country": 'UK'
    },
    {
      "name": "Cook  Tyson",
      "age": 32,
      "country": 'USA'
    }
  ];
}
```

How to create Custom Structural Directive?

Steps to Create Custom Structural Directive

Find the steps to create custom structural directive.

- **Step-1:** Create a class decorated with @Directive. Using selector metadata we will give a name to our directive. The directive name will be enclosed with bracket [ ] such as [cpIf]. Find the example.

```
@Directive({
     selector: '[cpIf]'
})
```

export class CpIfDirective { } In the above code the custom directive name is cpIf. Custom directive name should not be angular keyword, it should also not start with ng. We should use our own keyword such as company name as prefix in the nomenclature of our custom directive.

- **Step-2:** Create a setter method decorated with @Input(). We need to take care that the method name should be same as directive name. Find the code snippet.

```
@Input() set cpIf(condition: boolean) {
} If we want different method name then
@Input() alias should be same as directive
name. Find the code snippet.
 @Input('cpIf') set myCpIf(condition:
boolean) {
}
```

- **Step-3:** To make directive globally available in our application, we need to configure it in application module in the same way as we configure component. Find the sample example for configuration.

```
import { CpIfDirective }  from
'./directives/cp-if.directive.ts';

@NgModule({
----------------
----------------
  declarations: [
        ---------
        ---------
    CpIfDirective
        ---------
        ---------
  ]
----------------
----------------
})
export class AppModule { }
```

- **Step-4:** Now we are ready to use our custom structural directive in HTML template. We can use our directive in the same way as we use angular in-built structural directive such as NgFor and NgIf. We can use structural directive either with * prefix in host element or by using ng-template. Using structural directive with * is preferable because it is considered more readable. Now find the code snippet how to use structural directive.

g) Using directive with * prefix in host element.

```
<div *cpIf="showCpIf">
    <b>Hello cpIf Directive.</b>
</div> b. Using directive with ng-template.
<ng-template [cpIf]="showCpIf">
    <div>
  <b>Hello cpIf Directive.</b>
    </div>
</ng-template>
```

TemplateRef and ViewContainerRef

To change DOM layout we should use TemplateRef and ViewContainerRef in our structural directive.

TemplateRef : It represents an embedded template that can be used to instantiate embedded views.

ViewContainerRef: It represents a container where one or more views can be attached.

To use the above classes in our directive, first we need to instantiate them. Instantiate these classes using dependency injection in constructor as following.

constructor(private templateRef: TemplateRef<any>,

private viewContainer: ViewContainerRef) { }

To add host element in DOM layout, we need to call createEmbeddedView() method of ViewContainerRef. Find the line of code.

this.viewContainer.createEmbeddedView(this.templateRef); If we want to clear view container, call clear() method of ViewContainerRef as given below.

this.viewContainer.clear();

Create Directive for If Condition

We will create a custom structural directive that will take boolean argument and will work like NgIf directive. When we will pass true, then the host element and its descendants will be added in DOM layout and if we pass false then host element and its descendants will be removed from DOM layout. Find the code.

cp-if.directive.ts

import { Directive, TemplateRef, ViewContainerRef, Input } from '@angular/core';

```
@Directive({
     selector: '[cpIf]'
})
export class CpIfDirective {
  constructor( private templateRef:
TemplateRef<any>,
               private viewContainer:
ViewContainerRef) { }
  @Input() set cpIf(condition: boolean) {
     if (condition) {

this.viewContainer.createEmbeddedView(this.temp
lateRef);
     } else {
        this.viewContainer.clear();
    }
  }
}
```

Here the directive name is cpIf. Find the sample example to use it.

```
<div>
  <input type="radio" name="rad1" (click)=
"showCpIf= true"> True
  <input type="radio" name="rad1" (click)=
"showCpIf= false"> False
</div>
<br/>
<div *cpIf="showCpIf">
```

```
  <b>Hello cpIf Directive.</b>
</div>
<ng-template [cpIf]="!showCpIf">
   <div>
      <b>Message not Available.</b>
  </div>
</ng-template>
```

Create Directive for Loop

We will create a custom structural directive that will create host element as many times as given by user in DOM layout. Find the code.

```
cp-loop.directive.ts
import { Directive, TemplateRef,
ViewContainerRef, Input } from '@angular/core';
@Directive({
     selector: '[cpLoop]'
})
export class CpLoopDecorator {
  constructor( private templateRef:
TemplateRef<any>,
               private viewContainer:
ViewContainerRef) { }
  @Input('cpLoop') set loop(num: number) {
      for(var i=0; i < num; i++) {
  this.viewContainer.createEmbeddedView(this.te
mplateRef);
      }
  }
} Here directive name is cpLoop and we can use
it as given below.
<ul>
 <li *cpLoop="5" >
     Hello World!
 </li>
</ul>
```

What are Attribute Directives?

An Attribute directive changes the appearance or behavior of a

DOM element. Attribute directives are a way of changing the appearance or behavior of a component or a native DOM element. Example are **ng-class & ng-style.** Angular supports creating custom attribute directive.

```
!-- this property syntax for style -->
<div [style.color]="'orange'">style using
property syntax, this text is orange</div>
<!-- uses component variables-->
<div [ngStyle]="{'color': color, 'font-size':
size, 'font-weight': 'bold'}">
  {{svar}}
</div>
<div [ngStyle]="{'color': color, 'font-size':
size + 'px'}">
  style using ngStyle
</div>

<!-- this property syntax for css class -->
<div [className]="'blue'">CSS class using
property syntax, this text is blue</div>
<p [className]="'style1'"> style applied using
class name</p>
<!-- this property syntax for ng-class -->
<div [ngClass]="['bold-text', 'green']">array
of classes(using ngclass) </div>
<div [ngClass]="'italic-text blue'">string of
classes (using ngclass again)</div>
<div [ngClass]="{'small-text': true, 'red':
true}">object of classes (using ngclass -one
more example)</div>
<span [ngClass]="displayText">using
ngClass</span>
```

How to create Custom Attribute Directive?

Angular custom attribute is created to change appearance and behavior of HTML element. Find the steps to create custom attribute directive.

1) Create a class decorated with @ Directive().
2) Assign the attribute directive name using selector metadata of @Directive() decorator enclosed with bracket [] .
3) Use ElementRef class to access DOM to change host element appearance. To change appearance of HTML element in DOM, we need to use ElementRef within custom directive definition. ElementRef can directly access the DOM. We can use it directly without using directive but that can lead to XSS attacks. It is safer to use ElementRef within directive definition. Now let us create custom attribute directives.

```
cp-default-theme.directive.ts.

import { Directive, ElementRef, AfterViewInit
} from '@angular/core';
@Directive({
     selector: '[cpDefaultTheme]'
})
export class CPDefaultThemeDirective
implements AfterViewInit {
    constructor(private elRef: ElementRef) {
    }
    ngAfterViewInit(): void {
       this.elRef.nativeElement.style.color =
'blue';

this.elRef.nativeElement.style.fontSize =
'20px';
    }
}
```

AfterViewInit is the lifecycle hook that is called after a component view has been fully initialized. To use AfterViewInit, our class will implement it and override its method ngAfterViewInit().

Directives are declared in application module in the same way as we declare component. We need to configure our each and every directive in application module within the declarations block of @NgModule decorator.

```
import { CPDefaultThemeDirective }  from
'./attribute-directives/cp-default-
theme.directive';
@NgModule({
  ---
  declarations: [
        ---
  CPDefaultThemeDirective
  ]
})
```

export class AppModule { } Now we are ready to use our custom directive in our application in any component template. Custom directive selector name: cpDefaultTheme

How to use our custom directive?

<p cpDefaultTheme> cpDefaultTheme Directive Demo</p>

The text of the <p> element will be blue with font size 20px.

What are Component Directives?

Components are Directives with templates .Angular components are a subset of directives.

Unlike directives, components always have a template and Only one component can be instantiated per an element in a template.

For examples please refer to the components chapter.

Introduction to Pipes in Angular4

Every application starts out with what seems like a simple task: get data, transform them, and show them to users. Introducing Angular pipes, a way to write display-value transformations that you can declare in your HTML. A pipe takes in data as input and transforms it to a desired output. Built-in pipes

Angular comes with a stock of pipes such as DatePipe, DecimalPiple, Titlecasepipe, UpperCasePipe, LowerCasePipe, CurrencyPipe, JsonPipe, PercentPipe. They are all available for use in any template .

Using pipes

Lets see how to use pipes to transform a component's birthday property into a human-friendly date.

1) Date Pipe

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-hero-birthday',
  template: `<p>The hero's birthday is {{
birthday | date }}</p>`
})
export class HeroBirthdayComponent {
  birthday = new Date(1988, 3, 15); // April
15, 1988
}

<p>The hero's birthday is {{ birthday | date
}}</p>
<p>The hero's birthday is {{ birthday |
date:"MM/dd/yy" }} </p>

The code below assumes dateObj is (year:
2015, month: 6, day: 15, hour: 21, minute:
43, second: 11) in the local time and locale
is 'en-US':

@Component({
  selector: 'date-pipe',
  template: `<div>
    <!--output 'Jun 15, 2015'-->
    <p>Today is {{today | date}}</p>

    <!--output 'Monday, June 15, 2015'-->
    <p>Or if you prefer, {{today |
date:'fullDate'}}</p>

    <!--output '9:43 AM'-->
    <p>The time is {{today |
date:'shortTime'}}</p>

    <!--output 'Monday, June 15, 2015 at
9:03:01 AM GMT+01:00' -->
    <p>The full date/time is {{today |
date:'full'}}</p>
      </div>`
})
export class DatePipeComponent {
  today = Date.now();
  fixedTimezone = '2015-06-15T09:03:01+0900';
}
```

2) Using Currency Pipe

```
import { Component } from '@angular/core';
@Component({
  selector: 'currency-app',
  template: `
            <h3> Currency Pipe</h3>
        <div>
          <p> {{cur1 | currency:'INR':false}}
</p>
          <p> {{cur2 |
currency:'INR':false:'1.2-4'}} </p>
          <p> {{cur1 | currency}} </p>
          <p> {{cur2 |
currency:'USD':true:'2.2-4'}} </p>
        </div>
          `

})
export class CurrencyPipeComponent {
  cur1: number = 0.25;
  cur2: number = 10.263782;
}
Output.
Currency Pipe
INR0.25
INR10.2638
USD0.25
$10.2638
```

3) Using Titlecase, Uppercase, Lowecase

```
import {   Component} from '@angular/core';
@Component ({
   selector: 'my-app',
   templateUrl: 'app/app.component.html'
})
export class AppComponent {
   TutorialName: string = 'Angular4 Pippes;
   appList: string[] = ["Binding", "Display",
"Services"];
}
How to use Pipes
<div>
   The name of this Tutorial is
{{TutorialName}}<br>
   The first Topic is {{appList[0] |
uppercase }}<br>
   The second Topic is {{appList[1] |
lowercase }}<br>
   The third Topic is {{appList[2]| titlecase
}}<br>
</div>
```

Few more examples

```
<div>
          <p> {{cur1 | currency:'INR':false}}
</p>
          <p> {{cur2 |
currency:'INR':false:'1.2-4'}} </p>
```

```
          <p> {{cur1 | currency}} </p>
          <p> {{cur2 |
currency:'USD':true:'2.2-4'}} </p>
</div>
```

4) Using Json Pipe

Angular JsonPipe converts data into JSON string. It uses json keyword with pipe operator to convert any expression result into JSON string. In our example we will create object and array of objects and then we will display them into JSON string using JsonPipe. It is used as follows.

```
<pre> {{person | json}} </pre>
```

Create Component and Classes used in Example

In our example we are using following classes whose object will be converted into JSON string.

```
name.ts
export class Person {
    constructor(public FirstName:string,
public LastName:string;Gender:string) {
    }
}
```

```
jsonpipe.component.ts
import {Component} from '@angular/core';
import {Person} from './person';
@Component({
    selector: 'jsonpipe-app',
    template: `<pre> {{person | json}} </pre>
`
})
export class JsonPipeComponent {
   person = new
Person('Sara','Parker','Female');
}
```

The output is as below.

```
{
   “FirstName”:”Sara”,
    “LastName”:”Parker”,
     “Gender”: “Female”
}
```

How to create a Custom Pipes?

To create a custom Pipe use @Pipe Decorator and PipeTransform Interface

Here we will create a simple custom pipe named as welcome. The syntax is as follows.

string_expression | welcome

Now follow the steps to create our welcome pipe.

- Step 1: Create a typescript class named as WelcomePipe.
- Step 2: Import Pipe and PipeTransform interface from angular core module.
- Step 3: Decorate WelcomePipe with @Pipe . Its name metadata will define custom pipe name.
- Step 4: WelcomePipe will implement PipeTransform interface.
- Step 5: Override transform() method of PipeTransform interface. The parameter of transform() can be of any type. In our example we are using string data type and return type is also a string. Here we will perform task which needs to be done by our custom pipe and return the result. This is the result which will be returned by custom pipe.
- Step 6: To make custom pipe available at application level, declare WelcomePipe in @NgModule decorator.

  Here is the WelcomePipe class- welcome.pipe.ts

```
import {Pipe, PipeTransform} from
'@angular/core';
@Pipe({
    name: 'welcome'
})
export class WelcomePipe implements
PipeTransform {
  transform(value: string): string {
    let message = "Welcome " + value + “to the
World ofAngular”
```

```
    return message;
  }
}
```

The name metadata of @Pipe decorator has the value welcome and that will be the name of our custom pipe. In our custompipe.component.ts file we are using welcome pipe as given below.

{{person.name | welcome}} Output

Welcome Ganesha to the World of Angular

Here an additional example on how to create a custom a pipe that takes a string and reverses the order of the letters. Here's the code that would go into a reverse-str.pipe.ts file.

```
import { Pipe, PipeTransform } from
'@angular/core';
@Pipe({name: 'reverseStr'})
export class ReverseStr implements PipeTransform
{
  transform(value: string): string {
    let newStr: string = "";
    for (var i = value.length - 1; i >= 0; i--)
{
      newStr += value.charAt(i);
    }
    return newStr;
  }
}
```

How to use the above pipe

```
<p>{{ name | reverseStr }}</p>
```

Angular Data Binding and Event Handling

You can display data by binding controls in an HTML template to properties of an Angular component.

Showing component properties with interpolation

The easiest way to display a component property is to bind the property name through interpolation. With interpolation, you put the property name in the view template, enclosed in double curly braces: {{propertyofcomponent}}.

The example of interpolation as given below

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  template: `
    <h1>{{title}}</h1>
    <h2>My favorite hero is: {{myHero}}</h2>   `
})
export class AppComponent {
  title = 'Tour of Heroes';
  myHero = 'Windstorm';
}
```

ou added two properties to the formerly empty component: title and myHero.

The template displays the two component properties using double curly brace interpolation:

template: `    <h1>{{title}}</h1>    <h2>My favorite hero is: {{myHero}}</h2>`

Angular automatically pulls the value of the title and myHero properties from the component and inserts those values into the browser. Angular updates the display when these properties change.

Showing an array property with *ngFor

To display a list of heroes, begin by adding an array of hero names to the component and redefine myHero to be the first name in the array.

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

@Component({
  selector: 'app-root',
  template: `
  <h1>{{title}}</h1>
  <h2>My favorite hero is: {{myHero.name}}</h2>
  <p>Heroes:</p>
  <ul>
    <li *ngFor="let hero of heroes">
      {{ hero.name }}
      </li>
  </ul>
  <p *ngIf="heroes.length > 3">There are many 
heroes!</p>
`
})
export class AppComponent {
  title = 'Tour of Heroes';
  heroes = [
    new Hero(1, 'Windstorm'),
    new Hero(13, 'Bombasto'),
    new Hero(15, 'Magneta'),
    new Hero(20, 'Tornado')
  ];
  myHero = this.heroes[0];
}
```

Now use the Angular ngFor directive in the template to display each item using **interpolation** in the heroes list.

```
template: `  <h1>{{title}}</h1>   <h2>My 
favorite hero is: {{myHero}}</h2>
<p>Heroes:</p>
  <ul>
    <li *ngFor="let hero of heroes">
      {{ hero }}
    </li>
  </ul>`
```

```
This UI uses the HTML unordered list with <ul>
and <li> tags. The *ngFor in the <li> element
is the Angular "repeater" directive. It marks
that <li> element (and its children) as the
"repeater template":
<li *ngFor="let hero of heroes">
  {{ hero }}
</li>
```

Binding syntax: An overview

Data binding is a mechanism for coordinating what users see, with application data values. While you could push values to and pull values from HTML, the application is easier to write, read, and maintain if you turn these chores over to a binding framework. You simply declare bindings between binding sources and target HTML elements and let the framework do the work.

Angular provides many kinds of data binding. This guide covers most of them, after a high-level view of Angular data binding and its syntax.

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

Data direction	**Syntax**	**Type**

One-way from data source to view target	content_copy{{expression}}
	[target]="expression"
	bind-target="expression"
		Interpolation
		Property
		Attribute
		Class
		Style

One-way from view target to data source	content_copy(target)="statement" on-target="statement"	Event

Two-way	content_copy[(target)]="expression" bindon-target="expression"	Two-way

Binding targets

The **target of a data binding** is something in the DOM. Depending on the binding type, the target can be an (element | component | directive) property, an (element | component | directive) event, or (rarely) an attribute name. The following table summarizes:

Type	Target	Examples
Proper ty	Element property Component prop erty Directive propert y	src/app/app.component.html content_copy<img [src]="heroImageUrl"> <app-hero-detail [hero]="currentHero"></app-hero-detail> <div [ngClass]="{'special': isSpecial}"></div>
Event	Element event Component even t Directive event	src/app/app.component.html content_copy<button (click)="onSave()">Save</button> <app-hero-detail (deleteRequest)="deleteHero()"> </app-hero-detail> <div (myClick)="clicked=$event" clickable>click me</div>
Two-way	Event and property	src/app/app.component.html content_copy<input [(ngModel)]="name">
Attribu te	Attribute (the exception)	src/app/app.component.html content_copy<button [attr.aria-label]="help">help</button>

Class	class property	src/app/app.component.html content_copy<div [class.special]="isSpecial">Special </div>
Style	style property	src/app/app.component.html content_copy<button [style.color]="isSpecial ? 'red' : 'green'">

With this broad view in mind, you're ready to look at binding types in detail.

Property binding ([property])

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

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

src/app/app.component.html

content_copy<img [src]="heroImageUrl">

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

src/app/app.component.html

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

Another is setting a property of a directive:

src/app/app.component.html

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

Yet another is setting the model property of a custom component (a great way for parent and child components to communicate):

src/app/app.component.html

```
content_copy<app-hero-detail
[hero]="currentHero"></app-hero-detail>
```

One-way in

People often describe property binding as one-way data binding because it flows a value in one direction, from a component's data property into a target element property.

You cannot use property binding to pull values out of the target element. You can't bind to a property of the target element to read it. You can only set it.

Similarly, you cannot use property binding to call a method on the target element.

If the element raises events, you can listen to them with an event binding.

If you must read a target element property or call one of its methods, you'll need a different technique. See the API reference for ViewChild and ContentChild.

Binding target

An element property between enclosing square brackets identifies the target property. The target property in the following code is the image element's src property.

```
src/app/app.component.html
content_copy<img [src]="heroImageUrl">
Some people prefer the bind- prefix
alternative, known as the canonical form:
src/app/app.component.html
content_copy<img bind-src="heroImageUrl">
```

The target name is always the name of a property, even when it appears to be the name of something else. You see src and may think it's the name of an attribute. No. It's the name of an image element property.

Element properties may be the more common targets, but Angular looks first to see if the name is a property of a known directive, as it is in the following example:

src/app/app.component.html

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

Technically, Angular is matching the name to a directive input, one of the property names listed in the directive's inputs array or a property decorated with @Input(). Such inputs map to the directive's own properties.

If the name fails to match a property of a known directive or element, Angular reports an “unknown directive” error.

Avoid side effects

As mentioned previously, evaluation of a template expression should have no visible side effects. The expression language itself does its part to keep you safe. You can't assign a value to anything in a property binding expression nor use the increment and decrement operators.

Of course, the expression might invoke a property or method that has side effects. Angular has no way of knowing that or stopping you.

The expression could call something like getFoo(). Only you know what getFoo() does. If getFoo() changes something and you happen to be binding to that something, you risk an unpleasant experience. Angular may or may not display the changed value. Angular may detect the change and throw a warning error. In general, stick to data properties and to methods that return values and do no more.

Return the proper type

The template expression should evaluate to the type of value expected by the target property. Return a string if the target property expects a string. Return a number if the target property expects a number. Return an object if the target property expects an object.

The hero property of the HeroDetail component expects a Hero object, which is exactly what you're sending in the property binding:

```
src/app/app.component.html
content_copy<app-hero-detail
[hero]="currentHero"></app-hero-detail>
```

Remember the brackets

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

Don't make the following mistake:

src/app/app.component.html

```
content_copy<!-- ERROR:
HeroDetailComponent.hero expects a
     Hero object, not the string "currentHero"
-->
  <app-hero-detail hero="currentHero"></app-
hero-detail>
```

One-time string initialization

You should omit the brackets when all of the following are true:

- The target property accepts a string value.
- The string is a fixed value that you can bake into the template.
- This initial value never changes.

You routinely initialize attributes this way in standard HTML, and it works just as well for directive and component property initialization. The following example initializes the prefix property of the HeroDetailComponent to a fixed string, not a template expression. Angular sets it and forgets about it.

src/app/app.component.html

content_copy<app-hero-detail prefix="You are my" [hero]="currentHero"></app-hero-detail>

The [hero] binding, on the other hand, remains a live binding to the component's currentHero property.

Property binding or interpolation?

You often have a choice between interpolation and property binding. The following binding pairs do the same thing:

```
src/app/app.component.html
```

```
content_copy<p><img src="{{heroImageUrl}}"> is
the <i>interpolated</i> image.</p>
<p><img [src]="heroImageUrl"> is the
<i>property bound</i> image.</p>
```

```
<p><span>"{{title}}" is the <i>interpolated</i>
title.</span></p>
<p>"<span [innerHTML]="title"></span>" is the
<i>property bound</i> title.</p>
Interpolation is a convenient alternative to
property binding in many cases.
```

When rendering data values as strings, there is no technical reason to prefer one form to the other. You lean toward readability, which tends to favor interpolation. You suggest establishing coding style rules and choosing the form that both conforms to the rules and feels most natural for the task at hand.

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

Class binding

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

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

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

src/app/app.component.html

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

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

src/app/app.component.html

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

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

src/app/app.component.html

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

<!-- binding to `class.special` trumps the
class attribute -->
<div class="special"
     [class.special]="!isSpecial">This one is
not so special</div>
While this is a fine way to toggle a single
class name, the NgClass directive is usually
preferred when managing multiple class names at
the same time.
```

Style binding

You can set inline styles with a style binding.

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

```
src/app/app.component.html
content_copy<button [style.color]="isSpecial ?
'red': 'green'">Red</button>
<button [style.background-color]="canSave ?
'cyan': 'grey'" >Save</button>
Some style binding styles have a unit
extension. The following example conditionally
sets the font size in “em” and “%” units .
```

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

While this is a fine way to set a single style, the NgStyle directive is generally preferred when setting several inline styles at the same time.

Note that a style property name can be written in either dash-case, as shown above, or camelCase, such as fontSize.

Event binding ((event))

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

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

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

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

src/app/app.component.html

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

Target event

A name between parentheses - for example, (click) - identifies the target event. In the following example, the target is the button's click event.

```
src/app/app.component.html
```

```
content_copy<button
(click)="onSave()">Save</button>
Some people prefer the on- prefix alternative,
known as the canonical form:
src/app/app.component.html
content_copy<button on-click="onSave()">On
Save</button>
```

Element events may be the more common targets, but Angular looks first to see if the name matches an event property of a known directive, as it does in the following example:

src/app/app.component.html

```
content_copy<!-- `myClick` is an event on the
custom `ClickDirective` -->
<div (myClick)="clickMessage=$event"
clickable>click with myClick</div>
```

The myClick directive is further described in the section on aliasing input/output properties.

If the name fails to match an element event or an output property of a known directive, Angular reports an “unknown directive” error. event and event handling statements

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

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

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

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

Consider this example:

src/app/app.component.html

```
content_copy<input [value]="currentHero.name"

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

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

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

If the event belongs to a directive (recall that components are directives), $event has whatever shape the directive decides to produce.

Custom events with EventEmitter

Directives typically raise custom events with an Angular EventEmitter. The directive creates an EventEmitter and exposes it as a property. The directive calls EventEmitter.emit(payload) to fire an event, passing in a message payload, which can be anything. Parent directives listen for the event by binding to this property and accessing the payload through the $event object.

Consider a HeroDetailComponent that presents hero information and responds to user actions. Although the HeroDetailComponent has a delete button it doesn't know how to delete the hero itself. The best it can do is raise an event reporting the user's delete request.

Here are the pertinent excerpts from that HeroDetailComponent:

src/app/hero-detail.component.ts (template)

content_copytemplate: `

```
<div>
  <img src="{{heroImageUrl}}">
  <span [style.text-decoration]="lineThrough">
    {{prefix}} {{hero?.name}}
  </span>
  <button (click)="delete()">Delete</button>
```

```
</div>`
src/app/hero-detail.component.ts
(deleteRequest)
content_copy// This component makes a request
but it can't actually delete a hero.
deleteRequest = new EventEmitter<Hero>();
delete() {
  this.deleteRequest.emit(this.hero);
}
```

The component defines a deleteRequest property that returns an EventEmitter. When the user clicks delete, the component invokes the delete() method, telling the EventEmitter to emit a Hero object.

Now imagine a hosting parent component that binds to the HeroDetailComponent's deleteRequest event.

```
src/app/app.component.html (event-binding-to-
component)
content_copy<app-hero-detail
(deleteRequest)="deleteHero($event)"
[hero]="currentHero"></app-hero-detail>
When the deleteRequest event fires, Angular
calls the parent component's deleteHero method,
passing the hero-to-delete (emitted by
HeroDetail) in the $event variable.
Two-way binding ( [(...)] )
```

You often want to both display a data property and update that property when the user makes changes.

On the element side that takes a combination of setting a specific element property and listening for an element change event.

Angular offers a special two-way data binding syntax for this purpose, [(x)]. The [(x)] syntax combines the brackets of property binding, [x], with the parentheses of event binding, (x).

The [(x)] syntax is easy to demonstrate when the element has a settable property called x and a corresponding event named xChange. Here's a SizerComponent that fits the pattern. It has a size value property and a companion sizeChange event:

```
src/app/sizer.component.ts
content_copyimport { Component, EventEmitter,
Input, Output } from '@angular/core';

@Component({
  selector: 'app-sizer',
  template: `
  <div>
    <button (click)="dec()" title="smaller">-
</button>
    <button (click)="inc()"
title="bigger">+</button>
    <label [style.font-
size.px]="size">FontSize: {{size}}px</label>
  </div>`
})
export class SizerComponent {
  @Input()  size: number | string;
  @Output() sizeChange = new
EventEmitter<number>();

  dec() { this.resize(-1); }
  inc() { this.resize(+1); }

  resize(delta: number) {
    this.size = Math.min(40, Math.max(8,
+this.size + delta));
    this.sizeChange.emit(this.size);
  }
}
```

The initial size is an input value from a property binding. Clicking the buttons increases or decreases the size, within min/max values constraints, and then raises (emits) the sizeChange event with the adjusted size.

Here's an example in which the AppComponent.fontSizePx is two-way bound to the SizerComponent:

```
content_copy<app-sizer
[(size)]="fontSizePx"></app-sizer>
```

```
<div [style.font-
size.px]="fontSizePx">Resizable Text</div>
The AppComponent.fontSizePx establishes the
initial SizerComponent.size value. Clicking the
buttons updates the AppComponent.fontSizePx via
the two-way binding. The revised
AppComponent.fontSizePx value flows through to
the style binding, making the displayed text
bigger or smaller.
```

The two-way binding syntax is really just syntactic sugar for a property binding and an event binding. Angular desugarsthe SizerComponent binding into this:

```
content_copy<app-sizer [size]="fontSizePx"
(sizeChange)="fontSizePx=$event"></app-sizer>
```

The $event variable contains the payload of the SizerComponent.sizeChange event. Angular assigns the $eventvalue to the AppComponent.fontSizePx when the user clicks the buttons.

Clearly the two-way binding syntax is a great convenience compared to separate property and event bindings.

It would be convenient to use two-way binding with HTML form elements like <input> and <select>. However, no native HTML element follows the x value and xChange event pattern.

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

Built-in attribute directives

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

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

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

NgClass

You typically control how elements appear by adding and removing CSS classes dynamically. You can bind to the ngClass to add or remove several classes simultaneously.

A class binding is a good way to add or remove a single class.

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

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

Try binding ngClass to a key:value control object. Each key of the object is a CSS class name; its value is true if the class should be added, false if it should be removed.

Consider a setCurrentClasses component method that sets a component property, currentClasses with an object that adds or removes three classes based on the true/false state of three other component properties:

```
currentClasses: {};
setCurrentClasses() {
  // CSS classes: added/removed per current
state of component properties
  this.currentClasses =  {
    'saveable': this.canSave,
    'modified': !this.isUnchanged,
    'special':  this.isSpecial
  };
}
```

Adding an ngClass property binding to currentClasses sets the element's classes accordingly:

<div [ngClass]="currentClasses">This div is initially saveable, unchanged, and special</div>

It's up to you to call setCurrentClasses(), both initially and when the dependent properties change.

NgStyle

You can set inline styles dynamically, based on the state of the component. With NgStyle you can set many inline styles simultaneously.

A style binding is an easy way to set a single style value.

```
<div [style.font-size]="isSpecial ? 'x-large' :
'smaller'" >
  This div is x-large or smaller.
</div>
```

To set many inline styles at the same time, the NgStyle directive may be the better choice.

Try binding ngStyle to a key:value control object. Each key of the object is a style name; its value is whatever is appropriate for that style.

Consider a setCurrentStyles component method that sets a component property, currentStyles with an object that defines three styles, based on the state of three other component properties:

```
currentStyles: {};
setCurrentStyles() {
  // CSS styles: set per current state of
component properties
  this.currentStyles = {
    'font-style':  this.canSave      ? 'italic'
: 'normal',
    'font-weight': !this.isUnchanged ? 'bold'
: 'normal',
    'font-size':   this.isSpecial    ? '24px'
: '12px'
  };
}
```

Adding an ngStyle property binding to currentStyles sets the element's styles accordingly:

```
<div [ngStyle]="currentStyles">
```

This div is initially italic, normal weight, and extra large (24px).

```
</div>
```

It's up to you to call setCurrentStyles(), both initially and when the dependent properties change.

NgModel - Two-way binding to form elements with [(ngModel)]

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

Two-way data binding with the NgModel directive makes that easy. Here's an example:

```
import { Component,Input, Output } from
'@angular/core';
@Component({
  selector: 'demotwo-way',
  templateUrl: './twoway.component.html',
 })
export class DemoTwoWayComponent {
  hiMessage1:String="Hi There";
   hiMessage2:String="Ola Senorita!!!!!!!";
  Thetitle:String = 'Techno Solutions Ltd';
 }
twoway.component.html is as below.
  <p>one way from component to View</p>
<input [ngModel] ="hiMessage1"/>
{{hiMessage1}} <br/>
<p>Two way from component to View and vice
versa</p>
<input [(ngModel)] ="hiMessage2"/>
<p>{{hiMessage2}}</p> <br/>
<p>Two way from component to View-same as
above</p>
<input [value] ="Thetitle"
(input)="Thetitle=$event.target.value"/>
<p>{{Thetitle}}</p>
```

Event handling in Angular

Consider the component code given below.

```
import { Component } from '@angular/core';
@Component({
  selector: 'eventdemo-app',
  templateUrl:
'./Simpleeventdemo.component.html'
})
export class SimpleEventComponent {
    messageText = '';
    onClickMe() {
        this.messageText = "You actually
clicked me........!!!!";
    }
}
```

Simpleeventdemo.component.html is as below

```
<div>
    <h5>Event binding Example</h5>
     <button (click)="onClickMe()">Click
me!</button>
    <br/><br/>
    <span> {{messageText}} </span>
</div>
```

The component includes the onclickMe() method which is invoked when the button is clicked as the “click” event is associated with the said method. The event is specified in “()” .

```
Custom event handling in Angular  using @output
and eventemitter class.
The “citycust.component.ts” is a sbelow.
import {Component, EventEmitter, Input, Output}
from '@angular/core';
import {Student} from './student';
@Component({
  selector: 'city-cust',
  templateUrl: './citycust.component.html',
 // styleUrls: ['./hellocust.component.css']
})
export class CityCustComponent {
    @Input('myCity') strCity : string;
    @Output('myCityChange') cityObj = new
EventEmitter<string>();
    emitCity() {
      this.cityObj.emit(this.strCity);
        }
}
```

- @Input(): Defines input variable in component communication. It is used to communicate from parent to child component using property binding.
- @Output(): Defines output variable in component communication. It is used to communicate from child to parent component using custom event binding.

The above component includes input variable strCity with the alias “myCity”. The @Output() defines an output variable named 'myCityChange' is a custom event name.

EentEmitter is a class in angular framework. It has emit() method that emits custom events. We can use EventEmitter in custom event binding. To achieve it first import it in our component file as given below.

```
import {Component, EventEmitter, Input, Output}
from '@angular/core';
And then initialize it using @Output decorator
as follows.
@Output('myCityChange') cityObj = new
EventEmitter<string>();
Using emit() method of EventEmitter class ,
emits string data to parent component in custom
event binding as follows.
this.cityObj.emit(this.strCity);
```

The object emitted by emit() method can be accessed using event object $event. Custom event binding is used in parent child component communication. Here the data will be sent from child component to parent component. To send data emit() method is used as shown above.

Now we are ready to use variable 'myCityChange' as custom event. Event binding can be done using parenthesis () or on-keyword. Find event binding using () as shown below.

The citycust.component.html includes the markup as below.

```
<input [(ngModel)] ="strCity" />
<button (click)="emitCity()" >Update</button>
<div>
    <city-cust [(myCity)] ="city"> </city-cust>
    {{city}}
</div>
```

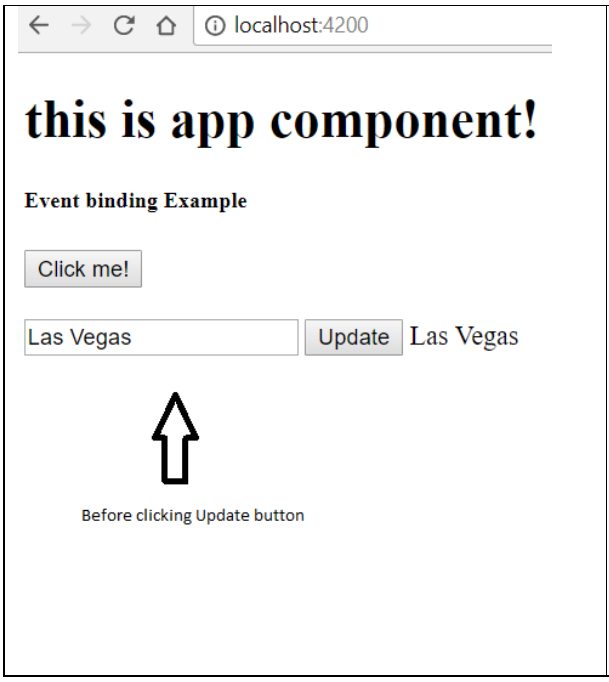
So lets look at the parent component - cityparent.component.ts that calls the child component event.

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

```
@Component({
    selector: 'city-parent',
    templateUrl: './cityparent.component.html'
})
export class CityParentComponent {
    //Property for MsgComponent and
AliasComponent
    msg = 'Hello --------';

    //Property for AliasComponent
    city = 'Las Vegas';
    }
```

The cityparent.component.html includes the markup as below.

```
<div>
    <city-cust [(myCity)] ="city"> </city-cust>
    {{city}}
</div>
```


localhost:4200

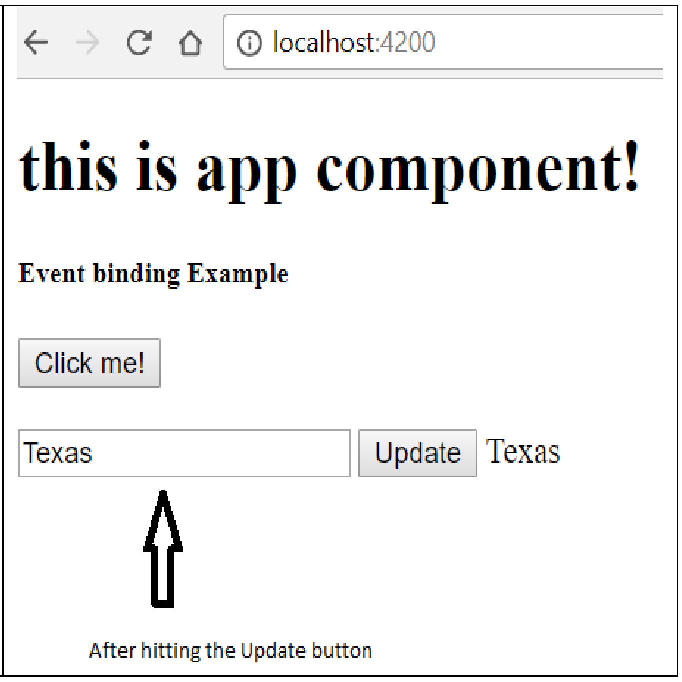
this is app component!

Event binding Example

Click me!

Las Vegas Update Las Vegas

Before clicking Update button


localhost:4200

this is app component!

Event binding Example

Click me!

Texas Update Texas

After hitting the Update button

Creating Forms in Angular

Forms are the mainstay of business applications. You use forms to log in, submit a help request, place an order, book a flight, schedule a meeting, and perform countless other data-entry tasks.

In developing a form, it's important to create a data-entry experience that guides the user efficiently and effectively through the workflow.

Template-driven forms

You can build forms by writing templates in the Angular template syntax with the form-specific directives and techniques described in this page.

To create HTML form using NgFormwith NgModel is called template-driven form. NgForm directive is used with HTML form tag that can be exported in local template variable to access form values and validation status and to pass entire form to our class on form submit. NgModel is used in form fields and can be exported in local template variable to get its values and validation status in HTML template. When we create HTML form, we have to use name attribute in form fields. In the context of NgForm, the role of NgModel is to register the form field with form using name attribute. NgModel is also used to bind the domain model of the class with view. On this page we will provide how to enable using NgForm in our angular application. We will provide how to use NgForm with form tag. We will create some form fields within form and use NgModel with them. We will also provide using one-way and two-way binding with NgModel within the form tag.

NgForm Directive

- NgForm directive is used with HTML <form> tag. To understand NgForm, we should be aware with following classes.
- FormControl : This class tracks the value and validation status of an individual form control.
- FormGroup : This class tracks the value and validity state

of a group of FormControl.

Now let us understand NgForm directive. NgForm is a directive that creates a top-level FormGroup instance and binds it to a form to track aggregate form value and validation status.

Find the steps to use NgForm in our application.

1) Configure FormsModule in imports metadata of @NgModule decorator in application module as given below.

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

@NgModule({
---------------
---------------
  imports: [
        BrowserModule,
    FormsModule
  ]
---------------
---------------
})
```

2) Once we import FormsModule, NgForm directive becomes active by default on all <form> tags. We can export NgForm into a local template variable. To import it we need to use NgForm as ngForm. Find the sample example.

```
<form #userForm="ngForm"
(ngSubmit)="onFormSubmit(userForm)">
----------------
----------------
<button>Submit</button>
</form>
```

To submit the form we can use <button> whose type is submit by default for the entire modern browser.

Exporting NgForm into local template variable is optional but useful. In the above code snippet, look into #userForm="ngForm". Here local template variable is userForm.

We use local template variable to submit form data. Look into

(ngSubmit)="onFormSubmit(userForm)"

Here when form is submitted, local template variable is passed that can be accessed in class as follows.

```
onFormSubmit(userForm: NgForm) {
    console.log(userForm.value);
    console.log('Name:' +
userForm.controls['name'].value);
    console.log('Form Valid:' +
userForm.valid);
    console.log('Form Submitted:' +
userForm.submitted);
} Using controls[] of NgForm, we can access the
form field value by field name in our class as
following.
userForm.controls['name'].value;
userForm.controls['city'].value;
userForm.controls['state'].value Here name,
city, state are the field names.
```

3) To reset form, NgForm has resetForm() method that is called as follows.

```
  resetUserForm(userForm: NgForm) {
     userForm.resetForm();;
  } To call the above function, create a button
in UI.
<button type="button"
(click)="resetUserForm(userForm)">Reset</button
> If we want to reset form with some default
values, then assign the default values with
form control name of the form.
resetUserForm(userForm: NgForm) {
   userForm.resetForm({
      name: Ganesha,
      city: California
   });
}
```

NgModel Directive

NgModel directive creates a FormControl instance from a domain model created in class and binds it to a form control element. FormControl keeps the track of user interaction and validation status of the form control. NgModel can be used standalone as well as with the parent form. When it is used as standalone, we can use it as follows.

```
<input [(ngModel)]="name" required
#userName="ngModel">

<p>Name value: {{ name }} </p>
<p>Name value: {{ userName.value }} </p>
<p>Name valid: {{ userName.valid }} </p> Now
look into #userName="ngModel".
```

Here userName is local template variable that can be used to get form control value, valid, invalid, pending etc properties in HTML template.

NgModel can be used as one-way binding and two-way binding. In one way binding using [ ] sign, when value changes in domain model in the class then that is set to view also. In two-way binding using [( )] sign, the value changes in domain model is set to view and values changes in view is also set to domain model.

NgForm with NgModel

We will use here NgForm with NgModel directive. When we use ngModel with <form> tag, we have also to supply a name attribute to register a control with parent form using that name.

When we are using parent form, we need not to use two-way binding to get form fields values in our class. We can pass <form> local template variable such as userForm to our submit method such as onFormSubmit(userForm) as given below.

```
<form #userForm="ngForm"
(ngSubmit)="onFormSubmit(userForm)"> On form
submit, the form values can be accessed in
class as given below.
onFormSubmit(userForm: NgForm) {
    console.log(userForm.value);
    console.log('Name:' +
userForm.controls['name'].value);
}
```

If we use neither one-way binding not two-way binding still we need to use ngModel attribute in fields when using with parent form as given below.

```
<input name="message" ngModel>
If we will not use ngModel attribute in fields
such as
<input name="message" >
```

Then this field will not be registered with parent form and its value will not be passed to our class on form submit.

Lets look at the sample examples.

Case-1: Here we are using two-way binding in parent form.

```
<form #userForm="ngForm"
(ngSubmit)="onFormSubmit(userForm)">
  <input name="message" [(ngModel)] = "msg">
  <button>Submit</button>
</form>
```

If we want form fields values using domain model then we can use two-way binding. On form submit we can also access form values using NgForm in our class. In case of parent form, most of the time, we need not to use two-way binding because we can access values in our class using NgForm itself. name attribute is required to register control with form.

- Case-2: Here we are using one-way binding in parent form.

```
<form #userForm="ngForm"
(ngSubmit)="onFormSubmit(userForm)">
  <input name="message" [ngModel] = "msg">
  <button>Submit</button>
</form>
```

We should use one-way binding in the scenario where we want to pre-populate form fields. name attribute is required to register control with form.

- Case-3: Here we are using neither one-way nor two way binding.

```
<form #userForm="ngForm"
(ngSubmit)="onFormSubmit(userForm)">
  <input name="message" ngModel>
  <button>Submit</button>
</form>
```

To register child control with the form, we need to use ngModel as field attribute. If we are using neither one-way nor two way binding, still we need to use ngModel attribute so that the field can be registered with parent form otherwise the field value will not be passed on form submit. name attribute is required to register control with form.

- Case-4: Here we are not using ngModel in any way.

```
<form #userForm="ngForm"
(ngSubmit)="onFormSubmit(userForm)">
  <input name="message">
  <button>Submit</button>
</form>
```

As we know that to register child control with the form, we need to use ngModel as field attribute. If we are not using ngModel as field attribute then that field will not be registered with parent form and hence the field value will not be passed on form submit to our class.

app.component.ts

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

@Component({
   selector: 'app-root',
   templateUrl: 'app.component.html'
})
export class AppComponent {
  onFormSubmit(userForm: NgForm) {
        console.log(userForm.value);
  console.log('Name:' +
userForm.controls['name'].value);
```

```
          console.log('Form Submitted:' +
userForm.submitted);
  }
  resetUserForm(userForm: NgForm) {
        userForm.resetForm();;
  }
}
```

app.component.html

```
<h3>Using NgForm with NgModel</h3>
<div>
   <form #userForm="ngForm"
(ngSubmit)="onFormSubmit(userForm)">
    <div>
     Name: <input name="name" ngModel required
ngModel>
    </div>
    <div>
     City: <input name="city" ngModel  gModel>
    </div>
    <div>
     State: <input name="state"  ngModel>
    </div>
    <div>
     <button>Submit</button>
          <button type="button"
(click)="resetUserForm(userForm)">Reset</button>
    </div>
   </form>
</div>
<div>
   <p>Form value: <b> {{ userForm.value | json
}} </b></p>

</div>
<div>
   <p>Name value: <b> {{ userName.value }}
</b></p>
</div>
```

```
<div>
   <p>City value: <b> {{ userCity.value }}
</b></p>
</div>
```

Angular Forms – Validations

The validators provided by Angular

- **required -** some input must be provided
- **minLength -** a number specifying the minimum length allowed
- **maxLength -** a number specifying the maximum length allowed
- **pattern -** a pattern (regex) that the input needs to follow

```
<div >
<label for="firstname">First Name</label>
<input type="text" name="firstname"
ngModel  #firstname="ngModel"  required
minlength="10" >
 <div style="color:red" *ngIf="!firstname?.valid
&& (firstname?.dirty ||firstname?.touched)"
class="alert alert-danger">
            <div
[hidden]="!firstname.errors.required">
                First Name is required
            </div>
            <div
[hidden]="!firstname.errors.minlength">
                First Name Minimum Length is

{{firstname.errors.minlength?.requiredLength}}
            </div>
        </div>
</div>
<div >
     <label for="pincode">Pin Code</label>
      <input type="text"  name="pincode"
     ngModel  #pincode="ngModel"  required
pattern="[0-9]{5}">
<div style="color:red" *ngIf="!pincode?.valid &&

 (pincode?.dirty ||pincode?.touched)" >
            <div [hidden]="!pincode.errors">
                Enter a valid pattern as : 5
digits only
            </div>
        </div>
</div>
```

Now lets see how to create Model Driven forms

What are Model Driven forms?

Angular reactive forms facilitate a reactive style of programming that favors explicit management of the data flowing between a non-UI data model (typically retrieved from a server) and a UI-oriented form model that retains the states and values of the HTML controls on screen. Reactive forms offer the ease of using reactive patterns, testing, and validation.

With reactive forms, you create a tree of Angular form control objects in the component class and bind them to native form control elements in the component template.

To create such form use formGroup, formGroupName and formControlName from the ReactiveFormsModule.

ReactiveFormsModule

To enable reactive form in our angular application we need to configure ng module ReactiveFormsModule in application module.

Reactive Form: Creating form using **FormControl** and **FormGroup** is said to be reactive form. They use ng module as ReactiveFormsModule.

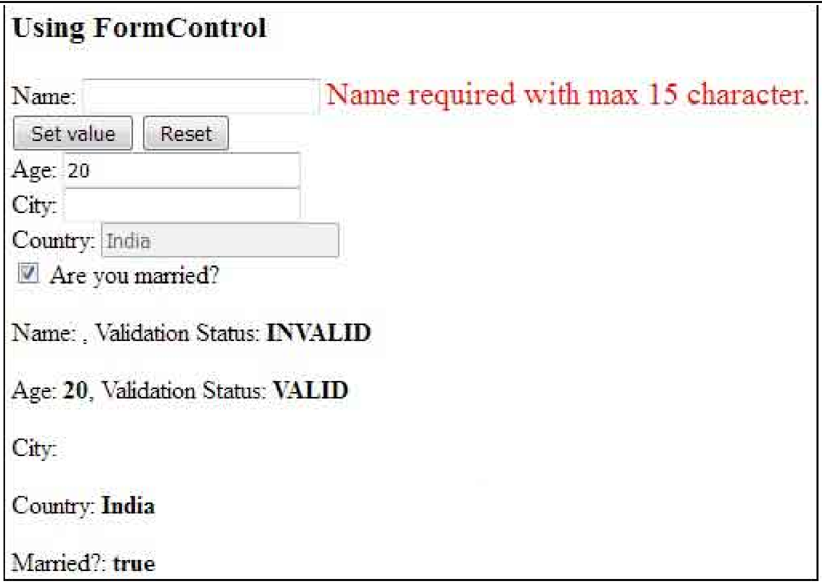
FormControl: It is a class that is used to get and set values and validation of a form control such as <input> and <select> tag.

FormGroup: It has the role to track value and validity state of group of FormControl.

```
import { Component } from '@angular/core';
import { FormControl, Validators } from
'@angular/forms';

@Component({
   selector: 'app-form-control',
   templateUrl: 'formcontrol.component.html',
   styleUrls: ['formcontrol.component.css']
})
export class FormControlDemoComponent {
  name = new FormControl('',
[Validators.required,
Validators.maxLength(15)]);
  age = new FormControl(20,
Validators.required);
  city = new FormControl();
  country = new FormControl({value: 'India',
disabled: true});
  married = new FormControl(true);

  setNameValue() {
      this.name.setValue('Ganesha  Omkara);
      console.log('Name: ' + this.name.value);
      console.log('Validation Status: ' +
this.name.status);
  }
  setResetName() {
      this.name.reset();
  }
  changeValue() {
      console.log(this.married.value);
      this.married = new
FormControl(!this.married.value);
  }
}

What will be the template for the above
component?
```

```
<h3>Using FormControl </h3>
<div>
 <div>
   Name: <input [formControl]="name">
         <label *ngIf="name.invalid" [ngClass] =
"'error'"> Name required with max 15 character.
</label>
 </div>
 <div>
   <button (click)="setNameValue()">Set
value</button>
   <button
(click)="setResetName()">Reset</button>
 </div>
 <div>
   Age: <input [formControl]="age">
 </div>
 <div>
   City: <input [formControl]="city">
 </div>
 <div>
   Country: <input [formControl]="country">
 </div>
 <div>
    <input type="checkbox"
[checked]="married.value"
(change)="changeValue()" />
   Are you married?
 </div>
</div>
<div>
 <p>Name: <b>{{ name.value }}</b>, Validation
Status: <b>{{ name.status }}</b></p>
 <p>Age: <b>{{ age.value }}</b>, Validation
Status: <b>{{ age.status }}</b></p>
 <p>City: <b>{{ city.value }}</b> </p>
 <p>Country: <b>{{ country.value }}</b> </p>
 <p>Married?: <b>{{ married.value }}</b> </p>
```

```
</div>
```



Name: Name required with max 15 character.
Age: 20
City:
Country: India
☑ Are you married?

Name: , Validation Status: **INVALID**

Age: **20**, Validation Status: **VALID**

City:

Country: **India**

Married?: **true**

FormControl with FormGroup using FormControlName

We will create form control with parent HTML <form> tag. All the form control which are the instances of **FormControl** class, will be grouped using FormGroup class. Find the code snippet.

```
userForm = new FormGroup({
  name: new FormControl('Mahesh',
Validators.maxLength(10)),
  age: new FormControl(20,
Validators.required),
  city: new FormControl(),
  country: new FormControl()
 }); The HTML code will be as given below.
<form [formGroup]="userForm"
(ngSubmit)="onFormSubmit()">
    Name: <input formControlName="name"
placeholder="Enter Name">
    Age: <input formControlName="age"
placeholder="Enter Age">
    City: <input formControlName="city"
placeholder="Enter City">
    Country: <input formControlName="country"
placeholder="Enter Country">
         <button type="submit">Submit</button>
 </form>
```

When the form will be submitted, the values can be accessed as explained below.

```
onFormSubmit(): void {
    console.log('Name:' +
this.userForm.get('name').value);
    console.log('Age:' +
this.userForm.get('age').value);
    console.log('City:' +
this.userForm.get('city').value);
    console.log('Country:' +
this.userForm.get('country').value);
}
```

To validate a particular form control, we need to get value of that form control. For example, to validate name FormControl, we need to create a method in our class as given below.

```
get userName(): any {
    return this.userForm.get('name');
} Now use it in HTML.
<div>
    Name: <input formControlName="name"
placeholder="Enter Name">
    <label *ngIf="userName.invalid" [ngClass] =
"'error'"> Name is too long. </label>
</div> We can also access FormControl value
using FormGroupDirective.
```

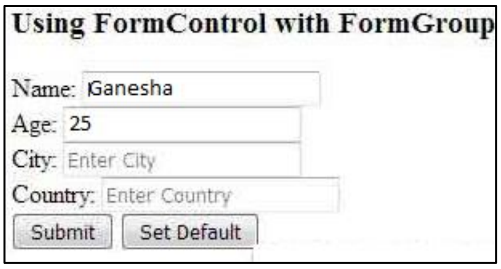
We have defined it as [formGroup]="userForm" in our form. Now we can use userForm to get values for validation as given below.

```
<label *ngIf=" userForm.get('name').invalid"
[ngClass] = "'error'"> Name is too long.
</label> Find the example.
```

Lets look at the component that make use of **FormControl and FormGroup** module to create form and validate the data as well.

```
import { Component } from '@angular/core';
import { FormControl, FormGroup, Validators }
from '@angular/forms';

@Component({
   selector: 'app-form-group',
   templateUrl: 'formgroup.component.html',
   styleUrls: ['formgroup.component.css']
})
export class FormGroupDemoComponent {
  usrNameChanges: string;
  usrNameStatus: string;
  userForm = new FormGroup({
  name: new FormControl('Ganesha',
Validators.maxLength(10)),
  age: new FormControl(25,
Validators.required),
  city: new FormControl(),
  country: new FormControl()
  });
  get userName(): any {
        return this.userForm.get('name');
  }
  onFormSubmit(): void {
  console.log('Name:' +
this.userForm.get('name').value);
  console.log('Age:' +
this.userForm.get('age').value);
  console.log('City:' +
this.userForm.get('city').value);
  console.log('Country:' +
this.userForm.get('country').value);
  }
  setDefaultValue() {
        this.userForm.setValue({name:
'Ganesha', age: 25, city: ' ', country: ' '});
  }
}
```

```
In the html file the below markup is to be
added to it.
<h3> Using FormControl with FormGroup </h3>
<div>
 <form [formGroup]="userForm"
(ngSubmit)="onFormSubmit()">
  <div>
    Name: <input formControlName="name"
placeholder="Enter Name">
  <label *ngIf="userName.invalid" [ngClass] =
"'error'"> Name is too long. </label>
  </div>
  <div>
    Age: <input formControlName="age"
placeholder="Enter Age">
  <label *ngIf="userForm.get('age').invalid"
[ngClass] = "'error'"> Age is required.
</label>
  </div>
  <div>
    City: <input formControlName="city"
placeholder="Enter City">
  </div>
  <div>
    Country: <input formControlName="country"
placeholder="Enter Country">
  </div>
  <div>
    <button type="submit">Submit</button>
    <button type="button" (click) =
"setDefaultValue()">Set Default</button>
  </div>
 </form>
```



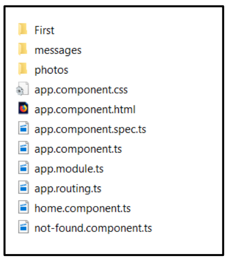
So folks!!! At the need of this chapter you have learnt how to work to create and process forms in Angular application.

Routing in Angular

Routing is a process of changing the state of the application by loading different components based on the request URL. It is used for Navigation based on the request URL. The Angular Router enables navigation from one view to the next as users perform application tasks.T he navigation is achieved by configuring routes in route configuration file-**app.routing.ts**. To work with routing features, import router module named ‘**RouterModule’** from **‘@angular/router’**.

To better understand routing lets look at the routing example.

Create the app folder with the structure as below.



First
messages
photos
app.component.css
app.component.html
app.component.spec.ts
app.component.ts
app.module.ts
app.routing.ts
home.component.ts
not-found.component.ts

Create home.component.ts as below

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

@Component({
  template: `<h1>{{message}}</h1>   `
})
export class HomeComponent
 {   message:string="Welcome to Home page...!!!"
 }
```

Create **notfound.component.ts** file as below

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

@Component({
  template: '<h1>{{notfoundTitle}}</h1>'
})
export class NotFoundComponent {
  notfoundTitle="OOps page Not
found.......!!!!";
}
```

Create **message.component.ts** as below

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

@Component({
    template: `<h1>{{messageTitle}}</h1>`
    })
export class MessagesComponent {
    messageTitle:string="This is Message page
......!!!!!";
}
```

Create **photos.component.ts** file as below/

```
import { Component } from '@angular/core';
@Component({
    template: `<h1>{{photosTitle}}</h1>`
})
export class PhotosComponent {
    photosTitle:string="This is Photos page
......!!!!!";
}
```

Create RouteConfiguration file - app.route.ts

Angular routing provides services, directives and operators that manage the routing and navigation in our application. The Angular Router is an optional service that presents a particular component view for a given URL. It is not part of the Angular core. It is in its own library package, @angular/router. Import what you need from it as you would from any other Angular package.

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

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

```
import { Router,RouterModule } from
'@angular/router';
import { HomeComponent } from
'./home.component';
```

```
import { MessagesComponent } from
'./messages/messages.component';
import { PhotosComponent } from
'./photos/photos.component';
import { NotFoundComponent } from './not-
found.component';

export const routing=RouterModule.forRoot([
{path:'',component:HomeComponent},
{path:'messages',component:MessagesComponent},
{path:'photos',component:PhotosComponent},
{path:'**',component:NotFoundComponent}
]);
```

Routing variable defines the array of roots that map a path to a component. Paths are configured in module to make available it globally.

Each Route maps a URL path to a component.

The empty path in the fourth route represents the default path for the application, the place to go when the path in the URL is empty, as it typically is at the start. This default route redirects to the route for the /home URL and, therefore, will display the HomeComponent.

The ** path in the last route is a wildcard. The router will select this route if the requested URL doesn't match any paths for routes defined earlier in the configuration. This is useful for displaying a "404 - Not Found" page or redirecting to another route.

In the app.component.html as below

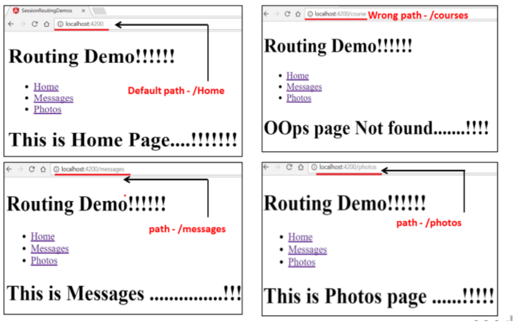
```
<h1>  {{title}}</h1>
<ul>
  <li><a routerLink="">Home</a></li>
  <li><a
routerLink="messages">Messages</a></li>
  <li><a routerLink="photos">Photos</a></li>
  </ul>
   <router-outlet></router-outlet>
```

```
    <br/>
```

- **RouterLink -**The **RouterLink** directives on the anchor tags give the router control over those elements. The navigation paths are fixed, so you can assign a string to the routerLink (a "one-time" binding).
- **Router Outlet** -After configuring our routes, the next step is to tell Angular where to load the components using a directive named router-outlet. When the router has matched a route and found the component(s) to load, it will dynamically create our component and inject it as a sibling alongside the router-outlet element.   The **<router-outlet>** is the placeholder where the component is inserted.

In the **app.module.ts** file import the ‘routing module and register the all of the above defined components.

The result will be as below.



Routing Demo!!!!!!

- Home
- Messages
- Photos

This is Home Page....!!!!!!!

Routing Demo!!!!!!

- Home
- Messages
- Photos

OOps page Not found.......!!!!

Routing Demo!!!!!!

- Home
- Messages
- Photos

This is Messages ....!!!

Routing Demo!!!!!!

- Home
- Messages
- Photos

This is Photos page ......!!!!!

How to set and Fetch Route Parameters?

While we don't need any type of query parameters for our particular project, we're going to create one just so that I can show you how you can work in parameters to your URL's and retrieve the values if need be.

```
Consider the below given example of route
configuration file:
  {
    path: 'about/:id',             // Add /:id
here
    component: AboutComponent
  }
```

This is designating that anything after about/ will be referred to as id, which is a route parameter.

Then in the app.component.html add the following markup:

```
  <li><a routerLink="about/48">About</a></li>
```

In the about.component.ts and import ActivatedRoute:

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

import { ActivatedRoute } from '@angular/router';   // Add this

This will enable accessing the route parameters. Next, create an instance of ActivatedRoute through dependency injection, which is done in the constructor of the class:

```
export class AboutComponent implements OnInit {
    constructor(private route: ActivatedRoute)
{
    this.route.params.subscribe(res =>
console.log(res.id));
  }
  ngOnInit() {   }
}
```

The above code retrieves the route parameters and prints it to the console.

That’s all folks!!!

So folks!!! At the need of this chapter you have learnt how to implement routing in Angular application.

Services in Angular

Every application is composed of a lot of subsystems. These sub systems perform different tasks like logging, data access, caching, specific application domain knowledge, etc. Angular helps to create such sub systems by allowing the developers to create services. An Angular service is a class that encapsulates some sort of functionality and provides this functionality via a service for the rest of the application.

Lets see how to create a service in Angular.

```
import { Injectable } from '@angular/core';
@Injectable()
export class  HelloService {

  public  sayHello() {
        return "Welcome to the world of angular
services....!!!";
    }
}
```

@Injectable() decorator is a marker used at class level. It tells Injector that this class is available for creation by Injector. We use @Injectable() in our service class so that the service object can be created automatically for dependency injection in any component or any other service class. @Injectable() is also responsible to instantiate an angular component, pipe, directive etc. This becomes possible because @Component, @Pipe and @Directive decorators use @Injectable decorator. If our service class decorated with @Injectable() has dependency of other service classes, then we need not to worry for its creation and dependency injection will be performed by Injector automatically.

How to consume service

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

@Component({
  selector: 'app-test',
  template: `
  <p>click button to invoke service</p>
  <input type="button"  (click)="display()"
value="Click here"><br/><br/>
  <Div *ngIf="flag">{{msg}}</Div> `,
  providers: [HelloService]
})
export class TestComponent  {
msg:string="";
flag:boolean=false;
constructor(private helloservice:HelloService)
   {     }
public display()
  { this.flag=true;
    this.msg=this.helloservice.sayHello();
  }
}
```

A service can be injected in another service as given below

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

@Injectable()
export class OtherService {
    public GetMyText() {
        return "Text From Other Service";
    }
}



import { Injectable } from '@angular/core';
import { OtherService } from './OtherService';

@Injectable()
export class MyService {
    constructor(private
otherService:OtherService) {
         }
     public GetTextFromOtherService() {
        return "Service called from Myservice-"
+ this.otherService.GetMyText();
    }
}
```

Working with Http Service

- Angular HTTP library provides Http client for server communication.
- .get() is the Http client method that uses HTTP GET method to communicate server using HTTP URL.
- Http.get() include the Url and
- it returns the instance of **RxJS Observable** that can be later subscribed to obtain result.
- Observable values can be directly fetched in HTML template using async pipe.
- Observable is an asynchronous pattern.
- It is an asynchronous stream of data to
- In the Observable pattern we have an Observable and an Observer.
- Observer observes the Observable.
- In many implementations an Observer is also called as a Subscriber.
- An Observable can have many Observers (also called Subscribers).
- Observable emits items or notifications over time to which an Observer (also called Subscriber) can subscribe.
- Observables help to manage asynchronous data, such as data coming from a backend service.
- To use observables, Angular uses a third-party library called Reactive Extensions (**RxJS**).

Angular HTTP library provides Http client for server communication. get() is the Http client method that uses HTTP GET method to communicate server using HTTP URL. We need to pass HTTP URL to Http.get() and it will return the instance of RxJS Observable. To listen values of Observable we need to subscribe it. We can also directly fetch Observable value in our

HTML template using async pipe. Observable can be converted into Promise using RxJS toPromise() method. Http.get() communicates to server only when we fetch values from Observable or Promise.

Http.get()

Http.get() performs a request with HTTP GET method. The syntax is as given below.

get(url: string, options?: RequestOptionsArgs) : Observable<Response> get() is the method of angular Http API that interacts with server using HTTP GET method. It accepts a HTTP URL and returns Observable instance. RequestOptionsArgs is optional. To use HTTP get(), we need to follow below steps.

First step is that we need to import HttpModule in @NgModule using imports metadata in our application module.

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

@NgModule({
---------
---------
  imports: [
        BrowserModule,
  HttpModule
  ]
---------
---------
})
```

- **Step-2::** We should perform server communication in a service class and not in component. This approach is preferable by design. In a service class we will use dependency injection to get the instance of angular Http as given below.

```
constructor(private http:Http) { } Step-3::
Pass URL to use http.get().
```

Observable<Response> ob = this.http.get(this.url); http.get() returns instance of Observable that can be later subscribed to get result. We can also fetch Observable directly in HTML template using async pipe.

Observable

We can use either Observable or Promise with http.get().

Observable: This is a RxJS API. Observable is a representation of any set of values over any amount of time. All angular Http methods return Observable. Observable provides methods such as map(), catch() etc. map() applies a function to each value emitted by source Observable and returns finally an instance of Observable. catch() is called when an error is occurred. catch() also returns Observable.

```
Http.get with Observable
Http.get() returns instance of
Observable<Response>. To change it into
instance of Observable<Book[]>, we need to use
RxJS map() operator.
getBooksWithObservable(): Observable<Book[]> {
    return this.http.get(this.url)
        .map(this.extractData);
}
The extractData() method. map() converts
Response object of http.get into Book.
private extractData(res: Response) {
    let body = res.json();
    return body;
} T
```

To display Observable in our HTML template we can go by two way.

1) Angular directive can directly use Observable with async pipe. Find the code snippet that we will write in component.

```
observableBooks: Observable<Book[]>
this.observableBooks =
this.bookService.getBooksWithObservable();
Now iterate observableBooks with ngFor as
follows.
<ul>
  <li *ngFor="let book of observableBooks |
async" >
    Id: {{book.id}}, Name: {{book.name}}
  </li>
</ul> async pipe will mark observableBooks to
listen for any changes.
```

2) Use the below code snippet to Subscribe to the Observable and fetch values stored by it.

```
observableBooks: Observable<Book[]>
books: Book[];
this.observableBooks =
this.bookService.getBooksWithObservable();
this.observableBooks.subscribe(
      books => this.books = books,
      error =>  this.errorMessage =
<any>error);
Now iterate books with ngFor as follows.
<ul>
  <li *ngFor="let book of books" >
    Id: {{book.id}}, Name: {{book.name}}
  </li>
</ul>
Create the service as
import { Injectable } from '@angular/core';
import { Http, Response } from
'@angular/http';
import { Observable } from 'rxjs';
import 'rxjs/add/operator/map';
import { Book } from './book';

@Injectable()
export class BookService {
    url = "http://localhost:4200/
data/books.json";
    constructor(private http:Http) { }
    getBooksWithObservable():
Observable<Book[]> {
        return this.http.get(this.url)
            .map(this.extractData);
    }
    private extractData(res: Response) {
    let body = res.json();
        return body;
    }
}
```

book.ts

```
export class Book {
   id: number;
   name: string;
   constructor() {
   }
}
```

books.json

```
[
  {"id": 1, "name": "PHP"},
  {"id": 2, "name": "MS Azure"},
  {"id": 3, "name": "MongoBD"}
]
```

Create the component that makes use of the service

```
import { Component, OnInit } from
'@angular/core';
import { Observable } from 'rxjs';
import { BookService } from './book.service';
import { Book } from './book';

@Component({
   selector: 'app-observable',
   templateUrl: './observable.component.html'
providers: [  BookService ],
})
export class ObservableComponent implements
OnInit {
   observableBooks: Observable<Book[]>
   books: Book[];
   errorMessage: String;
   constructor(private bookService:
BookService) { }
   ngOnInit(): void {
        this.observableBooks =
this.bookService.getBooksWithObservable();
  this.observableBooks.subscribe(
            books => this.books = books,
            error =>  this.errorMessage =
<any>error);
   }
}
```

Add the below mark-up in the template file

```
<h3>Book Details using Observable</h3>
<ul>
  <li *ngFor="let book of observableBooks |
async" >
    Id: {{book.id}}, Name: {{book.name}}
  </li>
</ul>
<h3>Book Details obtained by  "subscribe" to
the Observable </h3>
<ul>
  <li *ngFor="let book of books" >
    Id: {{book.id}}, Name: {{book.name}}
  </li>
```

```
</ul>
<div *ngIf="errorMessage"> {{errorMessage}}
</div>
```

Working with Firebase service

What is Firebase?

The Firebase Realtime Database is a cloud-hosted database. Data is stored as JSON and synchronized in realtime to every connected client.

Firebase Realtime Database - Store and sync data with our NoSQL cloud database. Data is synced across all clients in realtime, and remains available when your app goes offline.

When you build cross-platform apps with our iOS, Android, and JavaScript SDKs, all of your clients share one Realtime Database instance and automatically receive updates with the newest data.

The Firebase Realtime Database lets you build rich, collaborative applications by allowing secure access to the database directly from client-side code. Data is persisted locally, and even while offline, realtime events continue to fire, giving the end user a responsive experience. When the device regains connection, the Realtime Database synchronizes the local data changes with the remote updates that occurred while the client was offline, merging any conflicts automatically.

The Realtime Database is a NoSQL database and as such has different optimizations and functionality compared to a relational database. The Realtime Database API is designed to only allow operations that can be executed quickly. This enables you to build a great realtime experience that can serve millions of users without compromising on responsiveness.

```
 import { Injectable } from '@angular/core';
import { Http, Response, Headers } from
"@angular/http";
import 'rxjs/Rx';
import { Observable } from "rxjs/Rx";
```

```
@Injectable()
export class HttpService {
  constructor(private http: Http) {
  }
  getData() {
    return this.http.get('https://<<service
name>>.firebaseio.com/data.json')
      .map((response: Response) =>
response.json());
  }
}
```

How to invoke the service?

```
import { Component } from '@angular/core';
import { HttpService } from "./http.service";

@Component({
  moduleId: module.id,
  selector: 'myhttp-app',
  templateUrl: './httpapp.component.html'
})
export class HttpAppComponent {
  items: any[] = [];
 asyncString = this.httpService.getData();
  constructor(private httpService: HttpService)
{}
showdata=false;
onGetData()
{
    this.showdata=true;
    this.httpService.getOwnData()
      .subscribe(
        data => {
          const myArray = [];
          for (let key in data) {
            myArray.push(data[key]);
          }
          this.items = myArray;
        }
      );
  }
}
Add the below markup to print the data.
<button (click)="onGetData()">Get Data</button>
<div *ngIf="showdata">
  <table border="1">
  <tr><td>IName</td>
  <td>Email</td>
  </tr>
     <tr *ngFor ="let item of items">
      <td>{{item.username}}</td>
  <td>{{item.email}}</td>
  </tr>
</table>
  </div>
```

So folks!!! At the need of this chapter you have learnt to create and consume service in Angular. Additionally you have also learnt how to work use Http service to invoke remote services.

Angular Lab Manual

Lab1: Introduction to Angular

What is SPA? List the various frameworks used for SPA?

What is Angular?

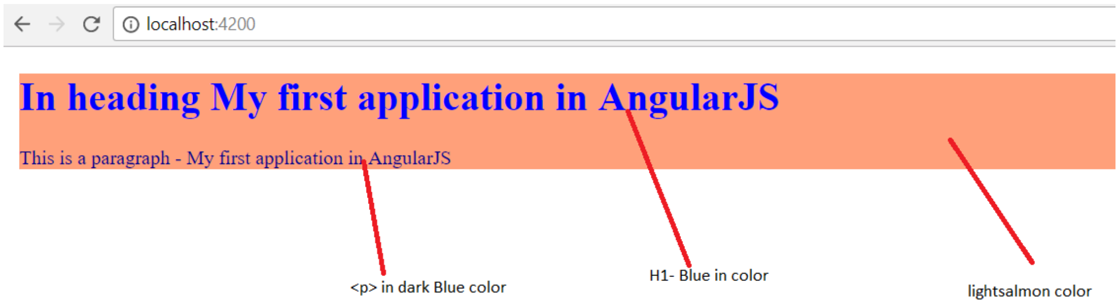
What are the building blocks of Angular?

Set up development environment for working with Angular applications

a) Install node.js
b) Install AngCLI
c) Install VS Code
d) Create a typescript program that explores various features such as types, interface, inheritance, compile and execute it.

Lab2: Components and Modules.

1) What are Module and components in Angular application?
2) Which is the root module of the Angular application?
3) Which decorator is used in module? List its metadata properties with the use of each?
4) Which decorator is used in component? List its metadata properties with the use of each?
5) State whether True or false – Module and components can be nested? State the real life scenario for each
6) Create an angular application that displays the below output by using
   e) template and styles
   f) templateUrl and styleUrls



In heading My first application in AngularJS

This is a paragraph - My first application in AngularJS

7) Create an angular application that includes

g) A new module other than the existing root module. (app.module.ts). use it via the root module.

h) Create a component that displays the below output in a div tag by creating a Component that uses a type. Also create the necessary css for the component and render the output based on css rules.

Employee Name	Amayra Parker
Gender	Female
Contact	111-223-431

i) Use these above module and component via root module.

8) Create an angular application that demonstrates the use of nested components. The output should render the header, Body and the footer. Header should include Contoso Compnay Ltd. The footer should include copyrights@Constos US. The central body portion should include some static list of Customers.

Lab3: Directives and Pipes

1) What are Directives? List the various types of directives in Angular? Define each of them with example.
2) Create an angular application that demonstrates the use of displays the list of Course in tabular format. (Course details such as – CourseName, Technology, fees, Duration.
3) Create an angular application that demonstrates the use of built-in attribute directive such as ng-style and ng-class .
4) Create a custom attribute directive that named as myRedAttrDirective directive. When applied to HTML element such as <p> and <div>, the text color within that element should appear red in color.
5) Modify the above assignment to apply the color and font size to the HTML elements based on the values supplied to the input properties .

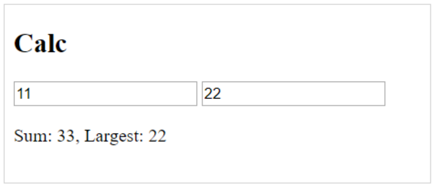
6) Create a simple custom pipe named as **welcome**. Example as follows.
7) {{'Seed Infotech Ltd' | welcome }} where welcome is the name of the pipe . it should produce the output as below
8) Seed Infotech Ltd to the World of Angular Application development
9) Create a custom pipe that named **Thestrformat** which will format the given string expression. The spaces between words will be replaced by given separator.
10) {{'Seed Infotech Ltd '| strformat:'-'}} should produce 'Seed-Infotech-Ltd'

Lab4: Event Handling & Data Binding

1) Create an angular app to demonstrate one-way data binding in Angular. It should display the following output using interpolation.(Hint : define and use employee type here. Use templateUrl and styleUrls properties of a component to achieve desired output).

Employee Name	Amayra Parker
Gender	Female
Contact	111-223-431

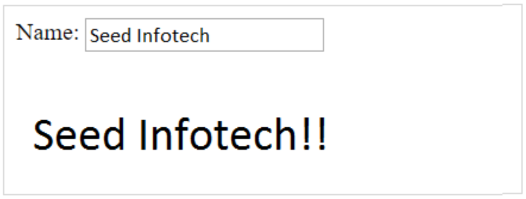
2) Create a Simple Interest calculator form. Display the S.I on the form. (Note : formula for SI=principal amount* rate of intrest * Time in years
3) Create the angular Application that displays the output as shown below.


Calc

11 | 22

Sum: 33, Largest: 22

4) Create an angular app to demonstrate two-way data binding in Angular.(Use templateUrl and styleUrls properties of a component to achieve desired output. Solve it using $event.target.value & [(ngModel)]

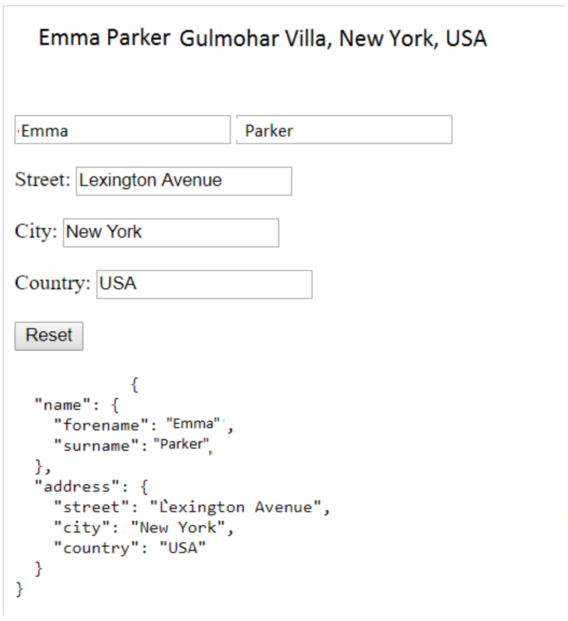

Name: Seed Infotech

Seed Infotech!!

5) Modify the above assignment no. 3 to demonstrate the use of event handling. That is add a button Calculate and Reset. The calculate value should be displayed when the Calculate button is clicked while the values should be reset when the Reset button is clicked.

6) Modify the above assignment no. 3 to demonstrate the use of eventemitter.

7) Create an angular application that displays the form output as shown below.


Emma Parker Gulmohar Villa, New York, USA

Emma | Parker

Street: Lexington Avenue

City: New York

Country: USA

```
{
  "name": {
    "forename": "Emma",
    "surname": "Parker",
  },
  "address": {
    "street": "Lexington Avenue",
    "city": "New York",
    "country": "USA"
  }
}
```

8) Create an angular application that demonstrates of style binding in Angular. Write the necessary code to perform the below tasks

j) sets a single style (font-weight). If the property **'isBold'** is true, then font-weight style is set to bold else normal.

sets font-size to 30 pixels.

k) Set multiple inline styles such as font-weight, font-style and font-size.

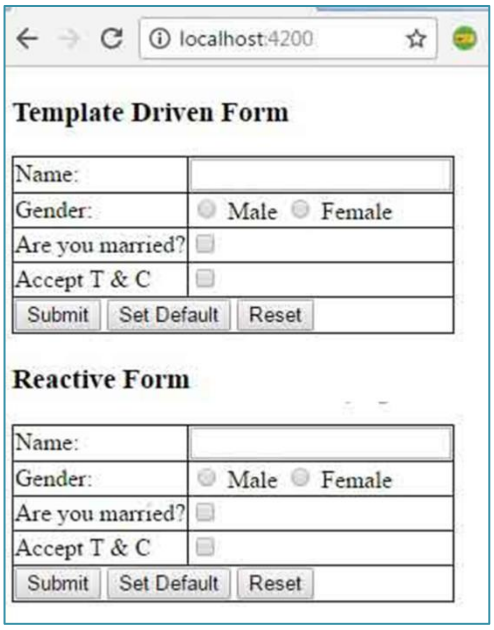
Lab 5: Angular Forms

9) Create an angular application that demonstrates

l) how to create Template Driven forms and

m) Model Driven forms. Display the data on the form in JSON format.

(Note: The design of the form is as shown in the below snapshot.)



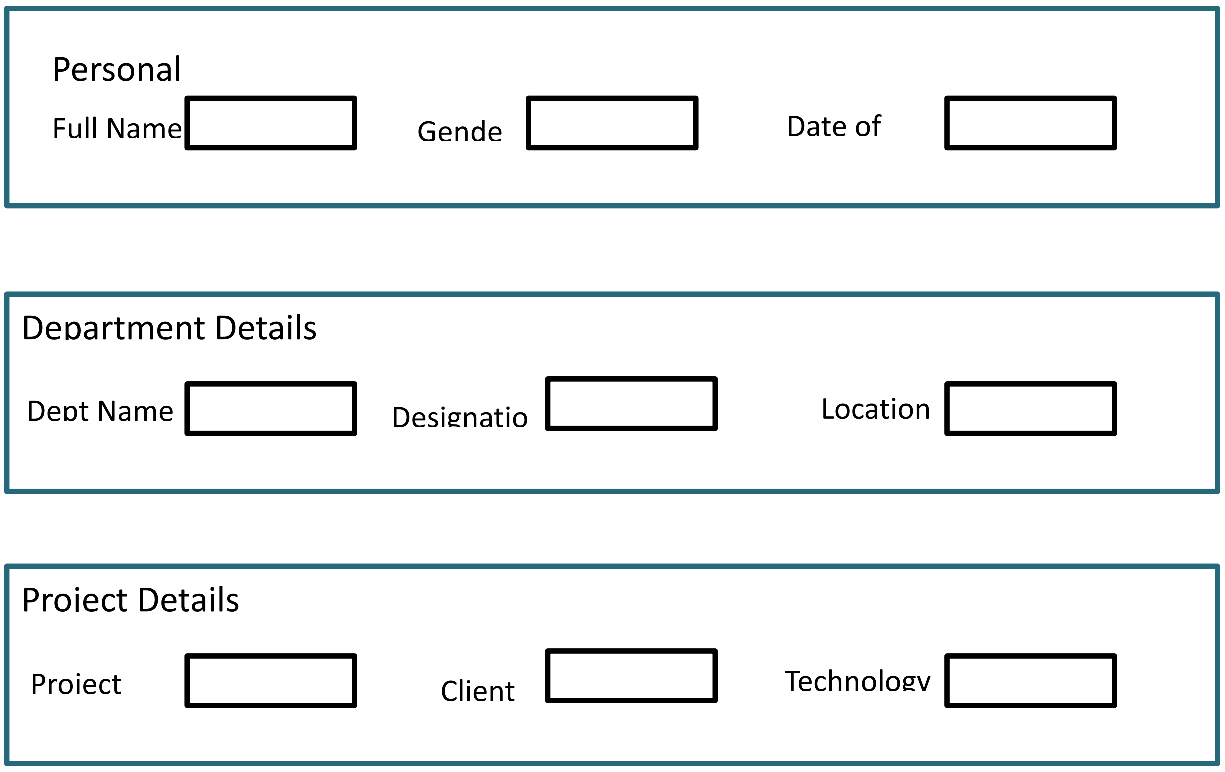
Template Driven Form

Name:	
Gender:	○ Male ○ Female
Are you married?	☐
Accept T & C	☐
Submit Set Default Reset	

Reactive Form

Name:	
Gender:	○ Male ○ Female
Are you married?	☐
Accept T & C	☐
Submit Set Default Reset	

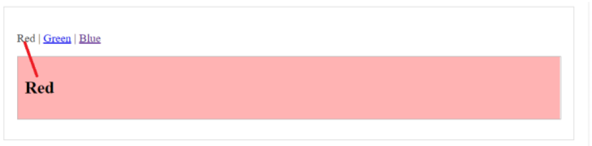
10) Design the model drive form as shown in the below snapshot. (Note – define type for each section on the form and use it to create Model driven form.)

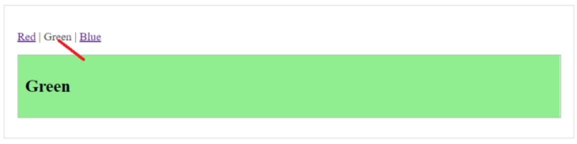

Personal
Full Name
Gende
Date of
Department Details
Dept Name
Designatio
Location
Project Details
Project
Client
Technology

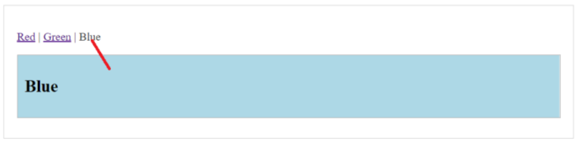
Lab 6: Routing

1) Create a angular application that includes the Main menu with the items such as ‘Home’, ‘Courses’ , ‘Services’ and ‘About Us’.
2) Extend the above assignment to create child routes as explained below.
   a) The Courses menu item should have ‘Cloud’, ‘Big Data’, ‘Web Development’ and ‘Data Analytics’ as its sub menu items.
   b) The Services menu item should have ‘Training, ‘Development’, ‘Staffing’ and ‘Networking’ as its sub menu items. When You hit any of the courses the message should be displayed as shown in the below example.
   c) ‘You selected the cloud Course’ or ‘You selected the BigData Course’
   d) When the user hits the link ‘About Us’ the following details should be displayed.

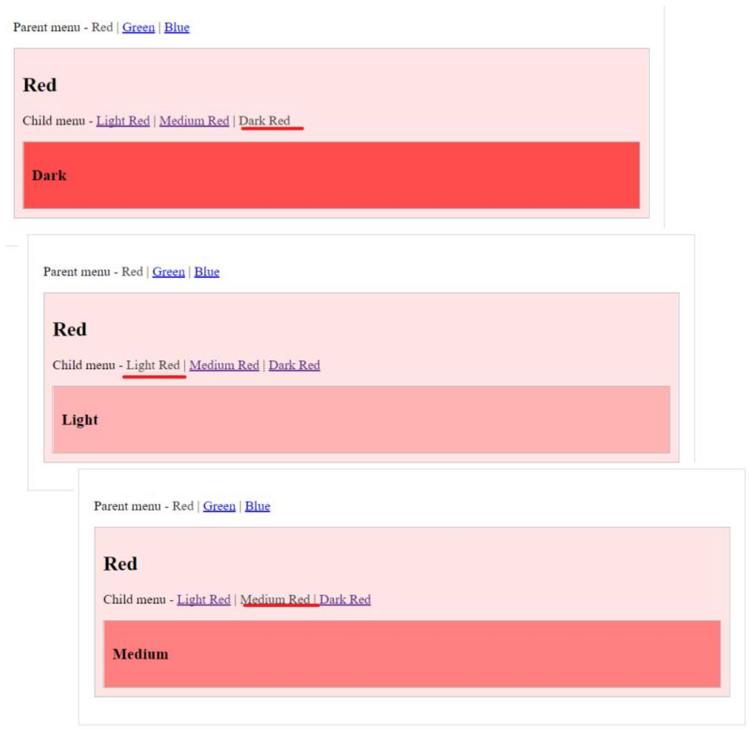
      5) Seed Infotech Ltd
         Income tax Lane , Karve Road
         Pune-Maharashtra
3) Design a menu as shown in the below snapshot







4) Modify the above menu to include child routes as shown in the below snapshot.



Parent menu - Red | Green | Blue

Red

Child menu - Light Red | Medium Red | Dark Red

Dark

Parent menu - Red | Green | Blue

Red

Child menu - Light Red | Medium Red | Dark Red

Light

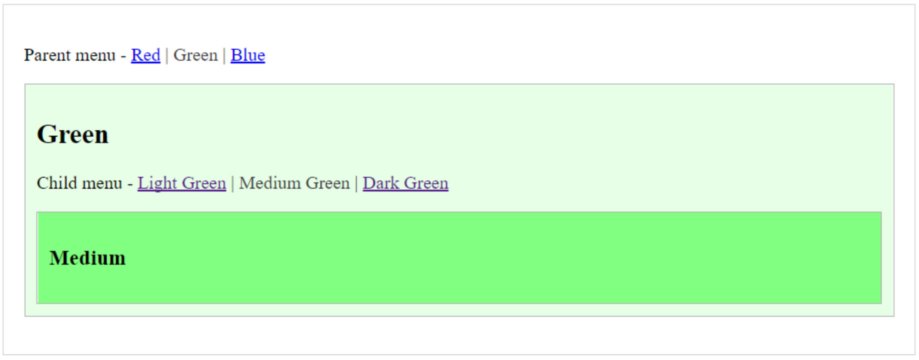
Parent menu - Red | Green | Blue

Red

Child menu - Light Red | Medium Red | Dark Red

Medium

5) Similarly Menu item Green should have child routes as below



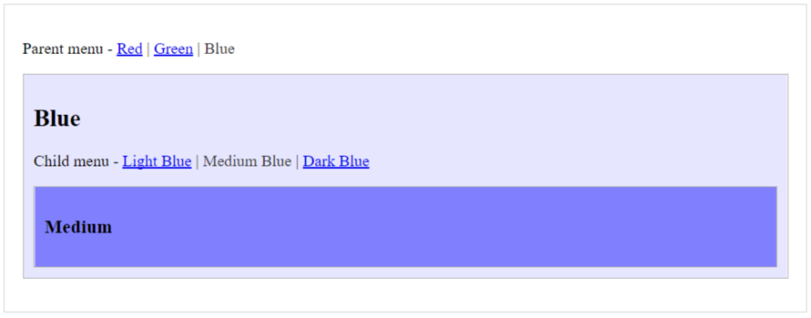
Parent menu - Red | Green | Blue

Green

Child menu - Light Green | Medium Green | Dark Green

Medium

6) Similarly the Blue menu item should have child routes as shown below



Parent menu - Red | Green | Blue

Blue

Child menu - Light Blue | Medium Blue | Dark Blue

Medium

Lab 7: Angular Services and Http

1) Create an Angular Service that returns the list of Sports. Create a component that consumes this service. The component should display the List of sports.
2) Create an Angular application that read and writes JSON data using Firebase service.

Lab 8: Angular testing

1) Perform unit testing of of application created in assignment 6 of lab2 using Jasmine and karma.

SEED Infotech Ltd.

'Panchasheel', 42/16, Erandawana, SEED Infotech Lane,
Off Karve Road, Pune - 411 004 Tel.: 020 25467484 / 92255 20000
E-mail : info@seedinfotech.com

seed®
beyond the obvious

www.seedinfotech.com