



PRESENTED BY: DEBASIS SAHA

CONTENTS

| | |
|--|----|
| Contents..... | 1 |
| Chapter 1: The Basics of Angular 4.0 | 5 |
| What is Angular JS? | 5 |
| General Features of Angular Js | 6 |
| Why Angular JS Known as Framework? | 6 |
| Architecture Concept | 6 |
| Advantages of Angular JS | 7 |
| What's New in Angular 4.0 In Compare Angular JS? | 8 |
| What is TypeScript?..... | 9 |
| How to Install Typescript?..... | 9 |
| Chapter 2 : First Program in Angular 4..... | 11 |
| tsconfig.json | 11 |
| package.json..... | 12 |
| system.config.js..... | 14 |
| First Program in Angular 4.0..... | 16 |
| Chapter 3 : Component – An Overview | 22 |
| What is a Component?..... | 22 |
| Advantages of Component Based Framework..... | 22 |
| Component Configuration..... | 24 |
| Life Cycles of the Component | 25 |
| Angular Component Architecture Layout | 25 |
| Create Angular Module | 25 |
| Create Angular Component | 26 |
| Add Component to Module | 26 |
| Bootstrap the Module..... | 26 |
| Bootstrap the Component | 27 |
| Sample Program | 27 |
| Chapter 4 : Data Binding | 33 |
| What is Data Binding ? | 33 |
| Why Data Binding is Required? | 33 |
| What is Dirty Checking? | 35 |

| | |
|---------------------------------------|----|
| Different Types of Data Binding | 35 |
| Interpolation | 35 |
| Property Binding | 36 |
| Two Way Binding..... | 38 |
| Event Binding | 39 |
| What is ngModel..... | 41 |
| Component Heirarchy | 41 |
| @Input() Decorator | 42 |
| @Output() Decorator..... | 43 |
| Chapter 5 : Directives | 48 |
| Component Vs Directives | 48 |
| Attribute Directives | 49 |
| ElementRef..... | 49 |
| Renderer..... | 50 |
| HostListener | 50 |
| NgStyle Directive..... | 50 |
| NgClass Directive..... | 51 |
| Structural Directives..... | 55 |
| ngIf | 55 |
| ngFor | 57 |
| ngSwitch..... | 58 |
| Chapter 6 : Pipes and ViewChild | 62 |
| What is Pipes? | 62 |
| Transition From Filter to Pipes | 62 |
| Why Pipes? | 63 |
| Uses of Pipes | 63 |
| Filters vs Pipes..... | 63 |
| Basic Pipes..... | 63 |
| New Pipes..... | 64 |
| The Async Pipe | 66 |
| Custom | 68 |
| What is ViewChild? | 70 |
| Chapter 7 : Working with Forms | 75 |

| | |
|---|-----|
| Angular 4.0 Forms – What it IS? | 75 |
| Template Driven Forms Features | 75 |
| Advantages and Disadvantages of Template Driven Forms | 75 |
| Advantages of Model Driven Forms | 76 |
| Form Control | 78 |
| Validating Reactive Forms | 78 |
| Reactive Forms Custom Validation | 78 |
| Transculation | 81 |
| What is Content Projection? | 82 |
| Chapter 8 : Service and Dependency Injection | 88 |
| What is Service? | 88 |
| Design Considerations..... | 88 |
| Create a Service..... | 89 |
| @Injectable | 89 |
| DepeWhat is Dependency Injection..... | 90 |
| Dependency Injection in Angular 4.0..... | 90 |
| Dependency Injection as a Design Pattern..... | 91 |
| Dependency Injection as a Framework..... | 92 |
| What are Providers?..... | 93 |
| Chapter 9 : Ajax Event Handling..... | 100 |
| Differences between Angular 1.x \$http and Angular 2 Http | 100 |
| Observables vs Promises | 100 |
| How to define Observables..... | 101 |
| Observable specificities..... | 102 |
| Observables vs Promises | 102 |
| Observables are cancellable. | 102 |
| Observable HTTP Events | 102 |
| Observables Array Operations | 103 |
| Http Post..... | 104 |
| Differences between \$http and angular/http | 104 |
| Chapter 10 : Routing | 112 |
| Why Routing ? | 112 |
| Route Definition Objects | 112 |

| | |
|---|-----|
| RouterModule | 113 |
| Redirecting the Router to Another Route | 113 |
| RouterLink | 114 |
| Dynamically Adding Route Components..... | 114 |
| What is Nested Child Routes? | 114 |
| Defining Child Routes | 115 |
| Passing Optional Parameters | 115 |
| Passing Query Parameters | 115 |
| Reading Query Parameters | 115 |
| Chapter 11 : Unit Testing..... | 122 |
| Unit Testing | 122 |
| Functional Testing | 122 |
| The Testing Tool chain | 123 |
| Jasmine..... | 123 |
| Karma | 124 |
| Why Unit Testing Required ? | 124 |
| Filename Conventions..... | 125 |
| ANGULAR TEST BED | 125 |
| When to use Angular Test Bed | 125 |
| OUR FIRST UNIT TEST | 125 |
| TESTING COMPONENTS | 135 |
| Testing Services | 140 |

CHAPTER 1: THE BASICS OF ANGULAR 4.0

Hypertext Markup Language (HTML) was invented in the year of 1990 by Tim Berners-Lee — a famous physics and computer scientist — while he was working at CERN, the European Organization for Nuclear Research. He was motivated about discovering a better solution to share the information among the researchers of the institution in a very quick and easy way. To support that, he also created the Hypertext Transfer Protocol (HTTP) and its first server, giving rise to the World Wide Web (WWW).

At that time, HTML was used just to create static documents with hyperlinks, allowing the navigation between them. However, in 1993, with the creation of Common Gateway Interface (CGI), it became possible to demonstrate or develop dynamic content generated by server-side applications. One of the first languages used for this purpose was Perl, followed by other languages such as Java, PHP, Ruby, and Python. However, the technology kept moving forward, at first with technologies such as Flash and Silverlight, which provided an amazing user experience through the usage of plugins. At the same time, the new versions of JavaScript, HTML, and CSS had been growing in popularity really fast, transforming the future of the Web by achieving a high level of user experience without using any proprietary plugin.

AngularJS is a part of this new generation of libraries and frameworks that came to support the development of more productive, flexible, maintainable, and testable web applications.

WHAT IS ANGULAR JS?

Basically Angular JS is an open source, JavaScript based web application development framework. Definition of AngularJS as put by its official documentation is as follows:

“AngularJS is a structural framework for dynamic web applications. It lets you use HTML as your template language and lets you extend HTML's syntax to express your application components clearly and succinctly. Its data binding and dependency injection eliminate much of the code you currently have to write. And it all happens within the browser, making it an ideal partner with any server technology.”

It was originally developed in 2009 by Misko Hevery and Adam Abrons. It is now maintained by Google. Angular JS supports JavaScript MVC (MVVM) frameworks.

The most common question before starting AngularJS is, “what is Angular JS?”. In common words, Angular is

- A MVC Structured Framework
- A Single Page Application Framework
- A Client-Side Templating
- A Language where written code can be easily tested by unit testing

Now, in the above key features of AngularJS contains main concept of innovations of AngularJS. The main idea of AngularJS is the separation between html manipulation and JavaScript logic in the web pages, so that html page and JavaScript logic can be developed simultaneously. It always gives us faster productivity as a developer's point of view. Also AngularJS provide us a structured JavaScript framework which can perform unit testing easily.

GENERAL FEATURES OF ANGULAR JS

The most important and remarkable general features of AngularJS are –

1. Angular JS is an efficient framework that can help developer to develop Rich Internet Applications (RIA).
2. Angular JS provides developer an options to write client side application using JavaScript using a clean Model View Controller (MVC) mechanism.
3. Code or Applications written in Angular JS are cross browser compatible. Angular JS framework automatically handles JavaScript code suitable for each browser.
4. AngularJS is open source, completely free, and used by thousands of developers around the world. It is licensed under the Apache license version 4.0.
5. AngularJS is a framework to build large scale, high performance, and easy-to-maintain web applications.

WHY ANGULAR JS KNOWN AS FRAMEWORK?

Before exploring AngularJS in depth, let us consider exactly what is AngularJS? What do we mean by a “framework,” and why would we want to use one? The dictionary definition tells us that a framework is “an essential supporting structure.” That sums up AngularJS very nicely, although AngularJS is much more than that. AngularJS is a large and helpful community, an ecosystem in which you can find new tools and utilities, an ingenious way of solving common problems, and, for many, a new and refreshing way of thinking about application structure and design. We could, if we wanted to make life harder for ourselves, write our own framework.

Realistically, however, for most of us, this just isn’t possible. It almost goes without saying that you need the support of some kind of framework, and that this framework almost certainly should be something other than your own undocumented (or less than well understood) ideas and thoughts on how things should be done. A good framework, such as AngularJS, is already well tested and well understood by others. Keep in mind that one day others may inherit your code, be on your team, or otherwise need to benefit from the structure and support a framework provides.

The use of frameworks isn’t uncommon; many programmers from all environment of coding rely on them. Business application developers use frameworks, such as the Microsoft Entity Framework, to ease their pain and speed up development when building database-related applications. For example, Java programmers use the LibGDX framework to help them create games. I hope I have explained you on the need for a framework and, more specifically, the fact that AngularJS is a great choice.

ARCHITECTURE CONCEPT

It's a long time since the famous Model-View-Controller (MVC) pattern provided by Microsoft Corporation started to gain popularity in the software development industry and became one of the legends of the enterprise architecture design. Basically, the model represents the knowledge that the view is responsible for presenting, while the controller mediates the relationship between model and view. After a lot of discussions about which architectural pattern need to follow by the Angular JS framework, its authors declared that from now on, AngularJS would adopt Model-View-Whatever (MVW). Regardless of the name, the most important benefit is that the framework provides a clear separation of the concerns between the application layers, providing modularity, flexibility, and testability.

In terms of concepts, a typical AngularJS application consists primarily of a view, model, and controller, but there are other important components, such as services, directives, and filters. The view, also called template, is entirely written in HTML, which provides a great opportunity to see web designers and JavaScript developers working side by side. It also takes advantage of the directives mechanism, which is a type of extension of the HTML vocabulary that brings the ability to perform programming language tasks such as iterating over an array or even evaluating an expression conditionally.

Behind the view, there is the component. At first, the component contains all the business logic implementation used by the view. However, as the application grows, it becomes really important to perform some refactoring activities, such as moving the code from the controller component to other components (for example, services) in order to keep the cohesion high.

The connection between the view and the controller is done by a shared object called model. It is located between them and is used to exchange information related to the model. The model is a simple **Plain-Old-JavaScript-Object (POJO)**. It looks very clear and easy to understand, bringing simplicity to the development by not requiring any special syntax to be created.

ADVANTAGES OF ANGULAR JS

AngularJS, commonly referred to as Angular, is an open-source web application framework maintained by Google and a community of individual developers and corporations to address many of the challenges encountered in developing single-page applications. The library works by first reading the HTML page, which has embedded into it additional custom tag attributes. Those attributes are interpreted as directives telling Angular to bind input or output parts of the page to a model that is represented by standard JavaScript variables. The values of those JavaScript variables can be manually set within the code, or retrieved from static or dynamic JSON resources.

The main advantages of the Angular JS are –

- No need to use observable functions; Angular analyses the page DOM and builds the bindings based on the Angular-specific element attributes. That requires less writing, the code is cleaner, easier to understand and less error prone.
- Angular modifies the page DOM directly instead of adding inner HTML code. That is faster.
- Data binding occurs not on each control or value change (no change listeners) but at particular points of the JavaScript code execution. That dramatically improves performance as a single bulk Model/View update replaces hundreds of cascading data change events.
- Quite a number of different ways to do the same things, thus accommodating to particular development styles and tasks.
- Extended features such as dependency injection, routing, animations, view orchestration, and more.
- Supported by IntelliJ IDEA and Visual Studio .NET IDEs.
- Supported by Google and a great development community.
- AngularJS is more intuitive as it makes use of HTML as a declarative language. Moreover, it is less brittle for reorganizing.
- AngularJS is a comprehensive solution for rapid front-end development. It does not need any other plugins or frameworks. Moreover, there are a range of other features that include Restful actions, data building, dependency injection, enterprise-level testing, etc.
- AngularJS is unit testing ready, and that is one of its most compelling advantages.

WHAT'S NEW IN ANGULAR 4.0 IN COMPARE ANGULAR JS?

Angular 4.0 was officially announced at the ng-conference in October, 2014. This version won't be a complex major update, rather a rewrite of the entire framework and will include breaking changes. The final version of Angular 4 was introduced by Google June 2016 and Later in March 2017, Google introduced another new version of Angular JS which is called Angular 4.0. Angular 4 or angular is a TypeScript-based open-source front-end web application platform led by the Angular Team at Google. Angular is a complete rewrite from the same team that built AngularJS. But let me tell you that Angular is completely different from AngularJS. Let us understand the differences between Angular and AngularJS:

- The architectural framework of an Angular application is different from AngularJS. The main building blocks for Angular are modules, components, templates, metadata, data binding, directives, services and dependency injection.
- Angular was a complete rewrite of 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 simpler expression syntax, focusing on "[]" for property binding, and "()" for event binding
- **Mobile development** – Desktop development is much easier when mobile performance issues are handled first. Thus, Angular first handles mobile development. The new Angular version will be focused on the development of mobile apps. The rationale is that it's easier to handle the desktop aspect of things, once the
- **Modularity** – Angular follows modularity. Similar functionalities are kept together in same modules. This gives Angular a lighter & faster core. Various modules from previous version of Angular Js has been removed from Angular's Core for better performance.
- Angular 4.0 will target ES6.0 and almost all modern browsers. Building for those browsers means that various hacks and workarounds that make angular harder to develop can be eliminated along developer to focus on code related to their business domain.

Now, let us talk about Angular 4. Angular community has introduced some significant changes to angular 4 and therefore, the major version number has been changed from 2 to 4 (skipping 3). The reason behind directly jumping to Angular 4 and skipping the version 3 was that the router package was in version 3.x, so instead of putting everything to 3.0 and the router to 4.0, the team chose to upgrade the versions of all the ng - modules to 4.0.

The following points introduced in Angular 4.0 :

- **TypeScript 2.1+ Required:** TypeScript 2.1 and 2.2 have brought really nice features you should check out. Angular 4 now supports them (and you will soon be able to activate the new strictNullChecks TypeScript option for example).
- **ModuleID Removed:** They have added a new SystemJS plugin which dynamically converts "component-relative" paths in templateUrl and styleUrls to "absolute paths" for you.
- **Ahead of Time Compilation - View Engine:** As we know, in AoT mode, Angular compiles your templates during the build, and generates JS code whereas in case of Just in Time mode, this compilation is done at runtime. Now, AoT has several advantages, like we will get to know that

our templates is incorrect at build time instead of having to wait at runtime, and the application starts faster (as the code generation is already done). The downside of AoT that people were facing was that the generated JS is generally bigger than the uncompiled HTML templates. So, in the vast majority of applications, the package is in fact bigger with AoT. The team worked quite hard to implement a new View Engine, that produces less code when you use the Ahead of Time compilation. The results are quite impressive on large apps, while still conserving the same performances.

- **Animations:** Angular Team have segregated animation package from @angular/core as a separate and dedication package. Therefore, if you don't use animations, this extra code will not end up in your production bundles.
- **Template is now ng-template:** The template tag is now deprecated, which mean you should use the ng-template tag instead. It still works though. Now Angular has its own template tag called ng-template. You will have a warning if you use the deprecated template somewhere when you update to Angular 4, so it will be easy to spot them.

WHAT IS TYPESCRIPT?

As per the previous discussion, it is clear that Angular 2 or 4 version is totally developed on the basis of Typescript whereas previous version of AngularJS is depends on JavaScript or JQuery library. Actually TypeScript is a super set scripting language of JavaScript. So before going to discuss about typescript, first we need to know what is TypeScript? As per the google, the definition of typescript is

"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."

It encourages software developers to more declarative style of programming like interfaces and static typing, offers modules and classes, and most importantly, integrates relatively well with popular JavaScript libraries and code. It totally follows the OOPS concept. Or we can say TypeScript is a transpiler. Many software developers will get very confused about Transpiler. It may be compiler but actually it is a transpiler. If you do not know about transpiler, then learn ahead. Actually transpiler means it basically converts one language into another language.

HOW TO INSTALL TYPESCRIPT?

So, before going to start Angular 4.0, we first need to know how to install typescript tool. For installing typescript, we first need to install NodeJS. The Latest version of NodeJS can be downloaded from the below URLs :-

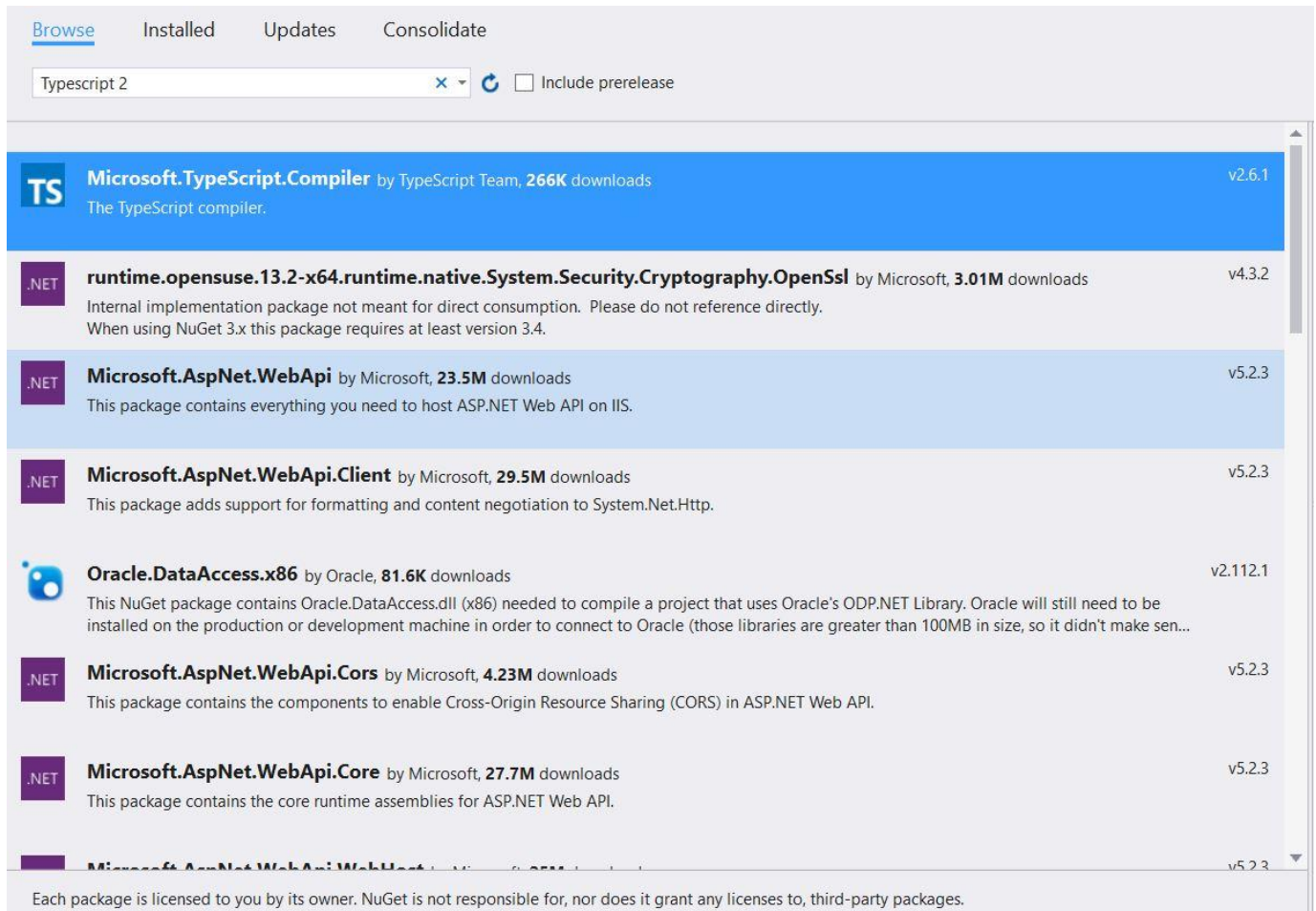
<https://nodejs.org/en/>

For install TypeScript we can download the latest version of typescript either by using command line argument using node.js or if we use Visual Studio, then we can directly download it from NuGet Package Manager.

Command Line for Install Typescript –

```
npm install -g typescript
```

Or from Visual Studio, We can install the Type Script using NuGet Package Manager Console -



Pic 1.1 – NuGet Package Manager Console in Visual Studio

CHAPTER 2 : FIRST PROGRAM IN ANGULAR 4

In this chapter, we will discuss about how to set the environment of an angular 4 projects using visual studio. Also we will develop our first program in Angular 4. Before going to start the program, lets' discuss about the project configuration.

Angular 4.0 project always contains 3 major configuration files. They are –

1. tsconfig.json
2. package.json
3. system.config.js

TSCONFIG.JSON

The existence of a tsconfig.json file in a project directory indicates that the directory is the root of a TypeScript project. The tsconfig.json file specifies the root files and the compiler options required to compile the project. A project is compiled in one of the following ways

- By invoking tsc with no input files, in which case the compiler searches for the tsconfig.json file starting in the current directory and continuing up the parent directory chain.
- By invoking tsc with no input files and a --project (or just -p) command line option that specifies the path of a directory containing a tsconfig.json file, or a path to a valid .json file containing the configurations.

When input files are specified on the command line, tsconfig.json files are ignored.

Sample of tsconfig.json

```
{
  "compileOnSave": true,
  "compilerOptions": {
    "target": "es5",
    "module": "commonjs",
    "moduleResolution": "node",
    "sourceMap": true,
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "removeComments": false,
    "noImplicitAny": false,
    "declaration": false,
    "typeRoots": [
      "node_modules/@types"
    ],
    "lib": [
      "es2017",
      "dom"
    ]
  }
}
```

The "compilerOptions" property can be omitted, in which case the compiler's defaults are used. For full list of supported Compiler Options, please visit the below links

<https://www.typescriptlang.org/docs/handbook/tsconfig-json.html>

By default all *visible* “@types” packages are included in your compilation. Packages in node_modules/@types of any enclosing folder are considered *visible*; specifically, that means packages within ./node_modules/@types/../../node_modules/@types/, ../../node_modules/@types/, and so on.

PACKAGE.JSON

This document is all you need to know about what's required in your package.json file. It must be actual JSON, not just a JavaScript object literal. A lot of the behavior described in this document is affected by the config settings described in npm-config.

name - The *most* important things in your package.json are the name and version fields. Those are actually required, and your package won't install without them. The name and version together form an identifier that is assumed to be completely unique. Changes to the package should come along with changes to the version. Some rules related to name attribute

- The name must be less than or equal to 214 characters. This includes the scope for scoped packages.
- The name can't start with a dot or an underscore.
- New packages must not have uppercase letters in the name.
- The name ends up being part of a URL, an argument on the command line, and a folder name. Therefore, the name can't contain any non-URL-safe characters.

Code of package.json

```
{
  "name": "angular4-article",
  "version": "1.0.0",
  "description": "Angular 4 article projects",
  "scripts": {
    "build": "tsc -p src/",
    "build:watch": "tsc -p src/ -w",
    "build:e2e": "tsc -p e2e/",
    "serve": "lite-server -c=bs-config.json",
    "serve:e2e": "lite-server -c=bs-config.e2e.json",
    "prestart": "npm run build",
    "start": "concurrently \"npm run build:watch\" \"npm run serve\"",
    "pree2e": "npm run build:e2e",
    "e2e": "concurrently \"npm run serve:e2e\" \"npm run protractor\" --kill-
others --success first",
    "preprotractor": "webdriver-manager update",
    "protractor": "protractor protractor.config.js",
    "pretest": "npm run build",
    "test": "concurrently \"npm run build:watch\" \"karma start
karma.conf.js\"",
    "pretest:once": "npm run build",
```

```

    "test:once": "karma start karma.conf.js --single-run",
    "lint": "tslint ./src/**/*.ts -t verbose"
  },
  "keywords": [],
  "author": "Debasis",
  "license": "MIT",
  "dependencies": {
    "@angular/common": "~4.3.4",
    "@angular/compiler": "~4.3.4",
    "@angular/core": "~4.3.4",
    "@angular/forms": "~4.3.4",
    "@angular/http": "~4.3.4",
    "@angular/platform-browser": "~4.3.4",
    "@angular/platform-browser-dynamic": "~4.3.4",
    "@angular/router": "~4.3.4",
    "@angular/animations": "~4.3.4",
    "angular-in-memory-web-api": "~0.3.0",
    "systemjs": "0.19.40",
    "core-js": "^2.4.1",
    "rxjs": "^5.5.6",
    "zone.js": "^0.8.4"
  },
  "devDependencies": {
    "concurrently": "^3.2.0",
    "lite-server": "^2.2.2",
    "typescript": "~2.4.0",
    "canonical-path": "0.0.2",
    "tslint": "^3.15.1",
    "lodash": "^4.16.4",
    "jasmine-core": "~2.4.1",
    "karma": "^1.3.0",
    "karma-chrome-launcher": "^2.0.0",
    "karma-cli": "^1.0.1",
    "karma-jasmine": "^1.0.2",
    "karma-jasmine-html-reporter": "^0.2.2",
    "protractor": "~4.0.14",
    "rimraf": "^2.5.4",
    "@types/node": "^6.0.46",
    "@types/jasmine": "2.5.36"
  },
  "repository": {}
}

```

version - The *most* important things in your package.json are the name and version fields. Those are actually required, and your package won't install without them. The name and version together form an identifier that is assumed to be completely unique. Changes to the package should come along with changes to the version.

description - Put a description in it. It's a string. This helps people discover your package, as it's listed in npm search

keywords - Put keywords in it. It's an array of strings. This helps people discover your package as it's listed in npm search

homepage - The url to the project homepage

license - You should specify a license for your package so that people know how they are permitted to use it, and any restrictions you're placing on it.

People fields : authors, contributors - The "author" is one person. "contributors" is an array of people. A "person" is an object with a "name" field and optionally "url" and "email".

files – The optional "files" field is an array of file patterns that describes the entries to be included when your package is installed as a dependency. If the files array is omitted, everything except automatically-excluded files will be included in your publish. If you name a folder in the array, then it will also include the files inside that folder (unless they would be ignored by another rule in this section.).

directories – The CommonJS Packages spec details a few ways that you can indicate the structure of your package using a directories object. If you look at npm's package.json, you'll see that it has directories for doc, lib, and man.

SYSTEM.CONFIG.JS

system.config.js is the one which allows to load modules(node modules) compiled using the TypeScript compiler.map refers to the name of modules to JS file that contains the JavaScript code. It allows to configure SystemJS to load modules compiled using the TypeScript compiler. For anonymous modules (one module per JS file), it allows to map the name of modules to JS files that actually contains the module JavaScript code

Code of system.config.js

```
/**
 * System configuration for Angular samples
 * Adjust as necessary for your application needs.
 */
(function (global) {
    System.config({
        paths: {
            // paths serve as alias
```



```

        'npm:': 'node_modules/'
    },
    // map tells the System loader where to look for things
    map: {
        // our app is within the app folder
        'app': '.',
        // angular bundles
        '@angular/animations':
npm:@angular/animations/bundles/animations.umd.js',
        '@angular/animations/browser':
npm:@angular/animations/bundles/animations-browser.umd.js',
        '@angular/core': 'npm:@angular/core/bundles/core.umd.js',
        '@angular/common': 'npm:@angular/common/bundles/common.umd.js',
        '@angular/compiler':
'npm:@angular/compiler/bundles/compiler.umd.js',
        '@angular/platform-browser': 'npm:@angular/platform-
browser/bundles/platform-browser.umd.js',
        '@angular/platform-browser/animations': 'npm:@angular/platform-
browser/bundles/platform-browser-animations.umd.js',
        '@angular/platform-browser-dynamic': 'npm:@angular/platform-
browser-dynamic/bundles/platform-browser-dynamic.umd.js',
        '@angular/http': 'npm:@angular/http/bundles/http.umd.js',
        '@angular/router': 'npm:@angular/router/bundles/router.umd.js',
        '@angular/router/upgrade': 'npm:@angular/router/bundles/router-
upgrade.umd.js',
        '@angular/forms': 'npm:@angular/forms/bundles/forms.umd.js',
        '@angular/upgrade': 'npm:@angular/upgrade/bundles/upgrade.umd.js',
        '@angular/upgrade/static': 'npm:@angular/upgrade/bundles/upgrade-
static.umd.js',
        // other libraries
        'rxjs': 'npm:rxjs',
        'angular-in-memory-web-api': 'npm:angular-in-memory-web-
api/bundles/in-memory-web-api.umd.js'
    },
    // packages tells the System loader how to load when no filename and/or
    no extension
    packages: {
        app: {
            main: './main.js',
            defaultExtension: 'js'
        },
        rxjs: {
            defaultExtension: 'js'
        }
    }
}

```

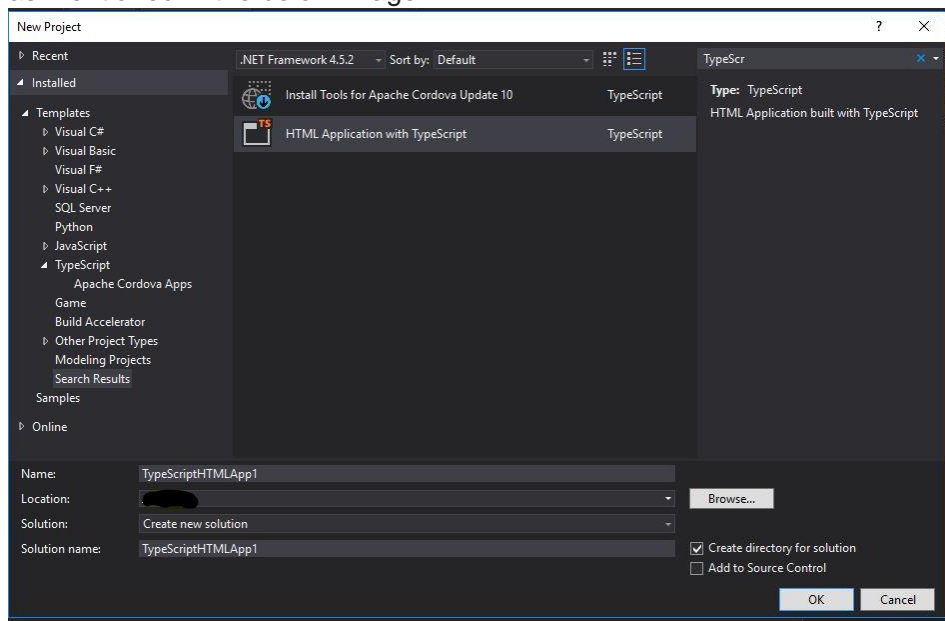


```
});
})(this);
```

FIRST PROGRAM IN ANGULAR 4.0

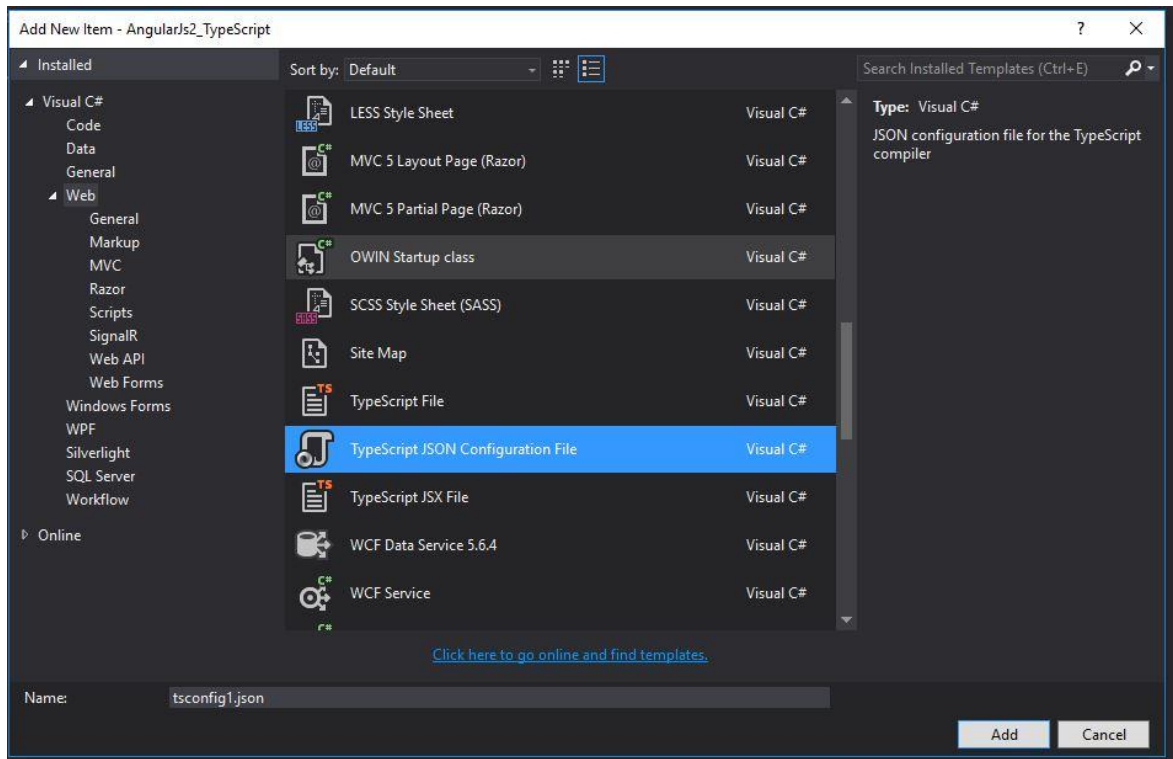
Now, we are going to develop our first program in AngularJS2. Now open the Visual Studio 2015 and do the following steps.

1. Click on File -> New -> Projects and in the New Project window, Search TypeScript Project as mentioned in the below image.



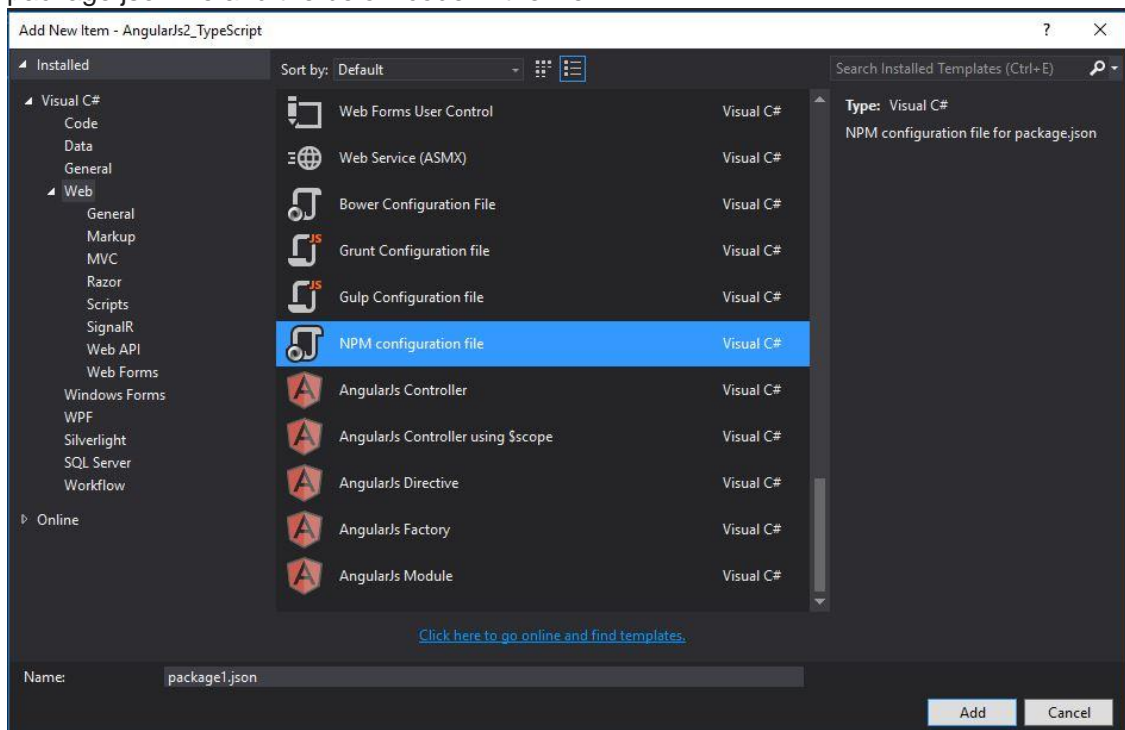
Pic 3.1 – Blank Solution in Visual Studio 2015

2. A blank project file has been created. Click on Project -> Add File and Select file Type TypeScript JSON Configuration File for create blank tsconfig.js file and then add the below code within the file,



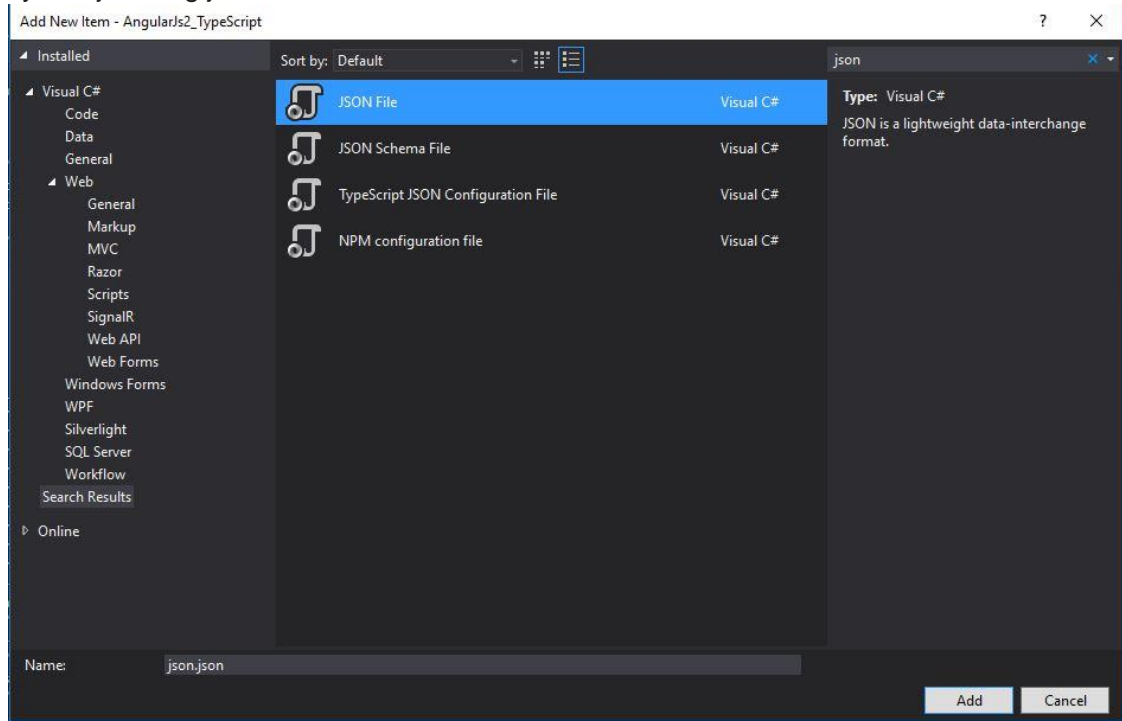
Pic 3.2 – Add tsconfig.json file in Project Folder

- Now Click on Project --> Add New Item and Select file Type NPM Configuration File for package.json file and the below code in the file.



Pic 3.3 – Add package.json file in Project Folder

- Click on Project -> Add New Item and Select file Type JavaScript File with file name systemjs.config.js and the below code.



Pic 3.4 - Add system.config.js file in Project Folder

- Now, open the package.json file location in the command prompt and execute the below command to load the required modules and supported files which are mentioned in the package.json file.
npm install
- This command installs or downloads the required packages from Angular 4.0 domain as mentioned in the package.json files.
- Now, add a TypeScript file named main.ts and the below code.

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

import { DemoModule } from './app.demo.module';

platformBrowserDynamic().bootstrapModule(DemoModule);
```

- Here, the Import keyword is used to include the browser module into the program. Now, add another TypeScript file named app.demo.module.ts

```
import { NgModule, Provider, ModuleWithProviders, NO_ERRORS_SCHEMA } from '@angular/core';
```

```
import { APP_BASE_HREF } from '@angular/common';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule } from "@angular/forms";
import { HttpClientModule } from '@angular/http';
import { RouterModule } from '@angular/router';
import { ReactiveFormsModule } from "@angular/forms";

import { HomeComponent } from './home/app.component.home';

@NgModule({
  imports: [BrowserModule, FormsModule, HttpClientModule, ReactiveFormsModule],
  declarations: [HomePageComponent],
  bootstrap: [HomePageComponent],
  schemas: [NO_ERRORS_SCHEMA]
})

export class DemoModule { }
```

9. Here, @NgModule is the module annotations. We will discuss about it in the next chapters. Now, add another TS file named app.component.home.ts and add the below code.

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

@Component({
  moduleId :module.id,
  selector: "home",
  template: "Welcome To Angular 4 Series "
})

export class HomeComponent {
  constructor() {
  }
}
```

10. Here, @Component is the component annotation which defines a component which contains a selector property. Basically selector property tells the browser which component needs to load. Now, add an HTML file named *index.html* and write down the below code.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <!--<base href="/">-->
```

```

<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<meta charset="utf-8">
<title>Angular 4 - Console</title>
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<meta name="description" content="">
<meta name="keywords" content="">
<meta name="author" content="">

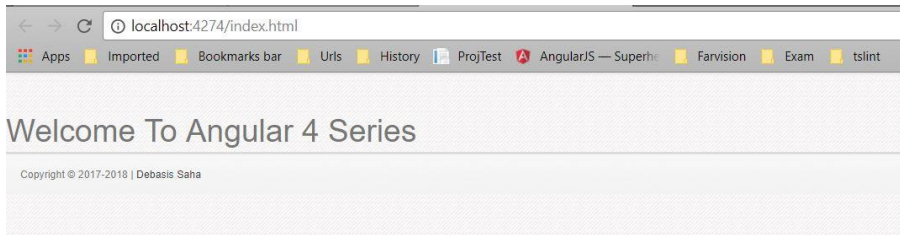
<link href="resource/css/bootstrap.min.css" rel="stylesheet">
<link rel="stylesheet" href="resource/css/font-awesome.min.css">
<link rel="stylesheet" href="resource/css/jquery-ui.css">
<link href="resource/css/style.css" rel="stylesheet">
<link rel="shortcut icon" href="img/favicon/favicon.ico">
<link href="app.css" rel="stylesheet" />
</head>
<body>
    <div class="content">
        <home-page></home-page>
    </div>
    <footer>
        <div class="container">
            <div class="row">
                <div class="col-md-12">
                    <p class="copy">Copyright &copy; 2017-2018 | <a
href="http://www.c-sharpcorner.com/members/debasis-saha">Debasis Saha</a>
</p>
                </div>
            </div>
        </div>
    </footer>
    <script src="resource/js/jquery.js"></script>
    <script src="resource/js/bootstrap.min.js"></script>
    <script src="resource/js/jquery-ui.min.js"></script>
    <script src="resource/js/jquery.slimscroll.min.js"></script>
    <script src="resource/js/custom.js"></script>

    <script src="node_modules/core-js/client/shim.min.js"></script>
    <script src="node_modules/zone.js/dist/zone.js"></script>
    <script src="node_modules/systemjs/dist/system.src.js"></script>
    <script src="systemjs.config.js"></script>
    <script>
        System.import('main.js').catch(function (err) { console.error(err);
    });
    </script>

```

```
<!-- Set the base href, demo only! In your app: <base href="/"> -->
<script>document.write('<base href="' + document.location + "')';</script>
</body>
</html>
```

11. Now, build and run the program and see the output.



Pic 3.5 – First Program run into the browser.

CHAPTER 3 : COMPONENT – AN OVERVIEW

Angular 4 is basically a component based framework. Components combine concepts that we are already familiar with from AngularJS. The Angular 4 Component combines the AngularJS Directive, Controller, and Scope. Angular 4 is actually component based MVC Framework. Components are the main building blocks of Angular 4 applications. But before going to discuss about component, first we need to understand what is component and why component based framework introduced by google in angular.

WHAT IS A COMPONENT?

A component is basically a class that is defined to visible any element on the screen. The components have some properties and by using them we can manipulate how the element should look and behave on the screen. We can create, destroy and update our own component as per our requirement. But when we writing code using TypeScript, a component is basically a TypeScript class decorated with @Component() decorator.

```
import {Component} from "@angular/core";
@Component({
  selector: 'first-prog',
  template: 'Welcome to Angular 4.0 Series'
})
export class FirstProgComponent{
}
```

Decorators are basically JavaScript functions which amend the decorated class in some way. Basically, it this functions additional meaning to a plain JavaScript class based on the decorator used. From the developers point of view, we just declaratively decorate a class by passing a mandatory configuration object as a parameter to the decorator function. The key idea behind developing an Angular 4 application is coming up with components and composing them together as required to build an application. A component is an independent complete block of code which has the required logic, view, and data as one unit. Logically every component in Angular 4 acts a MVC concept itself.

Components are a really neat idea. Each component class being an independent unit, it is highly reusable and can be maintained without messing up with other components. Scaling is as easy as adding more components to your existing application.

ADVANTAGES OF COMPONENT BASED FRAMEWORK

As per the current IT leaders, component based UI development is the future of the web development. Basically it's a technique which digital application need to implement right now. Development work with the help of Component based technique, it creates a sustainable technical architecture with saving time and costing. Below is the main advantages of the component based architecture –

1. **Reusability** – In Component based framework, component is the most granular units in our development work and development with components allows for their reuse in future development cycles. Since technology is invaluable because it's changes or updated every day. So if we develop an application in a component based format, then we are able to swap the best components in and out. One of the challenges of reuse with other development types is that they

are not internally built or that they include many dependencies. A component-based UI approach allows your application architecture to stay up to date over time instead of rebuilding it from scratch. You can build multiple applications that adhere to the intended design principles.

2. **A Component based UI Approach Accelerate Development Speed** – Component based development always support agile development. Components can be stored in a library from which team can access, integrate and modify them throughout the development process. Suppose that we were developing a financial application, and it required a listing of positions by bank transaction class. So we can pull a date-wise-bank-transaction-class grid component from the component library and quickly integrate it into the application. The developer does not have to worry about the service signature and creating the logic for the table. In the design process, instead of designing new components, the designer focuses time on extending the existing components and designing new components where required. This optimizes the design process without designing a new grid, layout, or navigation. Ultimately, this expedites the design and development process because of the level of reuse.
3. **It Provide Consistent User Experience throughout the Application** – One of the main challenge for an organization is ensuring that the entire application must provide consistent user experience and interactions. The component library acts as a point of governance for the business, designers, and quality assurance teams. In the case of Quality Assurance (QA), teams often have challenges validating the user interface due to a lack of an approved set of user interface standards. The component-based approach enables the creation of a library that provides that approved reference point. This enables the QA team to govern the compliance to UI standards across applications. It acts as a dynamic repository that the QA team can use to validate their tests. As an example, there are many ways to handle a file upload feature within an application. A new application may select an approach that is different from the approved version within the component library; QA can utilize the component library to confirm alignment and then open a UI related defect.
4. **Easy Integration in Development Process** – As components are created or development by the development team, so production quality user interface code can be managed within a centralized code repository. Application development teams are well versed in using source code repositories, and so they are able to extract the code as needed and incorporate it into the application. Leveraging the initial component as a starting point, development teams can extend it to meet their needs. Then they can submit it into the code repository for review and approval for inclusion.
5. **Component Based UI development optimizes the requirements and design process** – Using component based library as a reference, product managers, business analysts and user experience designers or UI developer can spend less time for defining and details application functionality and user operations. As they work through the definition process and requirements elaboration, they can reference a component as the baseline for the requirement, and then only spend time defining the required extensions and business logic. This minimizes the team's focus on how specific user interactions should work. Some examples are filtering, pagination in grids,

and display of complex data (positions, trades, exceptions). Below is the steps of the development process using Component Based Framework –

- a. **Requirement Definition** – Components are identified during requirement or design process and gaps understood.
 - b. **UI Design Optimization** – UI effort focused on extending existing components and defining new as needed.
 - c. **Source Code Reuse** – Developers first look at the component library before developing a component.
 - d. **Governance and Quality** - Used for governance to ensure alignment and identify any deviation easily.
 - e. **Integrated with QA** – Testing team leverage UI library to validate standards compliance and accelerate the validation process
 - f. **Integrated with Development** - Generated from application source code and part of the continuous integration / build process.
6. **It speeds up from design to development** – As the rate of change within businesses continues to accelerate, teams needs to find ways to accelerate the “Time to Value” for application development projects. Shifting to a design approach that demonstrates the user experience in the browser will have a significant impact on delivery timelines. There is often the issue of “lost in translation” that occurs when a wireframe or visual design is handed over to a development team for implementation. During the process of turning that into the application user interface, issues are surfaced and some portions are not translated correctly. This results in a misalignment of the delivered experience to the defined user experience.

COMPONENT CONFIGURATION

@Component decorator basically decorated a type script class as a component objects. Basically it is a function which takes different types of parameters. In the @component decorator, we can set the values of different properties to finalize the behavior of the components. The most used properties are as bellows :-

1. **selector** :- A component can be used by the selector expression.
2. **template** :- Basically template is the part of the component which rendered in the browser. We can directly define the html tags or code within the template properties. Sometimes we called this as inline template. For writing multiple line of html code, all code need to be covered within tilt (') symbol.
3. **templateUrl** :- Another way of rendered html tags in the browser. Here we need to provide the html file name with its related file path. Some times it is known as external template. It is better approach if the html part of the component is complex.
4. **moduleId** :- It is used to resolve the related path of template url or style url for the component objects.
5. **styles or styleUrls** :- Component can be used its own style by providing custom css or can refer to an external style sheet files which can be used by multiple components at a time. For proving inline style, we need to use styles and for provide external file path or url, we need to use styleUrls.

6. **providers** - We in the real life application, we need to use or inject different types of custom service within the component for implement the business logic for the component. For use any custom service within the component, we need to provide the service instance within the provider. Basically provider is an array type property where multiple service instance name can be provided by comma separation.

LIFE CYCLES OF THE COMPONENT

The life cycle of the component maintains by the Angular 4 himself. Below is the list of life cycle method of angular 2 components.

- **constructor** – method executed before execution of any one life cycle method. It is basically used for inject dependency.
- **ngOnChanges** – method called when an input control or binding value changes
- **ngOnInit** - method called after the execution of first ngOnChanges
- **ngDoCheck** – method called after every execution of ngOnChanges
- **ngAfterContentInit** – method execute after component content initialized
- **ngAfterContentChecked** – method execute after every check of component check
- **ngAfterViewInit** – method execute after component views initialize
- **ngAfterViewChecked** – method execute after every check of component views
- **ngOnDestroy** – method execute after before the component destroy.

ANGULAR COMPONENT ARCHITECTURE LAYOUT

If we remember the Angular architecture, every UI in angular framework must be composed as a tree of Angular Component – one component will be placed besides another component, both of which may be wrapped up by one outer component and so on. This hierarchy is start with one Root Component which is normally called as Bootstrapped Component. Here are the steps of the Create Root Component –

- Create Angular Module
- Create Angular Component
- Add Component to Module
- Bootstrap Module
- Bootstrap Component

CREATE ANGULAR MODULE

As we already discussed earlier that everything in angular 4 is belongs to an Angular Module. So for develop the root component, we first need to declared our first angular module. Angular module can be defined by creating a TypeScript class decorated with the “NgModule” decorator. Actually NgModule is a decorator defined within the “@angular/core” library. @NgModule takes a metadata object that tells Angular how to compile and run module code. It identifies the module's own components, directives, and pipes, making some of them public so external components can use them. In order to use it, we first need to import it as follows –

```
import { NgModule } from '@angular/core';  
@NgModule()  
export class DemoModule { }
```

CREATE ANGULAR COMPONENT

Finally, we reach a position where we need to create our first component using Angular 4. It can be done by creating a class decorated with @Component decorator which defined within the “@angular/core” library. Below the Code for the angular component –

```
import { Component } from "@angular/core";  
  
@Component({  
  selector: "home",  
  template: "<h1>Welcome To Angular 4 Series</h1> "  
})  
  
export class HomeComponent {  
  constructor() {  
  }  
}
```

In the above code, we pass two parameters to the component decorator –

1. selector – a simple text representing the tag name of the component.
2. template – UI part of the component

ADD COMPONENT TO MODULE

Now, the next step is to add the component within the angular module. It can be done using “declarations” option within “NgModule” decorator. For adding the component, we need to import the component within the module by using import keyword.

```
import { NgModule } from '@angular/core';  
import { HomeComponent } from './SampleCode/app.component.home';  
  
@NgModule({  
  declarations: [HomeComponent]  
})  
  
export class DemoModule { }
```

BOOTSTRAP THE MODULE

As we already discussed that a single angular application can contains more than one angular modules. But out of the all module, only one module can be bootstrapped at the beginning. In Angular 4 this bootstrapping process need to be done manually with the help of “platformBrowserDynamic” function which is defined within the “@angular/platform-browser-dynamic” library. Before bootstrap angular module, it is import to defined export keyword in the module class definition statement so that we can

import that class into to another file. Normally, as per standard guideline of the Angular, we define this bootstrap process within the **main.ts** file.

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

platformBrowserDynamic().bootstrapModule(SampleModule);
```

BOOTSTRAP THE COMPONENT

In the previous section, we discuss that if there are many modules that we need to bootstrap one module as a starting module which need to defined within the main.ts file. But at the same time, one module always contains multiple component. Then what happened means in this scenario which component will be starting component or how angular will identify its root component? For this reason, we also need to bootstrap the root component so that angular can identify it. It can be done by using the bootstrap option in “NgModule” decorator.

```
import { NgModule } from '@angular/core';
import { HomeComponent } from './SampleCode/app.component.home';

@NgModule({
  declarations: [HomeComponent],
  bootstrap: [HomeComponent],
})

export class SampleModule { }
```

SAMPLE PROGRAM

Now, we will create a basic component which will display some data into the web pages. For that, we first create a typescript file named app.component.helloworld.ts and add the below code,

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

@Component({
  selector: 'Hello-world',
  template: '<h1><b>Angular 4!</b>Hello World</h1>'
})

export class HelloWorldComponent {
  constructor() {
  }

  ngOnInit() {
    alert("Page Init Method Fired!!")
  }
}
```

```
    }
}
```

Now, add another TypeScript file named `app.component.home.ts` and the below code (basically this is the root component for us. We actually bootstrapped this component from the `app.module.ts` file

```
import { Component } from '@angular/core';
@Component({
  moduleId: module.id,
  selector: 'home-page',
  templateUrl: 'app.component.home.html'
})
export class HomeComponent {
  constructor() {
  }
}
```

As per the above code, we use `templateUrl` properties of the component decorator for the root component. For this, we need to add an HTML file called `app.component.home.html` (We placed TypeScript and HTML file in the same folder location. But we can also place these two types of files in different locations. In that scenario, we need to provide the HTML file name with related file path and the below code.

```
<strong>Angular 4 Code Samples</strong>
<br />
<Hello-world></Hello-world>
<br />
```

Now, add another TypeScript file for Angular module and write down the code given below.

```
import { NgModule, Provider, ModuleWithProviders, NO_ERRORS_SCHEMA } from
'@angular/core';
import { APP_BASE_HREF } from '@angular/common';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule } from "@angular/forms";
import { HttpModule } from '@angular/http';
import { RouterModule } from '@angular/router';
import { ReactiveFormsModule } from "@angular/forms";

import { HomeComponent } from './home/app.component.home';
import { HelloWorldComponent } from './SampleCode/app.component.helloworld';
```

```
@NgModule({
  imports: [BrowserModule, FormsModule, HttpModule, ReactiveFormsModule],
  declarations: [HomePageComponent, HelloWorldComponent ],
  bootstrap: [HomePageComponent],
  schemas: [NO_ERRORS_SCHEMA]
```

```
})
```

```
export class DemoModule { }
```

1. Now, add an HTML file named *index.html* and write down the below code.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <!--<base href="/">-->
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  <meta charset="utf-8">
  <title>Angular 4 - Console</title>
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta name="description" content="">
  <meta name="keywords" content="">
  <meta name="author" content="">

  <link href="resource/css/bootstrap.min.css" rel="stylesheet">
  <link rel="stylesheet" href="resource/css/font-awesome.min.css">
  <link rel="stylesheet" href="resource/css/jquery-ui.css">
  <link href="resource/css/style.css" rel="stylesheet">
  <link rel="shortcut icon" href="img/favicon/favicon.ico">
  <link href="app.css" rel="stylesheet" />
</head>
<body>
  <div class="content">
    <home-page></home-page>
  </div>
  <footer>
    <div class="container">
      <div class="row">
        <div class="col-md-12">
          <p class="copy">Copyright &copy; 2017-2018 | <a
href="http://www.c-sharpcorner.com/members/debasis-saha">Debasis Saha</a>
</p>
        </div>
      </div>
    </div>
  </footer>
  <script src="resource/js/jquery.js"></script>
```

```

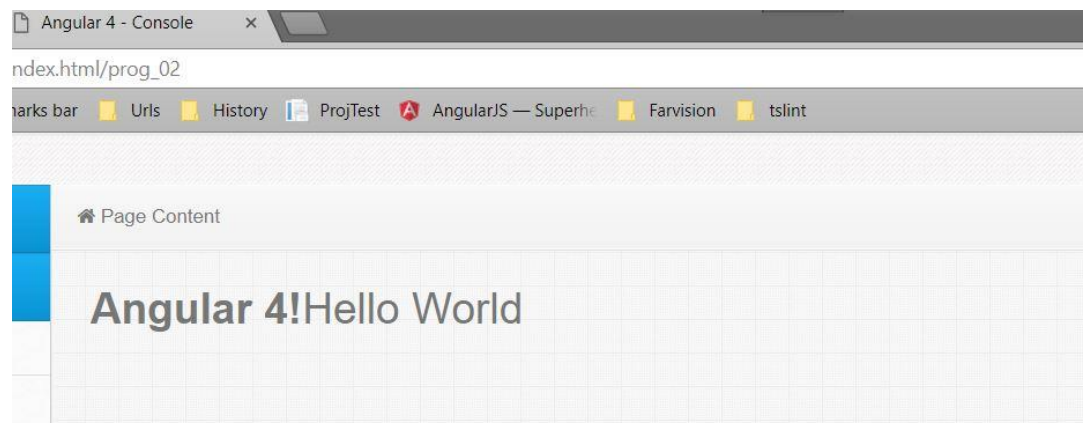
<script src="resource/js/bootstrap.min.js"></script>
<script src="resource/js/jquery-ui.min.js"></script>
<script src="resource/js/jquery.slimscroll.min.js"></script>
<script src="resource/js/custom.js"></script>

<script src="node_modules/core-js/client/shim.min.js"></script>
<script src="node_modules/zone.js/dist/zone.js"></script>
<script src="node_modules/systemjs/dist/system.src.js"></script>
<script src="systemjs.config.js"></script>
<script>
    System.import('main.js').catch(function (err) { console.error(err);
});
</script>

<!-- Set the base href, demo only! In your app: <base href="/" -->
<script>document.write('<base href="' + document.location + '"/>');</script>
</body>
</html>

```

Now, run the code. The output is shown below.



Pic 4.1 – Hello World Program result

Now, we want to create another component which will use templateUrl properties in spite of template properties. For that, create another TypeScript file named app.component.template.ts and add the below code.

```

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

@Component({
  moduleId: module.id,
  selector: 'template-url',

```

```

        templateUrl: 'app.component.template.html'
    })

    export class templateUrlComponent {
        constructor() {

        }
    }

```

Add another HTML file named app.component.template.html and add the below code.

```

<div>
    <h1>Angular 4 Component with Template & Style Decorator</h1>
    <br />
    <h2>C# Corner</h2>
</div>

```

Now, import the component in the app.module.ts file and add the component name within the declaration properties.

Add the HTML attribute in the app.component.home.html file as below

```

<strong>Angular 4 Code Samples</strong>

<br />

<Hello-world></Hello-world>

<br />

<template-url></template-url>

<br />

```

Now, run the code. The output s shown below.

Angular 4 Component with Template & Style Decorator

C# Corner

Pic 4.2 – Angular Component with Template Url

Now, in this case, we can apply the custom style using CSS for the HTML tags. For CSS style, we need to use styles properties, as below.

```

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

```



```
@Component({
  moduleId: module.id,
  selector: 'template-style',
  templateUrl: 'app.component.template.html',
  styles: ['h1{color:red;font-weight:bold}','h2{color:blue}']
})

export class TemplateUrlStyleComponent {
  constructor() {

  }
}
```

Now, run the code or refresh the browser.

Angular 4 Component with Template & Style Decorator

C# Corner

Pic 4.3 – Angular Component with Template Url & Custom Css Styles

CHAPTER 4 : DATA BINDING

Data Binding, one of the most important and fantastic feature of Angular 4.0. Now in this chapter, we will discuss about data binding concept in Angular 4.0

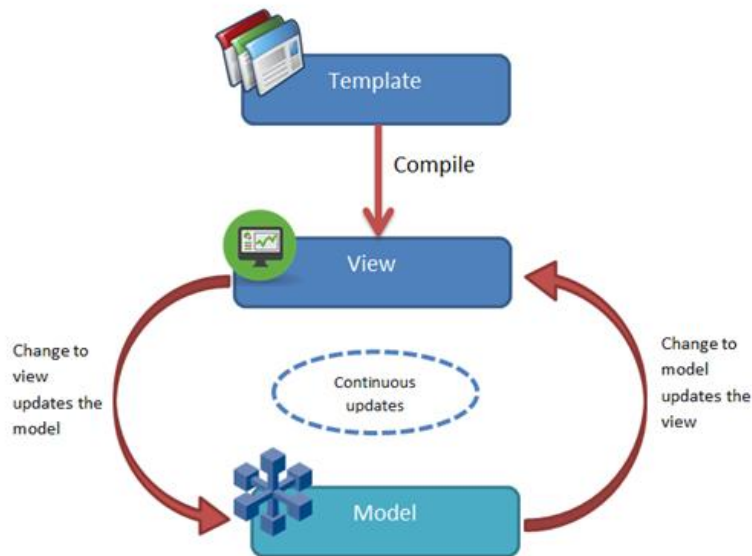
WHAT IS DATA BINDING ?

Data-binding in Angular apps is the automatic synchronization of data between the model and view components. The way that Angular implements data-binding lets you treat the model as the single-source-of-truth in your application. The view is a projection of the model at all times. Data binding is the process that establishes a connection between the application UI and business logic. If the binding has the correct settings and the data provides the proper notifications, then, when the data changes its value, the elements that are bound to the data reflect changes automatically.

WHY DATA BINDING IS REQUIRED?

In traditional web development, we need to develop a bridge between front end, where the user performs their application data manipulation and the back end, where the data is stored. Now, in this development, this process is driven by consecutive networking calls, communicating changes between the server (i.e. back end) and the client (i.e. front end). For the solution of this problem, angular involved this data binding concept.

Most web framework platforms focus on the one-way data binding. Basically, this involves reading the input from DOM, serializing the data, sending it to the back-end or server, waiting for the process to finish. After it, process modify the DOM to indicate any error occurred or reload the DOM element if the call is successful. While this process provides a traditional web application all the time it needs to perform data processing, this benefit is only really applicable to web apps with highly complex data structures. If your application has a simpler data structure format, with relatively flat models, then the extra work can needlessly complicate the process and decrease the performance of your application. Furthermore, all models need to wait for server confirmation before their data can be updated, meaning that related data depending upon those models won't have the latest information.



Pic 5.1 – Angular Data Model Structure

Angular framework addresses this with data binding concept. With data binding, the user interface changes are immediately reflected in the underlying data model, and vice-versa. This allows the data model to serve as an atomic unit that the view of the application can always depend upon to be accurate. Many web frameworks implement this type of data binding with a complex series of event listeners and event handlers – an approach that can quickly become fragile. Angular, on the other hand, makes this approach to data a primary part of its architecture. Instead of creating a series of callbacks to handle the changing data, Angular does this automatically without any needed intervention by the programmer. Basically, this feature is a great relief for the programmer.

So the primary benefit of data binding is that updates to (and retrievals from) the underlying data store happen more or less automatically. When the data store updates, the UI updates as well. This allows us to remove a lot of logic from the front-end display code, particularly when making effective use of Angular's declarative approach to UI presentation. In essence, it allows for true data encapsulation on the front-end, reducing the need to do complex and destructive manipulation of the DOM.

While this solves a lot of problems with a website's presentation architecture, there are some disadvantages to take into consideration. First, Angular uses a dirty-checking approach that can be slow in some browsers – not a problem for small presentation pages, but any page with heavy processing may run into problems in older browsers. Additionally, data binding is only truly beneficial for relatively simple objects. Any data that requires heavy parsing work, or extensive manipulation and processing, will simply not work well with two-way binding. Additionally, some uses of Angular – such as using the same binding directive more than once – can break the data binding process.

WHAT IS DIRTY CHECKING?

In angular framework, model and view are inter connected with the binding mechanism. So that any changes made in the view will update the model and vice versa. This update of model or view occurred due to the model change event which is called Digest Cycle and dirty check is one of the part of Digest Cycle. Actually, when any event or model value manipulation is done, angular itself check the old value and new value and if value does not match, then the digest cycle starts its work and update the view by checking the model value and find which object need to be changed. Actually digest cycle inform the watchers about the model change and then watchers synchronize the view with the model data. Now while this updation process is going on, it is possible that the value of model again changed . So now dirty check comes into picture , and check while the digest cycle(was going on) any thing is changed in model or not . If any thing changed it will call the digest cycle again and update the view accordingly, and this process will go on until dirty check find no updates done while last Digest cycle.

Angular always creates a change detector object per component, which tracks the last value of each template binding, such as `{{employee.name}}`. In normal scenerio, after every asynchronous browser event execution (such as a response from a server, or a click event, or a timeout event), Angular change detection executes and dirty checks every binding using those change detector objects.

If a change is detected, the change is propagated. E.g.,

- If an input property value changed, the new value is propagated to the component's input property.
- If a `{{}}` binding value changed, the new value is propagated to DOM property `textContent`.
- If the value of `x` changes in a style, attribute, or class binding – i.e., `[style.x]` or `[attr.x]` or `[class.x]` – the new value is propagated to the DOM to update the style, HTML attribute, or class.

Angular uses Zone.js to create its own zone (NgZone), which monkey-patches all asynchronous events (browser DOM events, timeouts, AJAX/XHR). This is how change detection is able to automatically run after each asynchronous event. I.e., after each asynchronous event handler (function) finishes, Angular change detection will execute.

DIFFERENT TYPES OF DATA BINDING

In Angular 4, there are different types of data binding mechanism is available. There are –

1. Interpolation
2. Property Binding
3. Two Way Binding
4. Event Binding

INTERPOLATION

Interpolation data binding is the most popular and easiest way of data binding in angular 4. This mechanism is also available in the earlier version of the angular framework. Actually context between the braces is the template expression that Angular first evaluates and then convert into strings. Interpolation use the braces expression i.e. `{{ }}` for render the bound value to the component template. It can be a static string or numeric value or an object of your data model. As per example, in angular we use it like below

```
{{model.firstName}}
```

Here model is the instance of the controller objects. But in angular 4, it is now much simpler where controller instance name removed from the expression i.e.

```
{{firstName}}
```

PROPERTY BINDING

In AngularJS 4.0, a new binding mechanism introduces which called property binding. But actually in nature is just same as interpolation. Some people also called this process as one way binding as like previous AngularJS concept. Property binding used [] to send the data from the component to the html template. The most common way to use property binding is that assign any property of the html element tag into the [] with the component property value, i.e.

```
<input type="text" [value]="data.name"/>
```

In the previous version of AngularJS, it can be done by using ng-bind directives. Basically property binding can be used to bind any property, component or a directive property. It can be used as

- a) Attribute binding
- b) Class binding
- c) Style binding

Code of app.component.interpolation.ts

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

@Component({
  moduleId: module.id,
  selector: 'interpolation',
  templateUrl: 'app.component.interpolation.html'
})

export class InterpolationComponent {
  value1: number = 10;
  array1: Array<number> = [10, 22, 14];
  dt1: Date = new Date();

  status: boolean = true;

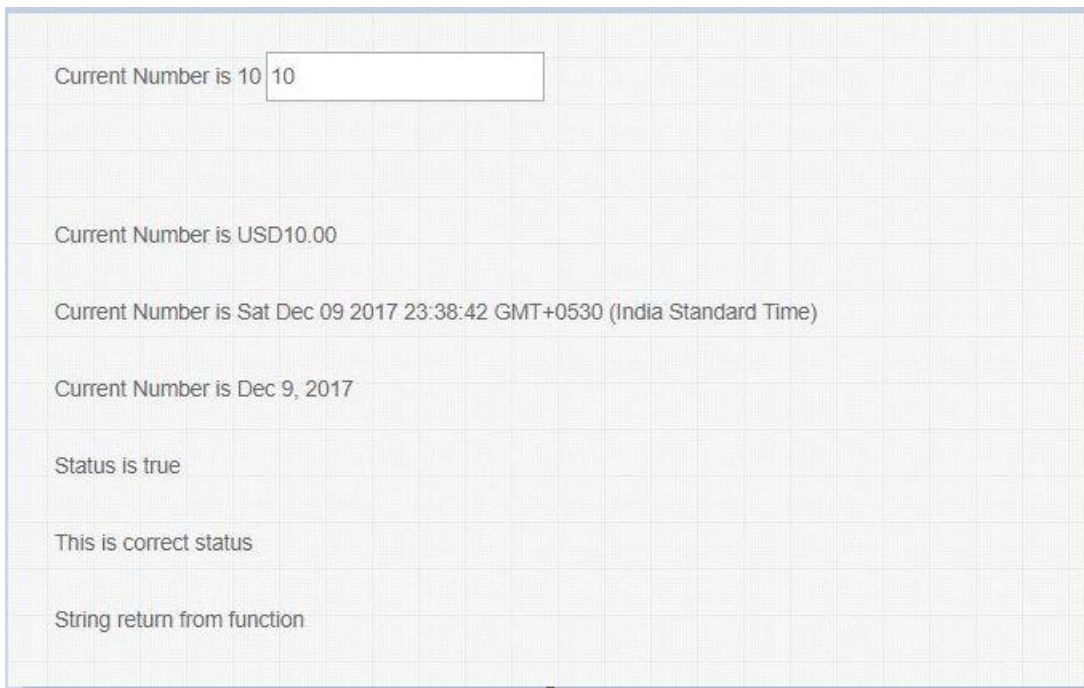
  returnString() {
    return "String return from function";
  }
}
```

Code of app.component.interpolation.html

```
<div>
  <span>Current Number is {{value1}}</span>
```

```
<input [value]="value1" />
<br /><br />
<br /><br />
<span>Current Number is {{value1 | currency}}</span>
<br /><br />
<span>Current Number is {{dt1}}</span>
<br /><br />
<span>Current Number is {{dt1 | date}}</span>
<br /><br />
<span>Status is {{status}}</span>
<br /><br />
<span>{{status ? "This is correct status" : "This is false status"}}</span>
<br /><br />
<span>{{returnString()}}</span>
</div>
```

Output is



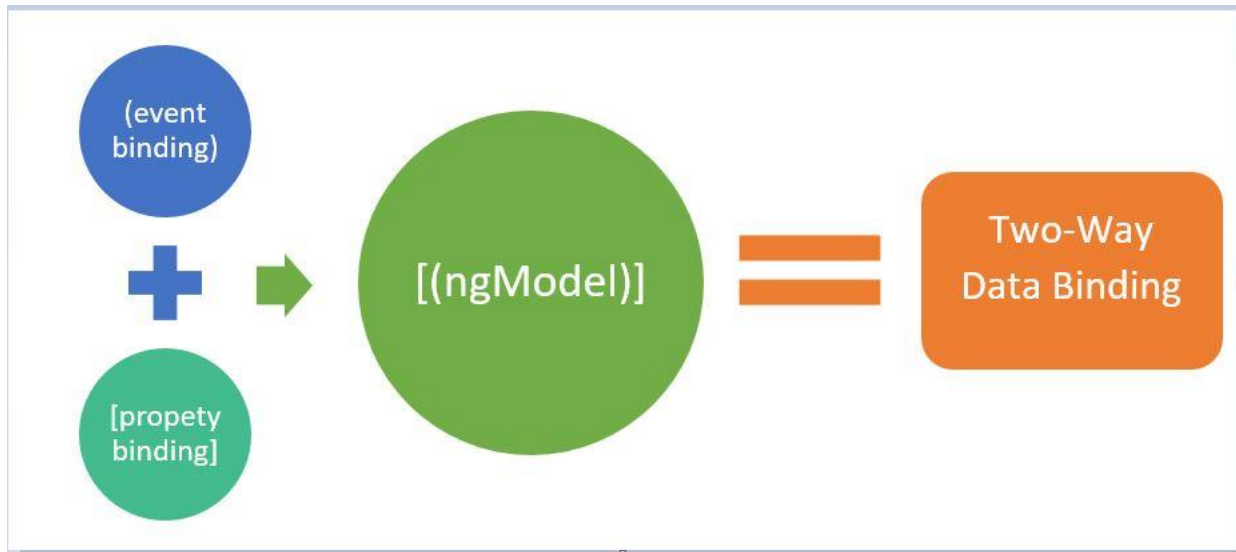
The screenshot shows a web application output with a light gray grid background. It displays the following text and input fields:

- Current Number is 10
- Current Number is USD10.00
- Current Number is Sat Dec 09 2017 23:38:42 GMT+0530 (India Standard Time)
- Current Number is Dec 9, 2017
- Status is true
- This is correct status
- String return from function

TWO WAY BINDING

The most popular and widely used data binding mechanism in two-way binding in the angular framework. Basically two-way binding mainly used in the input type field or any form element where user type or provide any value or change any control value in the one side, and on the other side, the same automatically updated in to the controller variables and vice versa. In the Angular 1.x, we use ng-model directives with the element for this purpose.

```
<input type="text" ng-model="firstName"/>
```



Similarly, in Angular 4.0 we have a directive called ngModel and it need to be used as below –

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

As we see, in Angular2 the syntax is different. Reason is that, we use [] since it is actually a property binding and parentheses is used for the event binding concept. I will discuss about ngModel directives in the later section of this article.

Code of app.component.twowaybind.ts

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

@Component({
  moduleId: module.id,
  selector: 'twowaybind',
  templateUrl: 'app.component.twowaybind.html'
})

export class TwoWayBindComponent {
  val: string = "";
}
```

Code of app.component.twowaybind.html

```
<div>
  <div>
    <span>Enter Your Name </span>
    <input [(ngModel)]="val" type="text" />
  </div>
  <div>
    <span>Your Name :- </span>
    <span>{{val}}</span>
  </div>
</div>
```

The Output is –



EVENT BINDING

Event Binding is one of new mechanism which introduced by angular 4.0 in a new concept. In the previous version of the angular js, we always use different types of directives like ng-click, ng-blur for bind any particular event action of a html control. But in angular2, we need to use the same property of the HTML element (like click, change etc) and use it within parenthesis. So, in angular 4, with properties we use square brackets and in events we use parenthesis.

```
<button (click)="showAlert();">Click</button>
```

Code of app.component.eventbinding.ts

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

@Component({
  moduleId: module.id,
  selector: 'event-bind',
  templateUrl: 'app.component.eventbinding.html'
})

export class EventBindingComponent implements OnInit {
  showAlert() {
```

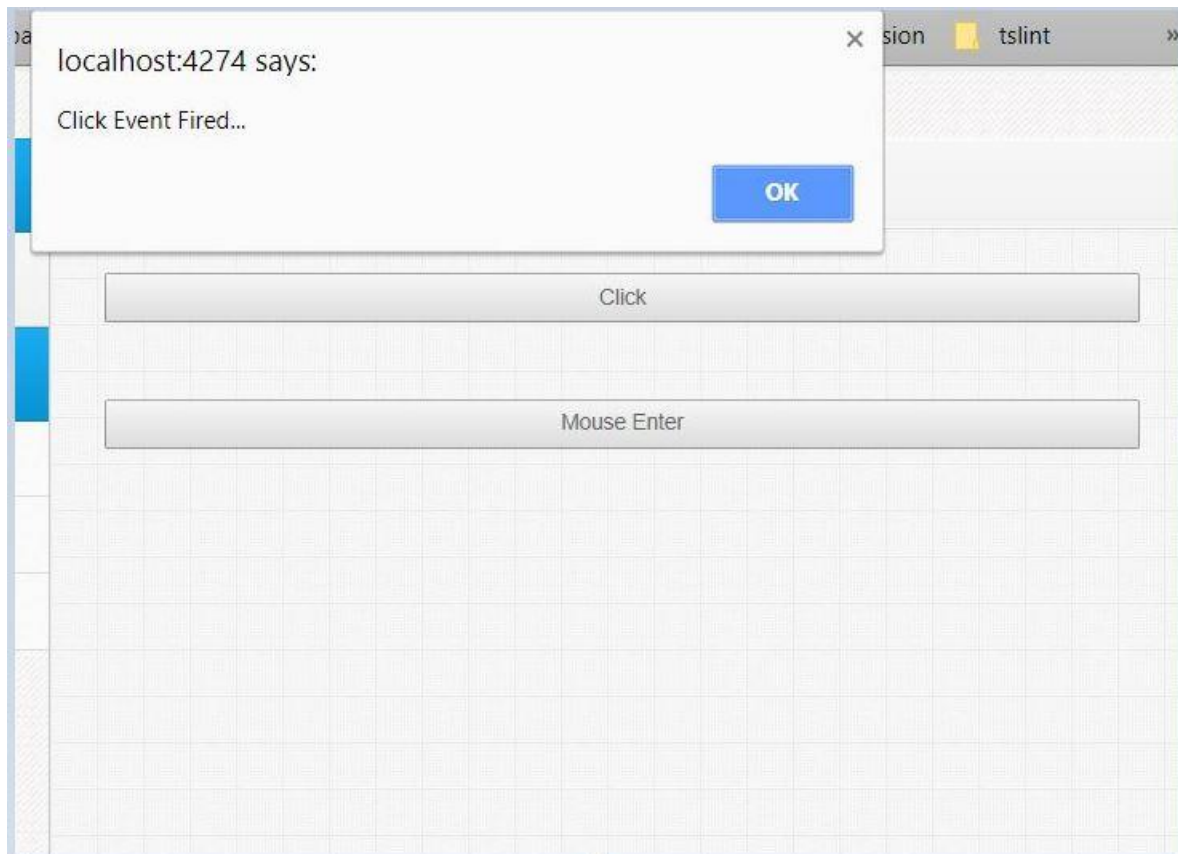


```
        console.log('You clicked on the button...');  
        alert("Click Event Fired...");  
    }  
  
    ngOnInit() {  
    }  
}
```

Code of app.component.eventbinding.html

```
<div>  
    <input type="button" value="Click" class="btn-block" (click)="showAlert()" />  
    <br /><br />  
    <input type="button" value="Mouse Enter" class="btn-block" (mouseenter)="showAlert()" />  
</div>
```

The Output is =



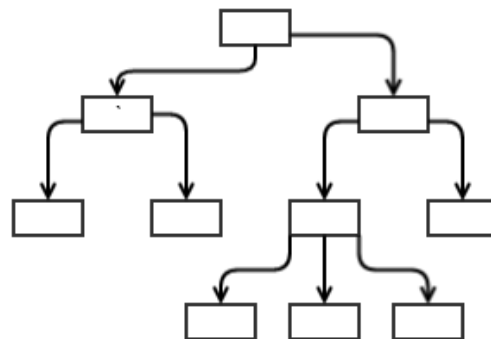
WHAT IS NGMODEL

As I already said in the previous section of the article, that ngModel basically performs both property binding and event binding. Actually property binding of the ngModel (i.e. [ngModel]) perform the activity to update the input element with value. Where as (ngModel) (basically in fired the (ngModelChange) event) instruct the outside world when there are any change occurred in the DOM Element.

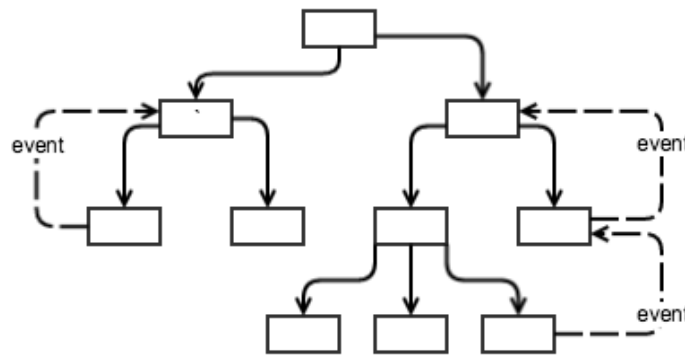
COMPONENT HEIRARCHY

Actually in practical world, type script support decorators (sometimes it is referred as annotations) which is basically used to declaratively add or change any existing thing. For example, class decorators can add metadata to the class's constructor function or even alter how the class behaves. In real world, components are not native JavaScript entities, Angular provides a way to define a component by pairing a constructor function with a html based view. We can do this by defining a constructor function (in TypeScript it is defined as a class) and using a decorator to associate our view with the constructor. The decorator can also set various configuration parameters for the component. This magic is accomplished using the `@Component` decorator which we already saw in the previous section of this series.

The above scenario describes the basic architecture of an individual component but Angular 4.0 applications are always made up of a hierarchy of components – they actually begin with a root component that contains as descendants all the components used in the application (basically in every angular module, bootstraps component is actually act as a root component). Components are intended to be self-contained because we want to encapsulate our component functions and we don't want other code to arbitrarily reach into our components to read or change properties. Also, we don't want our component to affect another component written by someone else. At the same time components do need to exchange data. In Angular 4, a component can receive data from its parent as long as the receiving component has specifically said it is willing to receive data. Similarly, components can send data to their parents by trigger an event the parent listens for. Let's look at how the component hierarchy behaves. To begin, we can draw it as follows:



Each rectangular box in the above component represent a component and technically this representation is called “graph” – a data structure consisting of nodes and connecting “edges.” The arrows represent the data flow from one component to another and we can see that data flows in only one direction – from the top downwards to descendants. Also, note there are no paths that allow you to travel from one node, through other nodes and back to the one where you started. The official name for this kind of data structure is that a “directed acyclic graph”, i.e. it flows in only one direction and has no circular paths in it.



This kind of structure has some important features: it is predictable, it is simple to traverse and it is easy to see what is impacted when a change is made. For Angular's purposes, when data changes in one node, it is easy to find the downstream nodes that could be affected. A simple example of how this might be used is a table with rows containing customers and information about them in which a table component contains multiple individual row components that represent each customer. The table component could manage a record set containing all the customers and pass the data on an individual customer to each of the row components it contains. This works fine for simply displaying data but in the real world data will need to flow the other way – back up the hierarchy – such as when a user edits a row. In that case the row needs to tell the table component that the data for a row has changed so the change can be sent back to the server. The problem is that as diagrammed above, data only flows down the hierarchy, not back up. To ensure we maintain the simplicity of one-way data flow down the hierarchy, Angular 4 uses a different mechanism for sending data back up the hierarchy: events.

Now, when a child component takes an action that a parent needs to know about, the child fires an event that is caught by the parent. The parent can take any action it needs which might include updating data that will, through the usual one-way downwards data flow, update downstream components. By separating the downward flow of data from the upwards flow of data, things are kept simpler and data management performs well.

@INPUT() DECORATOR

In a component-driven application architecture we typically use stateful and stateless components. A key concept is having some form of "stateful" component that delegates down into a "stateless" child, or children, component(s). Using the same concept above with bindings, which relies on parent data, we need to tell Angular what is coming into our component. @Input is a decorator to mark an input property. It is used to define an input property to achieve component property binding. @Input decorator binds a property within one component (child component) to receive a value from another component (parent component). This is one way communication from parent to child. The component property should be annotated with @Input decorator to act as input property. A component can receive a value from another component using component property binding. Now we will see how to use @Input. It can be annotated at any type of property such as number, string, array or user defined class. To use alias for the binding property name we need to assign an alias name as @Input(alias). Find the use of @Input with string data type.

```
@Input() caption : string;
```

Now find array data type with @Input decorator. Here we are aliasing the property name. In the component property binding, alias name arrObjects will be used.

```
@Input('ctArray') arrObjects : Array<string>
```

@OUTPUT() DECORATOR

In Angular 4, @Output is a decorator to mark an output property. @Output is used to define output property to achieve custom event binding. @Output will be used with the instance of EventEmitter. Now find the complete example step by step. @Output decorator binds a property of a component to send data from one component (child component) to calling component (parent component). This is one way communication from child to parent component. @Output binds a property of the type of angular EventEmitter class. This property name becomes custom event name for calling component. @Output decorator can also alias the property name as @Output(alias) and now this alias name will be used in custom event binding in calling component. Find the @Output decorator using aliasing.

```
@Output('addData') addEvent = new EventEmitter<any>();
```

In the above code snippet addData will become custom event name. Now find @Output decorator without aliasing.

```
@Output() sendMsgEvent = new EventEmitter<string>();
```

Here sendMsgEvent will be custom event name.

Code of app.component.student.ts

```
import { Component, Input, Output, EventEmitter } from '@angular/core';
```

```
@Component({  
  moduleId: module.id,  
  selector: 'student-data',  
  templateUrl: 'app.component.student.html'  
})
```

```
export class StudentDataComponent {  
  
  @Input() private Caption1: string = '';  
  @Input() private Caption2: string = '';  
  @Input() private Caption3: string = '';  
  @Input() private Caption4: string = '';  
  
  @Input() private Placeholder1: string = '';  
  @Input() private Placeholder2: string = '';  
  
  private _model: object = {};
```

```
@Output() private onFormSubmit: EventEmitter<any> = new
EventEmitter<any>();

constructor() {

}

private onSubmit(): void {
    if (typeof (this._model) === "undefined") {
        alert("Form not Filled Up Properly");
    }
    else {
        alert("Data Is Correct");
        this.onFormSubmit.emit(this._model);
    }
}

private onClear(): void {
    this._model = {};
}
}
```

Code of app.component.student.html

```

<div class="form-horizontal">
  <h2 class="aligncenter">Student Details</h2><br />
  <div class="row">
    <div class="col-xs-12 col-sm-2 col-md-2">
      <span>{{Caption1}}</span>
    </div>
    <div class="col-xs-12 col-sm-4 col-md-4">
      <input type="text" id="txtFName" placeholder="{{Placeholder1}}"
[ (ngModel) ]="_model.FName" />
    </div>
    <div class="col-xs-12 col-sm-2 col-md-2">
      <span>{{Caption2}}</span>
    </div>
    <div class="col-xs-12 col-sm-4 col-md-4">
      <input type="text" id="txtLName" placeholder="{{Placeholder2}}"
[ (ngModel) ]="_model.LName" />
    </div>
  </div>
  <br />
  <div class="row">
    <div class="col-xs-12 col-sm-2 col-md-2">
      <span>{{Caption3}}</span>
    </div>
    <div class="col-xs-12 col-sm-4 col-md-4">
      <input type="date" id="txtDate" [ (ngModel) ]="_model.DOB" />
    </div>
    <div class="col-xs-12 col-sm-2 col-md-2">
      <span>{{Caption4}}</span>
    </div>
    <div class="col-xs-12 col-sm-4 col-md-4">
      <input type="date" id="txtAppDate"
[ (ngModel) ]="_model.ApplicationDate" />
    </div>
  </div>
  <br />
  <div class="row">
    <div class="col-xs-12 col-sm-2 col-md-2">
    </div>
    <div class="col-xs-12 col-sm-4 col-md-4">
    </div>
    <div class="col-xs-12 col-sm-2 col-md-2">
      <input type="button" value="Submit" class="btn btn-primary"
(click)="onSubmit()" />
    </div>
  </div>
</div>

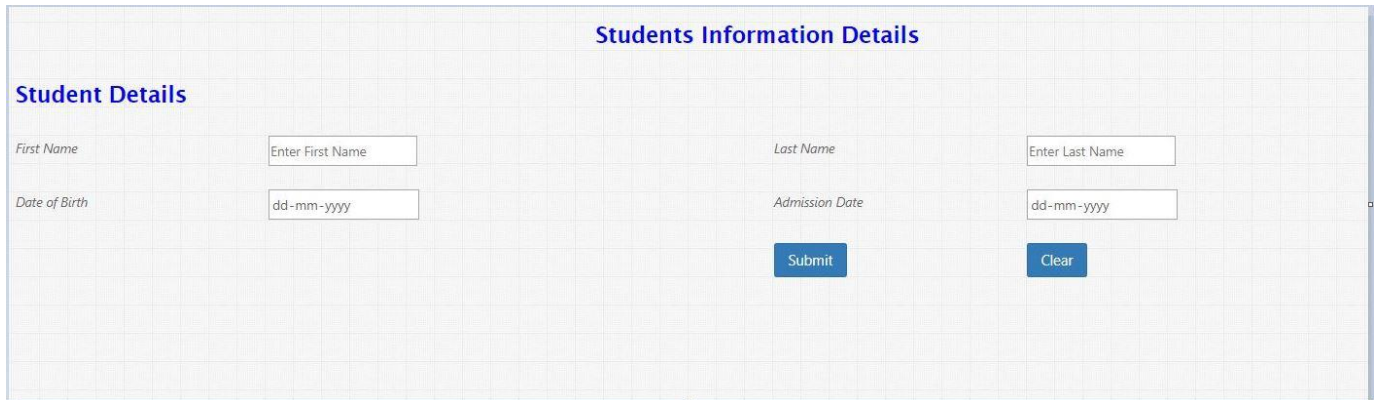
```

```

        </div>
        <div class="col-xs-12 col-sm-4 col-md-4">
            <input type="button" value="Clear" class="btn btn-primary"
(click)="onClear()" />
        </div>
    </div>
</div>

```

Output of Student Component –



Code of app.component.output.ts

```

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

@Component({
  moduleId: module.id,
  selector: 'student-form-output',
  templateUrl: 'app.component.output.html'
})

export class OutputDataComponent {
  private _firstNamePlaceholder: string = 'Enter First Name';
  private _lastNamePlaceholder: string = 'Enter Last Name';

  private _model: any;

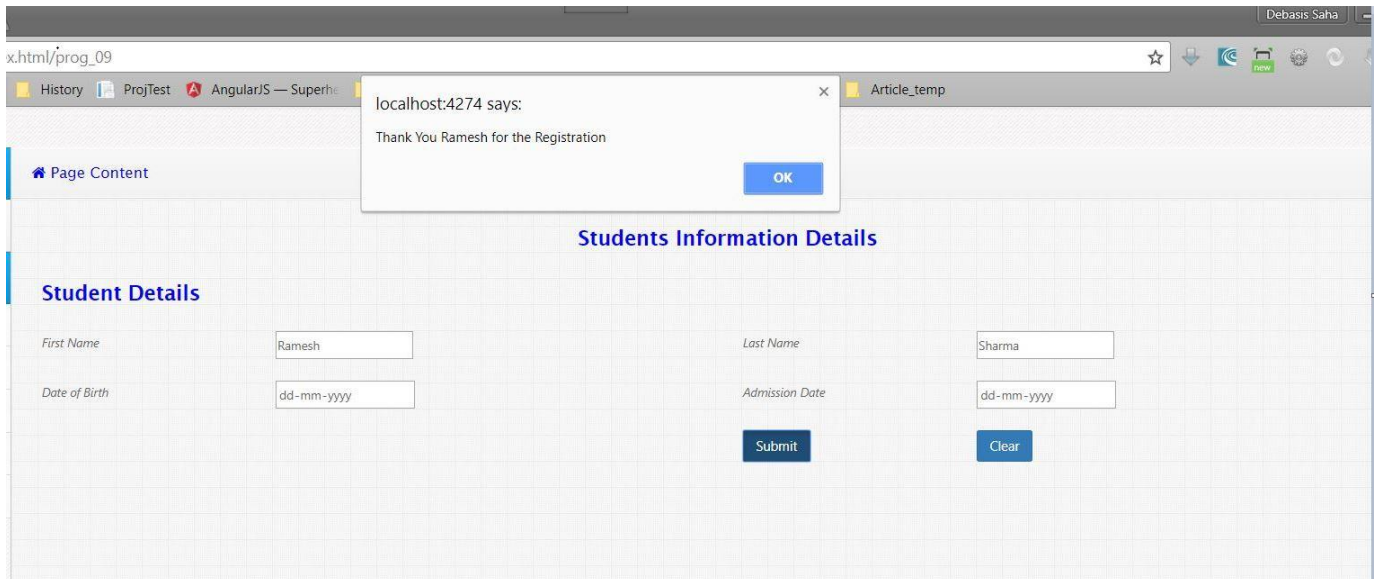
  private onSaveData(data: any): void {
    this._model = data;
    alert("Thank You " + this._model.FName + " for the Registration");
  }
}

```

Code of app.component.output.html

```
<div class="form-horizontal">
  <center> <h2>Students Information Details</h2></center>
  <br />
  <student-data [Caption1]='First Name' [Caption2]='Last Name'
[Caption3]='Date of Birth' [Caption4]='Admission Date'
[Placeholder1]='_firstNamePlaceholder'
[Placeholder2]='_lastNamePlaceholder'
(onFormSubmit)="onSaveData($event)"></student-data>
</div>
```

The Output is –

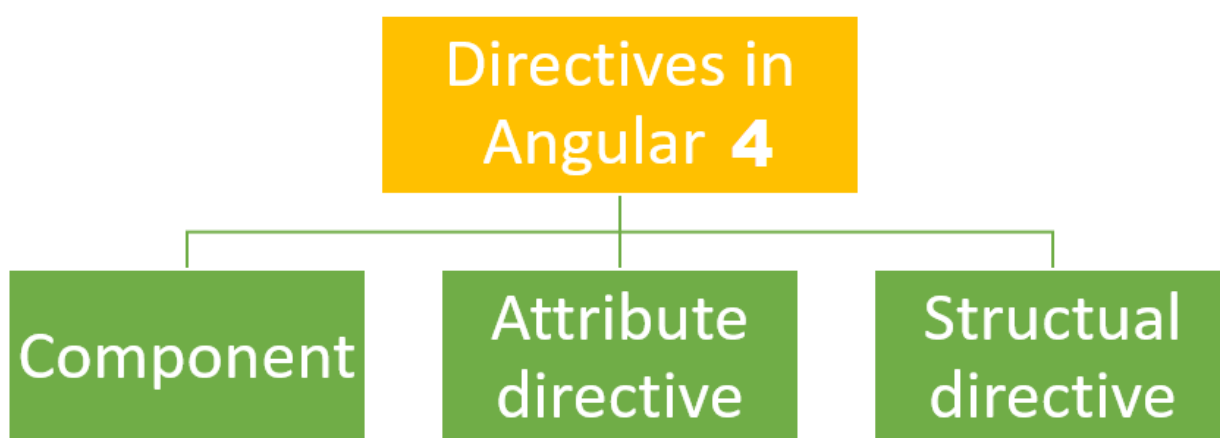


The screenshot shows a web browser window with a form titled "Students Information Details". The form is displayed on a grid background. It contains four input fields arranged in two rows. The first row has "First Name" with the value "Ramesh" and "Last Name" with the value "Sharma". The second row has "Date of Birth" with the value "dd-mm-yyyy" and "Admission Date" with the value "dd-mm-yyyy". Below the input fields are two buttons: "Submit" and "Clear". A modal dialog box is open in the center of the screen, displaying the message "localhost:4274 says: Thank You Ramesh for the Registration" and an "OK" button.

CHAPTER 5 : DIRECTIVES

A Directive modifies the DOM to change appearance, behavior or layout of DOM elements. Directives are one of the core building blocks Angular 4 uses to build applications. In fact, Angular 4 components are in large part directives with templates. From an Angular 1 perspective, Angular 4 components have assumed a lot of the roles directives used to. The majority of issues that involve templates and dependency injection rules will be done through components, and issues that involve modifying generic behavior is done through directives. There are three main types of directives in Angular 4:

1. **Component** - directive with a template.
2. **Attribute directives** - directives that change the behavior of a component or element but don't affect the template
3. **Structural directives** - directives that change the behavior of a component or element by affecting how the template is rendered.



Before the Component API shipped in Angular 1.5, directives with templates that were used as elements were defined in a similar way as an attribute directive using the same API, which could lead to messy directive declarations that can be hard to grok. Components use the directive API under the hood, but give us a cleaner interface for defining them by hiding a lot of the defaults that would otherwise be cumbersome.

COMPONENT VS DIRECTIVES

Since as per the above list, component itself act as directives. So as a programmer, it is very common to assume that component and directives are the same. But actually it is not true. Below the comparison of the Directives and component.

| Component | Directives |
|---|---|
| A component is register with the @Component Decorator | A Directives is register with the @Directives Decorator |

| | |
|---|--|
| Component is a directive which uses shadow DOM to create encapsulated visual behavior called components. Components are typically used to create UI widgets | Directive is used to add behavior to an existing DOM element |
| Component is used to break up the application into smaller components. | Directive is use to design re-usable components. |
| Only one component can be present per DOM element. | Many directives can be used per DOM element. |
| @View decorator or templateUrl template are mandatory in the component. | Directive doesn't use View. |

ATTRIBUTE DIRECTIVES

Attribute directives are a way of changing the appearance or behavior of a component or a native DOM element. Ideally, a directive should work in a way that is component agnostic and not bound to implementation details. Attribute directives actually modifies the appearance or behavior of an element. The attribute directive changes the appearance or behavior of a DOM element. These directives look like regular HTML attributes in templates. The ngModel directive which is used for two-way binding is an perfect example of an attribute directive. For create attribute directives, we always need to use or inject the below objects in our custom attribute directive component class. For creating an attribute directives, we have remember the below topics –

1. Import required modules like Directives, ElementRef and Renderer from Angular core library
2. Create a TypeScript Class
3. Use @Directive decorator in the class
4. Set the value of the selector property in @directive decorator function. The directive would be used, using the selector value on the elements.
5. In the constructor of the class, inject ElementRef and Renderer object.

We are injecting ElementRef in the directive's constructor to access the DOM element. We are also injecting Renderer in the directive's constructor to work with DOM's element style. We are calling the renderer's setElementStyle function. In the function, we pass the current DOM element by using the object of ElementRef and setting the behavior or property of the current element. We can use this attribute directive by its selector in our component:

ELEMENTREF

While creating custom attribute directive, we inject ElementRef in the constructor to access the DOM element. Actually elementref provide access to the underlying native element. **ElementRef** is a service that grants us direct access to the DOM element through its **nativeElement** property. That's all we need to set the element's color using the browser DOM API.

RENDERER

While creating custom attribute directive, we inject `Renderer` in the constructor to access the DOM element's style. Actually we call the `renderer's setElementStyle` function. In this function, we pass the current DOM element with the help of `ElementRef` object and set the required attribute of the current element.

HOSTLISTENER

Some times we may require to access input property within the attribute directive so that as per given attribute directive, we can apply related attribute within DOM Element. For trap user actions, we can call different methods to handle the user actions. To access the method for operate user actions we need to decorate the methods within the **@HostListener** method.

For example, Angular 4 has built-in attribute directives such as `ngClass` and `ngStyle` that work on any component or element.

NGSTYLE DIRECTIVE

Angular 4 provides a built-in directive, `ngStyle`, to modify a component or element's style attribute. Here's an example:

```
@Component({
  selector: 'app-style-example',
  template: `
    <p style="padding: 1rem"
      [ngStyle]="{
        'color': 'red',
        'font-weight': 'bold',
        'borderBottom': borderStyle
      }">
    <ng-content></ng-content>
    </p>
  `
})
export class StyleExampleComponent {
  borderStyle = '1px solid black';
}
```

Notice that binding a directive works the exact same way as component attribute bindings. Here, we're binding an expression, an object literal, to the `ngStyle` directive so the directive name must be enclosed in square brackets. `ngStyle` accepts an object whose properties and values define that element's style. In this case, we can see that both kebab case and lower camel case can be used when specifying a style property. Also notice that both the html style attribute and Angular 4 `ngStyle` directive are combined when styling the element. We can remove the style properties out of the template into the component as a property object, which then gets assigned to `NgStyle` using property binding. This allows dynamic changes to the values as well as provides the flexibility to add and remove style properties.

```
@Component({
  selector: 'app-style-example',
  template: `
    <p style="padding: 1rem"
      [ngStyle]="alertStyles">
    <ng-content></ng-content>
    </p>
  `,
})
export class StyleExampleComponent {
  borderStyle = '1px solid black';
  alertStyles = {
    'color': 'red',
    'font-weight': 'bold',
    'borderBottom': this.borderStyle
  };
}
```

NGCLASS DIRECTIVE

The ngClass directive changes the class attribute that is bound to the component or element it's attached to. There are a few different ways of using the directive.

Binding a String

We can bind a string directly to the attribute. This works just like adding an html class attribute. In this case, we're binding a string directly so we avoid wrapping the directive in square brackets. Also notice that the ngClass works with the class attribute to combine the final classes.

```
@Component({
  selector: 'app-class-as-string',
  template: `
    <p ngClass="centered-text underlined" class="orange">
    <ng-content></ng-content>
    </p>
  `,
  styles: [
    .centered-text {
      text-align: center;
    }
    .underlined {
      border-bottom: 1px solid #ccc;
    }
    .orange {

```

```

        color: orange;
    }
    `]
  })
  export class ClassAsStringComponent {
  }

```

Binding An Array

Here, since we are binding to the `ngClass` directive by using an expression, we need to wrap the directive name in square brackets. Passing in an array is useful when you want to have a function put together the list of applicable class names.

```

@Component({
  selector: 'app-class-as-array',
  template: `
    <p [ngClass]="['warning', 'big']">
    <ng-content></ng-content>
  </p>
  `,
  styles: [`
    .warning {
      color: red;
      font-weight: bold;
    }
    .big {
      font-size: 1.2rem;
    }
  `]
})
export class ClassAsArrayComponent {
}

```

Binding An Object

Lastly, an object can be bound to the directive. Angular 4 applies each property name of that object to the component if that property is true. Here we can see that since the object's `card` and `flat` properties are true, those classes are applied but since `dark` is false, it's not applied. `NgClass` Directive

```

@Component({
  selector: 'app-class-as-object',
  template: `
    <p [ngClass]="{ card: true, dark: false, flat: flat }">
    <ng-content></ng-content>
    <br>
  `
})

```

```

<button type="button" (click)="flat=!flat">Toggle Flat</button>
</p>
`,
styles: [`
.card {
border: 1px solid #eee;
padding: 1rem;
margin: 0.4rem;
font-family: sans-serif;
box-shadow: 2px 2px 2px #888888;
}
.dark {
background-color: #444;
border-color: #000;
color: #fff;
}
.flat {
box-shadow: none;
}
`]
})
export class ClassAsObjectComponent {
flat: boolean = true;
}

```

Code of app.component.attdirective.ts

```

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

@Component({
  moduleId: module.id,
  selector: 'attribute-directive',
  templateUrl: 'app.component.attdirective.html',
  styles: [".red {color:red;}", ".blue {color:blue}", ".cyan {color : cyan}"]
})

export class AttrDirectiveComponent implements OnInit {
  showColor: boolean = false;

  constructor() { }

  ngOnInit() { }

  changeColor(): void {
    this.showColor = !this.showColor;
  }
}

```

```

    }
}

```

Code of app.component.attrdirective.html

```

<div>
  <h3>This is a Attribute Directives</h3>
  <span [class.red]="true">Attribute Change</span><br />
  <span [ngClass]="{'blue':true}">Attribute Change by Using NgClass</span><br />
  <span [ngStyle]="{'font-size':'14px','color':'green'}">Attribute Change by
Using NgStyle</span>
  <br /><br />
  <span [class.cyan]="showColor">Attribute Change</span><br />
  <span [ngClass]="{'cyan':showColor}">Attribute Change by Using
NgClass</span><br />
  <input type="button" value="Change Color" (click)="changeColor()" />
  <br /><br />
  <span [class.cyan]="showColor">Attribute Change</span><br />
  <span [ngClass]="{'cyan':showColor, 'red' : !showColor}">Attribute Change
by Using NgClass</span><br />
  <br />
</div>

```

Output



STRUCTURAL DIRECTIVES

Attribute directives are a way of changing the appearance or behavior of a component or a native DOM element. Ideally, a directive should work. Structural Directives are a way of handling how a component or element renders through the use of the template tag. This allows us to run some code that decides what the final rendered output will be. Angular 4 has a few built-in structural directives such as `ngIf`, `ngFor`, and `ngSwitch`. Note: For those who are unfamiliar with the template tag, it is an HTML element with a few special properties. Content nested in a template tag is not rendered on page load and is something that is meant to be loaded through code at runtime. Structural directives have their own special syntax in the template that works as syntactic sugar.

```
@Component({
  selector: 'app-directive-example',
  template: `
    <p *structuralDirective="expression">
      Under a structural directive.
    </p>
  `
})
```

Instead of being enclosed by square brackets, our dummy structural directive is prefixed with an asterisk. Notice that the binding is still an expression binding even though there are no square brackets. That's due to the fact that it's syntactic sugar that allows using the directive in a more intuitive way and similar to how directives were used in Angular JS 1.

Angular 4 provides a built-in directive `ngIf`, `ngFor` and `ngSwitch`, to modify a component or element's style attribute.

NGIF

In Angular 1.x version, there was the `ng-show` and `ng-hide` directives which would show or hide the DOM elements on what the given expression evaluates by setting the display CSS property. In Angular 4.0, these two directives have been removed from the framework and introduced a new directive named `ngIf`. The main difference of `ngIf` directive over `ng-show` or `ng-hide` is that it actually removes the element or components entirely from DOM. For `ng-show` or `ng-hide`, Angular keeps the DOM elements/components in the page, so any component behaviors may keep running even the component is not visible in the page. In Angular 4.0, `ng-show` or `ng-hide` directive is not available but we can obtain the same functionality by using the `[style.display]` property of any element. Now, one question always arises in our mind that why Angular removes component or elements from DOM in case of `ngIf` directives? Actually, although in the earlier version, it hides or makes invisible the component or element, but still the elements or component are attached with DOM. So it continues to fire its event listener. Also it keeps changing while the model data has been changed due to model binding. So in this way, these invisible components or elements use resources which might be useful for some other place. The performance and memory burden can be substantial and the user may not be benefitted at all. Setting `ngIf` value to false does affect the component resource consumption. Angular removes the element from DOM, stops the changes detection for the associated component, detaches it from DOM events and destroys the components. The component can be garbage collected and

free up memory. Components often have child components which themselves have children. All of them has been destroyed when ngIf destroys the common ancestor.

Code of app.component.ngif.ts

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

@Component({
  moduleId: module.id,
  selector: 'toggle-text',
  templateUrl: 'app.component.ngIf.html'
})

export class NgIfComponent implements OnInit {
  showInfo: boolean = false;
  caption: string = 'Show Text';

  constructor() { }
  ngOnInit() { }

  changeData(): void {
    this.showInfo = !this.showInfo;
    if (this.showInfo) {
      this.caption = 'Hide Text';
    }
    else {
      this.caption = 'Show Text';
    }
  }
}
```

Code of app.component.ngif.html

```
<div>
  <input type="button" value="{{caption}}" (click)="changeData()"/>
  <br />
  <h2 *ngIf="showInfo"><span>Demonstrate of Structural Directives -
*ngIf</span></h2>
</div>
```

Output



NGFOR

The ngFor directives instantiates a template once per item from an iterable. The context of each instantiated template inherits from the outer context with the given loop variable. ngFor provides several exported values that can be used to local variables :-

index → will be set to the current loop iteration for each template context

first → will be set to a boolean value indicating whether the item is the first one in the iteration.

last → will be set to a boolean value indicating whether the item is the last one in the iteration.

even → will be set to a boolean value indicating whether this item has an even index.

odd → will be set to a boolean value indicating whether this item has an odd index

Angular uses object identity to track insertions and deletions within the iterator and reproduce those changes in the DOM. This has important implications for animations and any stateful controls (such as <input> elements which accept user input) that are present. Inserted rows can be animated in, deleted rows can be animated out, and unchanged rows retain any unsaved state such as user input.

Code for app.component.ngFor.ts

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

@Component({
  moduleId: module.id,
  selector: 'product-list',
  templateUrl: 'app.component.ngFor.html'
})

export class NgForComponent implements OnInit {
  productList: Array<string> = ['iPhone', 'Galaxy 7.0', 'Blackberry 10Z'];

  constructor() { }
  ngOnInit() { }
}
```

Code for app.component.ngFor.html

```
<div>
  <h2>Demonstrate ngFor</h2>
```

```

<ul>
  <li *ngFor="let item of productList">
    {{item}}
  </li>
</ul>
</div>

```

Output



NGSWITCH

The ngSwitch directives is actually compromise of two directives, an attribute directives and a structural directives. It is similar like switch statement in JavaScript or other languages. ngSwitch stamps our nested views when their match expression value matches the value of the switch expression. The expression bound to the directives defines what will compared against in the switch structural directives. If an expression bound to ngSwitchCase matches the one given to ngSwitch, those components are created and the others destroyed. If none of the cases match, then components that have ngSwitchDefault bound to them will be created and the others destroyed. Note that multiple components can be matched using ngSwitchCase and in those cases all matching components will be created. Since components are created or destroyed be aware of the costs in doing so.

Code for app.component.ngSwitch.ts

```

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

@Component({
  moduleId: module.id,
  selector: 'student-list',
  templateUrl: 'app.component.ngSwitch.html'
})

export class NgSwitchComponent implements OnInit {
  studentList: Array<any> = new Array<any>();

  constructor() { }
  ngOnInit() {
    this.studentList = [
      { SrlNo: 1, Name: 'Rajib Basak', Course: 'Bsc(Hons)', Grade: 'A' },

```

```

    { SrlNo: 2, Name: 'Rajib Basak1', Course: 'BA', Grade: 'B' },
    { SrlNo: 3, Name: 'Rajib Basak2', Course: 'BCom', Grade: 'A' },
    { SrlNo: 4, Name: 'Rajib Basak3', Course: 'Bsc-Hons', Grade: 'C' },
    { SrlNo: 5, Name: 'Rajib Basak4', Course: 'MBA', Grade: 'B' },
    { SrlNo: 6, Name: 'Rajib Basak5', Course: 'MSc', Grade: 'B' },
    { SrlNo: 7, Name: 'Rajib Basak6', Course: 'MBA', Grade: 'A' },
    { SrlNo: 8, Name: 'Rajib Basak7', Course: 'MSc.', Grade: 'C' },
    { SrlNo: 9, Name: 'Rajib Basak8', Course: 'MA', Grade: 'D' },
    { SrlNo: 10, Name: 'Rajib Basak9', Course: 'B.Tech', Grade: 'A' }
  ];
}
}

```

Code for app.component.ngSwitch.html

```

<div>
  <h2>Demonstrate ngSwitch</h2>
  <table style="width:100%;border:solid;border-color:blue;border-
width:thin;">
    <thead>
      <tr >
        <td>Srl No</td>
        <td>Student Name</td>
        <td>Course</td>
        <td>Grade</td>
      </tr>
    </thead>
    <tbody>
      <tr *ngFor="let student of studentList;"
[ngSwitch]="student.Grade">
        <td>
          <span *ngSwitchCase="'A'" [ngStyle]="{'font-
size':'18px','color':'red'}">{{student.SrlNo}}</span>
          <span *ngSwitchCase="'B'" [ngStyle]="{'font-
size':'16px','color':'blue'}">{{student.SrlNo}}</span>
          <span *ngSwitchCase="'C'" [ngStyle]="{'font-
size':'14px','color':'green'}">{{student.SrlNo}}</span>
          <span *ngSwitchDefault [ngStyle]="{'font-
size':'12px','color':'black'}">{{student.SrlNo}}</span>
        </td>
        <td>
          <span *ngSwitchCase="'A'" [ngStyle]="{'font-
size':'18px','color':'red'}">{{student.Name}}</span>

```

```

        <span *ngSwitchCase="'B'" [ngStyle]="{'font-
size':'16px','color':'blue'}">{{student.Name}}</span>
        <span *ngSwitchCase="'C'" [ngStyle]="{'font-
size':'14px','color':'green'}">{{student.Name}}</span>
        <span *ngSwitchDefault [ngStyle]="{'font-
size':'12px','color':'black'}">{{student.Name}}</span>
    </td>
    <td>
        <span *ngSwitchCase="'A'" [ngStyle]="{'font-
size':'18px','color':'red'}">{{student.Course}}</span>
        <span *ngSwitchCase="'B'" [ngStyle]="{'font-
size':'16px','color':'blue'}">{{student.Course}}</span>
        <span *ngSwitchCase="'C'" [ngStyle]="{'font-
size':'14px','color':'green'}">{{student.Course}}</span>
        <span *ngSwitchDefault [ngStyle]="{'font-
size':'12px','color':'black'}">{{student.Course}}</span>
    </td>
    <td>
        <span *ngSwitchCase="'A'" [ngStyle]="{'font-
size':'18px','color':'red'}">{{student.Grade}}</span>
        <span *ngSwitchCase="'B'" [ngStyle]="{'font-
size':'16px','color':'blue'}">{{student.Grade}}</span>
        <span *ngSwitchCase="'C'" [ngStyle]="{'font-
size':'14px','color':'green'}">{{student.Grade}}</span>
        <span *ngSwitchDefault [ngStyle]="{'font-
size':'12px','color':'black'}">{{student.Grade}}</span>
    </td>
</tr>
</tbody>
</table>
</div>

```

Output of the program

Demonstrate ngSwitch

| Sri No | Student Name | Course | Grade |
|--------|--------------|-----------|-------|
| 1 | Rajib Basak | Bsc(Hons) | A |
| 2 | Rajib Basak1 | BA | B |
| 3 | Rajib Basak2 | BCom | A |
| 4 | Rajib Basak3 | Bsc-Hons | C |
| 5 | Rajib Basak4 | MBA | B |
| 6 | Rajib Basak5 | MSc | B |
| 7 | Rajib Basak6 | MBA | A |
| 8 | Rajib Basak7 | MSc. | C |
| 9 | Rajib Basak8 | MA | D |
| 10 | Rajib Basak9 | B.Tech | A |

CHAPTER 6 : PIPES AND VIEWCHILD

When we want to develop any application, we always start the application with a simple basic common task i.e. retrieve data, transform data and then display those data in front of the user through our user interface. Retrieve data from any type of data source is totally depends on data service provider like web services, web api etc. So once data is arrived, we can push those raw data values directly to our user interface for viewing to the user. But sometimes, this is not happened exactly in the same manner. For example, in most use cases, users prefer to see a date in a simple format like 15/02/2017 rather than the raw string format Wed Feb 15 2017 00:00:00 GMT-0700 (Pacific Daylight Time). So it is clear from the above example, that some value requires a editing before viewing into the user interface. Also, it can be noticed that, same type of transformation required by us in many different user interface or vice versa. So, in this scenario we think about some style type property which we can create centrally and applied whenever we required globally. So for this purpose, Angular 4 introduce Angular Pipes, a definite way to write display – value transformations that we can declare in our HTML.

WHAT IS PIPES?

In Angular 4.0, Pipes are classes implementing a single function interface, accepting an input value with an optional parameter array, returning a transformed value.

TRANSITION FROM FILTER TO PIPES

In Angular 1.x, we are familiar with the term filter. Filters are a great way of returning new collection of data or formatting the new, rather done any changes or mutating existing. Filters are basically just a function, which takes a single value or collection of value as input and return a new value or collection of value based on the logical responsibilities. In Angular 4.0, pipes are the modernized version of filters. Most of the inbuild filters of Angular 1.x has been converted as pipes in Angular 4.0 with some new pipes. Pipes are accessed in our templates in the same way that filters were--with the "pipe" character |. Below table shows comparison of pipes or filters in both Angular 1.x and Angular 4.0.

In Angular 1.x version, filters are very helpful for formatting any type of value as output in our templates. So with Angular 4.0, we get the same feature with, but now it is known as Pipes. Below table shows comparison of pipes or filters in both Angular 1.x and Angular 4.0.

| Filter / Pipes Available | Angular 1.x | Angular 4.0 |
|--------------------------|-------------|-------------|
| currency | ✓ | ✓ |
| date | ✓ | ✓ |
| uppercase | ✓ | ✓ |
| lowercase | ✓ | ✓ |
| Json | ✓ | ✓ |
| limitTo | ✓ | ✓ |
| Number | ✓ | |
| orderBy | ✓ | |

| | | |
|---------|---|---|
| Filter | ✓ | |
| async | | ✓ |
| decimal | | ✓ |
| percent | | ✓ |

So we can call Pipes as a modernized version of Filters in Angular 1.x.

WHY PIPES?

Actually, Pipes does not give any new feature in our template. In Angular 4.0, we can use logic in the templates also. We can also execute or call a function to obtain the desired value in the template. Basically, pipes provide a sophisticated and handsome way to perform the above task within the templates. Basically, pipes make our code clean and structured.

Syntax of Pipes

myValue | myPipe:param1:param2 | mySecondPipe:param1

The pipe expression or syntax starts with the value followed by the symbol pipe (|), then the pipe name. The params for that pipe can be sent separated by colon (:) symbol. The order of execution is from left to right. However the pipe operator only works in your templates and not in JavaScript code. In JavaScript the pipe symbol works as bitwise operator.

USES OF PIPES

- We can display only some filtered elements from an array.
- We can modify or format the value.
- We can use them as a function.
- We can do all of the above combined.

FILTERS VS PIPES

In Angular 1.x, filters act as helpers, actually very similar to functions where we can pass the input and other parameters and it returns a formatted value. But in Angular 4.0, pipes work as an operator. The basic idea is, we have an input and we can modify the input applying more than one pipe in it. This not only simplifies the nested pipe logic, but also gave us a beautiful and clean syntax for our templates. Secondly, in case of async operations (which is newly introduced in Angular 4.0, we will later in this chapter), we need to set things manually in case of Angular 1.x filters. But pipes are smart enough to handle async operations.

BASIC PIPES

Most of the pipes provided by Angular 4.0 will be familiar with us if we already worked in Angular 1.x. Actually pipes does not provide any new feature in Angular 4.0. In Angular 4.0, we can use logics in the template. Also we can execute or fire any function within the template to obtain its return value. The pipe syntax starts with the actual input value followed by the pipe (|) symbol and then the pipe name. The parameters of that pipe can be sent separately by the colon (:) symbol. The order of execution of a pipe is right to left. Normally pipes work within our template and not in JavaScript code.

NEW PIPES

The **decimal** and **percent** pipes are new in Angular 4.0. These take an argument that indicate the digit information which should be used – that is how many integer and fraction digits the number should be formatted with. The argument we pass for formatting follows this pattern:

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

Code of app.component.inbuildpipe.ts

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

@Component({
  moduleId: module.id,
  selector: 'inbuild-pipe',
  templateUrl: 'app.component.inbuildpipe.html'
})

export class InBuildPipeComponent implements OnInit {
  private todayDate: Date;
  private amount: number;
  private message: string;

  constructor() { }

  ngOnInit(): void {
    this.todayDate = new Date();
    this.amount = 100;
    this.message = "Angular 4.0 is a Component Based Framework";
  }
}
```

Code of app.component.inbuildpipe.html

```
<div>
  <h1>Demonstrate of Pipe in Angular 4.0</h1>

  <h2>Date Format</h2>
  Full Date : {{todayDate}}<br />
  Short Date : {{todayDate | date:'shortDate'}}<br />
  Medium Date : {{todayDate | date:'mediumDate'}}<br />
  Full Date : {{todayDate | date:'fullDate'}}<br />
  Time : {{todayDate | date:'HH:MM'}}<br />
  Time : {{todayDate | date:'hh:mm:ss a'}}<br />
```

Time : {{todayDate | date:'hh:mm:ss p'}}

<h2>Number Format</h2>

No Formatting : {{amount}}

2 Decimal Place : {{amount | number:'2.2-2'}}

<h2>Currency Format</h2>

No Formatting : {{amount}}

USD Doller(\$) : {{amount | currency:'USD':true}}

USD Doller : {{amount | currency:'USD':false}}

INR() : {{amount | currency:'INR':true}}

INR : {{amount | currency:'INR':false}}

<h2>String Message</h2>

Actual Message : {{message}}

Lower Case : {{message | lowercase}}

Upper Case : {{message | uppercase}}

<h2> Percentage Pipes</h2>

2 Place Formatting : {{amount | percent :'.2'}}

</div>

Output

Demonstrate of Pipe in Angular 4.0

Date Format

Full Date : Mon Jan 08 2018 23:43:41 GMT+0530 (India Standard Time)

Short Date : 1/8/2018

Medium Date : Jan 8, 2018

Full Date : Monday, January 8, 2018

Time : 23:01

Time : 11:43:41 PM

Time : 11:43:41 p

Number Format

No Formatting : 100

2 Decimal Place : 100.00

Currency Format

No Formatting : 100

USD Doller(\$) : \$100.00

USD Doller : USD100.00

INR() : ₹100.00

INR : INR100.00

String Message

Actual Message : Angular 4.0 is a Component Based Framework

Lower Case : angular 4.0 is a component based framework

Upper Case : ANGULAR 4.0 IS A COMPONENT BASED FRAMEWORK

Percentage Pipes

2 Place Formatting : 10,000.00%

THE ASYNC PIPE

Angular 4.0 also introduced a special type of pipe called Async Pipe, which basically allow us to bind our templates directly to values that arrive asynchronously. This feature is great way for working with promises and observables (we will discuss about this in the later chapter). This type of pipes accepts a promise or observable as input value, updating the view with the appropriate value(s) when the promise is resolved or observable emits a new value. This can have a dual impact of reducing component-level code while also providing incredible performance optimization opportunities when utilized correctly. To see how this works, we'll create a simple promise and have it resolve with a string. After a 5 second delay, the value from the resolved promise will be displayed on the screen.

Code of app.component.asyncpipes.ts

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

@Component({
  moduleId: module.id,
  selector: 'async-pipe',
  templateUrl: 'app.component.asyncpipe.html'
})

export class AsyncComponent implements OnInit {
  private message: string;

  promise: Promise<any>;

  constructor() {
    this.promise = new Promise(function (resolve, reject) {
      setTimeout(function () {
        resolve("Hey, I'm from a promise.");
      }, 5000)
    });
  }

  ngOnInit(): void {

  }
}
```

Code of app.component.asyncpipes.html

```
<h1>Async Pipes</h1>
<h2>{{ promise | async}}</h2>
```

Output

Async Pipes

Hey, I'm from a promise.

CUSTOM

Now we can define a custom pipes in Angular 4.0. For configure custom pipes, we need to used pipes object. For this, we need to define custom pipe with @Pipe decorator and use it by adding a pipes property to the @View decorator with the pipe class name. We use the transform method to do any logic necessary to convert the value that is being passed in as input value. We can get a hold of the arguments array as the second parameter and pass in as many as we like from the template.

Code of app.pipes.propercase.ts

```
import { Pipe, PipeTransform } from "@angular/core"

@Pipe({
  name: 'propercase'
})

export class ProperCasePipe implements PipeTransform {
  transform(value: string, reverse: boolean): string {
    if (typeof (value) == 'string') {
      let intermediate = reverse == false ? value.toUpperCase() :
value.toLowerCase();
      return (reverse == false ? intermediate[0].toLowerCase() :
intermediate[0].toUpperCase()) + intermediate.substr(1);
    }
    else {
      return value;
    }
  }
}
```

Code of app.component.custompipe.ts

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

@Component({
  moduleId: module.id,
  selector: 'custom-pipe',
  templateUrl: 'app.component.custompipe.html'
})
export class CustomPipeComponent implements OnInit {
  private message: string;

  constructor() { }
  ngOnInit(): void {
    this.message = "This is a Custom Pipe";
  }
}
```

Code of app.component.custompipe.html

```
<div>
  <div class="form-horizontal">
    <h2 class="aligncenter">Custom Pipes - Proper Case</h2><br />
    <div class="row">
      <div class="col-xs-12 col-sm-2 col-md-2">
        <span>Enter Text</span>
      </div>
      <div class="col-xs-12 col-sm-4 col-md-4">
        <input type="text" id="txtFName" placeholder="Enter Text"
        [(ngModel)]="message" />
      </div>
    </div>
    <div class="row">
      <div class="col-xs-12 col-sm-2 col-md-2">
        <span>Result in Proper Case</span>
      </div>
      <div class="col-xs-12 col-sm-4 col-md-4">
        <span>{{message | propercase}}</span>
      </div>
    </div>
  </div>
</div>
```

Output



WHAT IS VIEWCHILD?

Basically, viewchild is one of new features which introduced in Angular 4.0 framework. Since Angular 4.0 is basically depends on component architecture. So when we try to develop any web page or UI, it is most obvious that that page or UI must be a component which basically contains a number of multiple different types of components within that component. So in simple word, it is basically a parent component – child component based architecture. In this scenario, there are some situations occurred when a parent component needs to interact with the child component. There are multiple ways to achieve this interaction between parent and child component. One the ways is ViewChild decorator. So if we want to get access to a child component, directive or DOM element from a parent component class, then we can use this ViewChild decorator. So when a parent component need to execute or call any method of the child component, it can inject child component as a viewchild within the parent component. ViewChild returns the first element that matches a given component, directive or template reference selector. In cases where you'd want to access multiple children, you'd use ViewChildren instead.

For implement ViewChild, we need to use @ViewChild decorator in the parent component. The @ViewChild decorator provide access to the class of child component from the parent component. The @ViewChild is a decorator function that takes the name of a component class as its input and finds its selector in the template of the containing component to bind to. @ViewChild can also be passed a template reference variable.

Now for illustrate this, We will develop a calculator type UI. In this, we will develop two component.

1. First component i.e. child component which contains the two textbox from taking inputs and four button for perform four basic mathematical operation. After completion of the operation, each button will emit its final value to parent component so that parent component can show those value or perform any other operations on basis of those value.
2. Now parent component want to clear the values from both own component and also from child component. For clear the child component value, parent component will access the child component's clear() by using the @ViewChild decorator.

Code of app.component.child.ts

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

@Component({
  moduleId: module.id,
  selector: 'child',
  templateUrl: 'app.component.child.html'
})

export class ChildComponent implements OnInit {
  private firstNumber: number = 0;
  private secondNumber: number = 0;
  private result: number = 0;

  @Output() private addNumber: EventEmitter<number> = new
  EventEmitter<number>();
  @Output() private subtractNumber: EventEmitter<number> = new
  EventEmitter<number>();
  @Output() private multiplyNumber: EventEmitter<number> = new
  EventEmitter<number>();
  @Output() private divideNumber: EventEmitter<number> = new
  EventEmitter<number>();

  constructor() { }

  ngOnInit(): void {
  }

  private add(): void {
    this.result = this.firstNumber + this.secondNumber;
    this.addNumber.emit(this.result);
  }

  private subtract(): void {
    this.result = this.firstNumber - this.secondNumber;
    this.subtractNumber.emit(this.result);
  }

  private multiply(): void {
    this.result = this.firstNumber * this.secondNumber;
    this.multiplyNumber.emit(this.result);
  }
}
```



```
<div class="ibox-content">  
  <div class="row">  
    <div class="col-md-4">  
      Enter First Number  
    </div>  
    <div class="col-md-8">  
      <input type="number" [(ngModel)]="firstNumber" />  
    </div>  
  </div>  
  <div class="row">  
    <div class="col-md-4">  
      Enter Second Number  
    </div>  
    <div class="col-md-8">  
      <input type="number" [(ngModel)]="secondNumber" />  
    </div>  
  </div>  
  <div class="row">  
    <div class="col-md-4">  
    </div>  
    <div class="col-md-8">  
      <input type="button" value="+" (click)="add()" />  
      &nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&~  
      <input type="button" value="-" (click)="subtract()" />  
      &nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&~  
      <input type="button" value="X" (click)="multiply()" />  
      &nbsp;&nbsp;&nbsp;&nbsp;&~  
      <input type="button" value="/" (click)="divide()" />  
    </div>  
  </div>
```

```
</div>  
</div>
```

Code of app.component.parent.ts

```
import { Component, OnInit, ViewChild } from '@angular/core';  
import { ChildComponent } from './app.component.child';  
  
@Component({  
  moduleId: module.id,  
  selector: 'parent',  
  templateUrl: 'app.component.parent.html'  
})  
  
export class ParentComponent implements OnInit {  
  private result: string = '';  
  
  @ViewChild('calculator') private _calculator: ChildComponent;  
  
  constructor() {  
  }  
  
  ngOnInit(): void {  
  }  
  
  private add(value: number): void {  
    this.result = 'Result of Addition ' + value;  
  }  
  
  private subtract(value: number): void {  
    this.result = 'Result of Subtraction ' + value;  
  }  
  
  private multiply(value: number): void {  
    this.result = 'Result of Multiply ' + value;  
  }  
  
  private divide(value: number): void {  
    this.result = 'Result of Division ' + value;  
  }  
  
  private reset(): void {  
    this.result = '';  
    this._calculator.clear();  
  }  
}
```

```

    }
}

```

Code of app.component.parent.html

```

<div>
  <h2>Simple Calculator</h2>
  <div>
    <child (addNumber)="add($event)" (subtractNumber)="subtract($event)"
(multiplyNumber)="multiply($event)"
      (divideNumber)="divide($event)" #calculator></child>
    </div>
    <h3>Result</h3>
    <span>{{result}}</span>
    <br />
    <br />
    <input type="button" value="Reset" (click)="reset()" />
  </div>

```

Output



CHAPTER 7 : WORKING WITH FORMS

ANGULAR 4.0 FORMS – WHAT IT IS?

In today's web application, a large category of frontend applications are very much form dependent, especially in the case of large enterprise type development. Most of these applications contain simply a huge or large form which contains multiple tabs, dialogs and with non-trivial business validation logic. Forms are a very important part of the applications. In a component-based application, we always want to split a form into a small and reusable piece of code which is stored within the Smart and Dumb components. These components are normally spread over the entire application which provides several architectural benefits including flexibility and design changes.

In the Angular 4.0 framework, we have two different mechanisms for related to form – binding.

1. Template Driven Form
2. Reactive or Model Driven Forms

In this article, we will discuss about the Template Driven Forms.

Angular 4 tackles forms via the famous ngModel. The instantaneous two-way data binding of ng-model in Angular 1 is really a life-saver as it allows to transparently keep in sync a form with a view model. Forms built with this directive can only be tested in an end-to-end test because this requires the presence of a DOM, but still this mechanism is very useful and simple to understand. Unlike the case of AngularJS 1.0, ngModel and other form-related directives are not available by default, we need to explicitly import them in our application module. To include the form module in our application, we need to inject the FormModule in our application and bootstrap it.

TEMPLATE DRIVEN FORMS FEATURES

- Easy to use
- Suitable for simple scenarios and fails for complex scenarios
- Similar to Angular 1.0
- Two-way data binding (using [(NgModel)]) syntax
- Minimal component code
- Automatic track of the form and its data
- Unit testing is another challenge

ADVANTAGES AND DISADVANTAGES OF TEMPLATE DRIVEN FORMS

In this simple example we cannot really see it, but keeping the template as the source of all form validation truth is something that can become pretty hairy rather quickly.

As we add more and more validator tags to a field or when we start adding complex cross-field validations the readability of the form decreases, to the point where it will be harder to hand it off to a web designer.

The up-side of this way of handling forms is its simplicity, and it's probably more than enough to build a very large range of forms.

On the downside the form validation logic cannot be unit tested. The only way to test this logic is to run an end-to-end test with a browser, for example using a headless browser like PhantomJS.

While using directives in our templates gives us the power of rapid prototyping without too much boilerplate, we are restricted in what we can do. Reactive forms on the other hand, lets us define our form through code and gives us much more flexibility and control over data validation.

ADVANTAGES OF MODEL DRIVEN FORMS

- **UNIT TESTABLE** : Since we have the form model defined in our code, we can unit test it.
- **LISTEN TO FORM AND CONTROLS CHANGES** : With reactive forms, we can listen to form or control changes easily. Each form group or form control expose a few events which we can subscribe to (e.g. statusChanges, valuesChanges, etc).

To begin, we must first ensure we are working with the right directives and the right classes in order to take advantage of procedural forms. For this, we need to ensure that the ReactiveFormsModule was imported in the bootstrap phase of the application module. This will give us access to components, directives and providers like FormBuilder, FormGroup, and FormControl.

Code of app.component.formdemo1.ts

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

@Component({
  moduleId: module.id,
  selector: 'form-demo-1',
  templateUrl: 'app.component.formdemo1.html'
})

export class FormDemo1Component implements OnInit {

  private formData: any = {};
  private showMessage: boolean = false;

  constructor() {
  }

  ngOnInit(): void {
  }

  registerUser(formdata: NgForm) {
    this.formData = formdata.value;
    this.showMessage = true;
  }
}
```

Code of app.component.formdemo1.html

```

<h2>Template Driven Form</h2>
<div>
  <form #signupForm="ngForm" (ngSubmit)="registerUser(signupForm)">
    <table style="width:60%;" cellpadding="5" cellspacing="5">
      <tr>
        <td style="width :40%;">
          <label for="username">User Name</label>
        </td>
        <td style="width :60%;">
          <input type="text" name="username" id="username"
[(ngModel)]="username" required>
        </td>
      </tr>
      <tr>
        <td style="width :40%;">
          <label for="email">Email</label>
        </td>
        <td style="width :60%;">
          <input type="text" name="email" id="email"
[(ngModel)]="email" required>
        </td>
      </tr>
      <tr>
        <td style="width :40%;">
          <label for="password">Password</label>
        </td>
        <td style="width :60%;">
          <input type="password" name="password" id="password"
[(ngModel)]="password" required>
        </td>
      </tr>
      <tr>
        <td style="width :40%;"></td>
        <td style="width :60%;">
          <button type="submit">Sign Up</button>
        </td>
      </tr>
    </table>
  </form>
  <div *ngIf="showMessage">
    <h3>Thanks You {{formData.username}} for registration</h3>
  </div>
</div>

```

Output



FORM CONTROL

Note that the `FormControl` class is assigned to similarly named fields, both on this and in the `FormBuilder#group({ })` method. This is mostly for ease of access. By saving references to the `FormControl` instances on this, you can access the inputs in the template without having to reference the form itself. The form fields can otherwise be reached in the template by using `loginForm.controls.username` and `loginForm.controls.password`. Likewise, any instance of `FormControl` in this situation can access its parent group by using its `.root` property (e.g. `username.root.controls.password`). A `FormControl` requires two properties: an initial value and a list of validators. Right now, we have no validation.

VALIDATING REACTIVE FORMS

Building from the previous login form, we can quickly and easily add validation. Angular provides many validators out of the box. They can be imported along with the rest of dependencies for procedural forms. We are using `.valid` and `.untouched` to determine if we need to show errors - while the field is required, there is no reason to tell the user that the value is wrong if the field hasn't been visited yet. For built-in validation, we are calling `.hasError()` on the form element, and we are passing a string which represents the validator function we included. The error message only displays if this test returns true.

REACTIVE FORMS CUSTOM VALIDATION

As useful as the built-in validators are, it is very useful to be able to include your own. Angular allows you to do just that, with minimal effort. A simple function takes the `FormControl` instance and returns null if everything is fine. If the test fails, it returns an object with an arbitrarily named property. The property name is what will be used for the `.hasError()` test.

```
<div [hidden]="!password.hasError('needsExclamation')">
  Your password must have an exclamation mark!
</div>
```

Code of app.component.formdemo2.ts

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

@Component({
  moduleId: module.id,
  selector: 'form-demo-2',
  templateUrl: 'app.component.formdemo2.html'
})

export class FormDemo2Component implements OnInit {

  private formData: any = {};

  username = new FormControl('', [
    Validators.required,
    Validators.minLength(5)
  ]);

  password = new FormControl('', [
    Validators.required,
    hasExclamationMark
  ]);

  loginForm: FormGroup = this.builder.group({
    username: this.username,
    password: this.password
  });

  private showMessage: boolean = false;

  constructor(private builder: FormBuilder) {
  }

  ngOnInit(): void {
  }

  registerUser() {
    this.formData = this.loginForm.value;
    this.showMessage = true;
  }
}
```



```
function hasExclamationMark(input: FormControl) {
    const hasExclamation = input.value.indexOf('!') >= 0;

    return hasExclamation ? null : { needsExclamation: true };
}
```

Code of app.component.formdemo2.html

```
<h2>Reactive Form Module</h2>
<div>
    <form [formGroup]="loginForm" (ngSubmit)="registerUser()">
        <table style="width:60%;" cellpadding="5" cellspacing="5">
            <tr>
                <td style="width :40%;">
                    <label for="username">User Name</label>
                </td>
                <td style="width :60%;">
                    <input type="text" name="username" id="username"
[formControl]="username">
                    <div [hidden]="username.valid || username.untouched">
                        <div>
                            The following problems have been found with the
username:

                                </div>
                                <div [hidden]="!username.hasError('minlength')">
                                    Username can not be shorter than 5 characters.
                                </div>
                                <div [hidden]="!username.hasError('required')">
                                    Username is required.
                                </div>
                            </div>
                        </td>
                    </tr>
                    <tr>
                        <td style="width :40%;">
                            <label for="password">Password</label>
                        </td>
                        <td style="width :60%;">
                            <input type="password" name="password" id="password"
[formControl]="password">
                            <div [hidden]="password.valid || password.untouched">
                                <div>
                                    The following problems have been found with the
password:

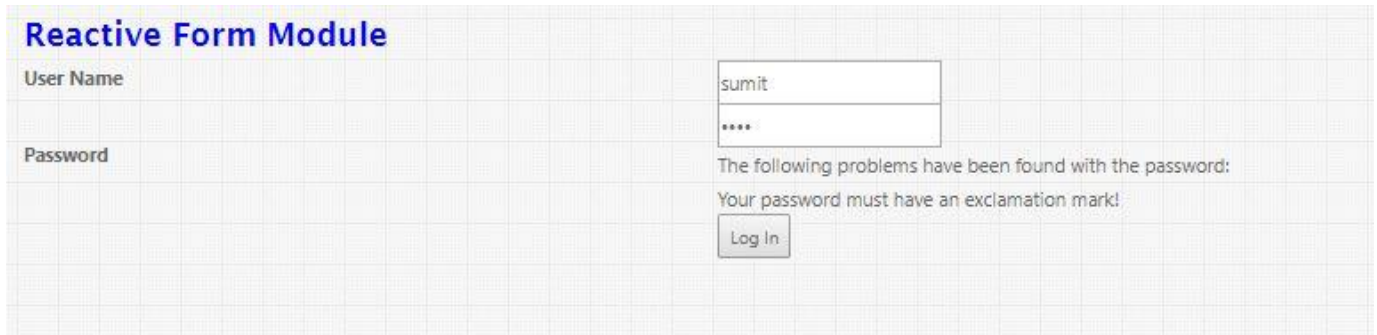
                                        </div>
                                    </div>
                                </div>
                            </td>
                        </tr>
                    </table>
                </div>
            </form>
```

```

        <div [hidden]="!password.hasError('required')">
            The password is required.
        </div>
        <div [hidden]="!password.hasError('needsExclamation')">
            Your password must have an exclamation mark!
        </div>
    </div>
</td>
</tr>
<tr>
    <td style="width :40%;"></td>
    <td style="width :60%;">
        <button type="submit" [disabled]="!loginForm.valid">Log
In</button>
    </td>
</tr>
</table>
</form>
<div *ngIf="showMessage">
    <h3>Thanks You {{formData.username}} for registration</h3>
</div>
</div>

```

Output



TRANSCULATION

In Angular 1.0, there is a concept of Transclusion. Actually, transclusion in an Angular 1.x is represent the content replacement such as a text node or html, and injecting it into a template at a specific entry time. Same thing in Angular 4.0 is totally forbidden. This is now done in Angular 4.0 through modern web APIs such as shadow DOM which is known as content projection.

WHAT IS CONTENT PROJECTION?

So now we know what we are looking from an Angular 1.x perspective, so that we can easily migrate the same in Angular 4.0. Actually projection is a very important concept in Angular. It enables developer to develop or build reusable components and make the application more scalable and flexible.

In Web Components, we *had* the <content> element, which was recently deprecated, which acted as a Shadow DOM insertion point. Angular 4 allows Shadow DOM through the use of ViewEncapsulation. Early beta versions of Angular 4 adopted the <content> element, however due to the nature of a bunch of Web Component helper elements being deprecated, it was changed to <ng-content>. Actually, View encapsulation defines whether the template and styles defined within the component can affect the whole application or vice versa. Angular provides three encapsulation strategies:

1. **Emulated (default)** - styles from main HTML propagate to the component. Styles defined in this component's @Component decorator are scoped to this component only.
2. **Native** - styles from main HTML do not propagate to the component. Styles defined in this component's @Component decorator are scoped to this component only.
3. **None** - styles from the component propagate back to the main HTML and therefore are visible to all components on the page. Be careful with apps that have None and Native components in the application. All components with None encapsulation will have their styles duplicated in all components with Native encapsulation.

To illustrate ng-content that, suppose we have a children component –

```
@Component({
  selector: 'child',
  template: `
    <div>
      <h4>Child Component</h4>
      {{ _childInfo }}
    </div>
  `
})
export class ChildComponent {
  _childInfo = "Base Area";
}
```

What should we do if we want to replace {{_childInfo}} to any HTML that provided to ChildComponent? One tempting idea is to define an @Input containing the text, but what if you wanted to provide styled HTML, or other components? Trying to handle this with an @Input can get messy quickly, and this is where content projection comes in. Components by default support projection, and you can use the ngContent directive to place the projected content in your template.

So, change ChildComponent to use projection:

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

@Component({
  selector: 'child',
  template: `
    <div style="border: 1px solid blue; padding: 1rem;">
      <h4>Child Component</h4>
      <ng-content></ng-content>
    </div>
  `
})
export class ChildComponent {
}
```

Then, when we use ChildComponent in the template:

```
<child>
  <p>My <i>dynamic</i> content.</p>
</child>
```

This is telling Angular, that for any markup that appears between the opening and closing tag of <child>, to place inside of <ng-content></ng-content>. When doing this, we can have other components, markup, etc projected here and the ChildComponent does not need to know about or care what is being provided.

But what if we have multiple <ng-content></ng-content> and want to specify the position of the projected content to certain ng-content? For example, for the previous ChildComponent, if we want to format the projected content into an extra area1 and area2 section. Then in the template, we can use directives, say, <area1> to specify the position of projected content to the ng-content with select="area1".

Code of app.component.modal.ts

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

@Component({
  moduleId: module.id,
  selector: 'modal-window',
  templateUrl: 'app.component.modal.html'
```

```

}))

```

```

export class ModalComponent implements OnInit {
    @Input() private display: string = 'none';
    @Input('header-caption') private header: string = 'Modal';

    constructor() {
    }

    ngOnInit(): void {
    }

    private fnClose(): void {
        this.display = 'none';
    }

    showModal(): void {
        this.display = 'block';
    }

    close(): void {
        this.fnClose();
    }

    setModalTitle(args: string): void {
        this.header = args;
    }
}

```

Code of app.component.modal.html

```

<div class="modal" id="myModal" tabindex="-1" role="dialog" aria-
labelledby="exampleModalLabel" aria-hidden="true" [ngStyle]="{'display' :
display }">
    <div class="modal-dialog">
        <div class="modal-content animated bounceInRight">
            <div class="modal-header">
                <button type="button" class="close"
(click)="fnClose()">&times;</button>
                <h3 class="modal-title">{{header}}</h3>
            </div>
            <div class="modal-body">
                <ng-content select="content-body"></ng-content>
            </div>
        </div>
    </div>

```

```

        </div>
        <div class="modal-footer">
            <ng-content select="content-footer"></ng-content>
        </div>
    </div>
</div>
</div>

```

Code of app.component.modaldemo.html

```

<div>
    <h2>Demonstrate Modal Window using ngContent</h2>
    <input type="button" value="Show Modal" class="btn-group"
(click)="fnOpenModal()" />
    <br />
    <modal-window [header-caption]="caption" #modal>
        <content-body>
            <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit.
Vestibulum tincidunt est vitae ultrices accumsan. Aliquam ornare lacus
adipiscing, posuere lectus et, fringilla augue.</p>
            <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit.
Vestibulum tincidunt est vitae ultrices accumsan. Aliquam ornare lacus
adipiscing, posuere lectus et, fringilla augue.</p>
            <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit.
Vestibulum tincidunt est vitae ultrices accumsan. Aliquam ornare lacus
adipiscing, posuere lectus et, fringilla augue.</p>
            <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit.
Vestibulum tincidunt est vitae ultrices accumsan. Aliquam ornare lacus
adipiscing, posuere lectus et, fringilla augue.</p>
            <p>Lorem ipsum dolor sit amet, consectetur adipiscing elit.
Vestibulum tincidunt est vitae ultrices accumsan. Aliquam ornare lacus
adipiscing, posuere lectus et, fringilla augue.</p>
        </content-body>
        <content-footer>
            <input type="button" class="btn-default active" class="btn btn-
primary" value="Modal Close" (click)="fnHideModal();" />
        </content-footer>
    </modal-window>
</div>

```

Code of app.component.modaldemo.ts

```
import { Component, OnInit, ViewChild } from '@angular/core';
import { ModalComponent } from './app.component.modal';

@Component({
  moduleId: module.id,
  selector: 'parent-content',
  templateUrl: 'app.component.modaldemo.html'
})

export class ModalDemoComponent implements OnInit {

  private caption: string = 'Custom Modal';
  @ViewChild('modal') private _ctrlModal: ModalComponent;

  constructor() {
  }

  ngOnInit(): void {
  }

  private fnOpenModal(): void {
    this._ctrlModal.showModal();
  }

  private fnHideModal(): void {
    this._ctrlModal.close();
  }
}
```

Output



CHAPTER 8 : SERVICE AND DEPENDENCY INJECTION

WHAT IS SERVICE?

An Angular 4 service is simply a JavaScript function, including with its related properties and methods which can perform a particular task or a group of task. Actually service is a mechanism to use share responsibilities within one or multiple components. As we already know, we can create components in angular 4 and nest multiple component together within a component using selector. Once our component are nested, we need to manipulate some data within the multiple components. In this case, service is the best way to handle this. Service is the best place where we can take data from other source or write down some calculations. Similarly service can shared between multiple components as we wants.

Angular 4.0 has greatly simplified the concept of Service over Angular 1.x. In Angular 1, there was service, factory, provider, delegate, value etc. and it was not always clear when to use which one. So for that reason, angular 4 simply changes the concept of the angular 4.0. There are simply two steps for creating service in angular 4.

1. Create a class with @Injectable decorator.
2. Register the class with provider or inject the class by using dependency injection.

In Angular 4.0, a service is basically used when a common functionality or business logic need to be provided or written need to shared in different name. Actually, service is totally a reusable objects. Assuming that our Angular application has contains some of the components performing the logging for error tracking purpose. In such a case, you will end up have a error log method in each of these components. This obviously is a bad design approach as the error log method is duplicated across components. If you want to change the semantics of error logging then you will need to change the code in all these components which in turn will impact the whole application. So a good design approach will be to have a common service component performing the logging feature. The log method in each of the components will be removed and placed in the specific logging service class. The components can use the logging feature by injecting the logging service. Service injection is one form of dependency injection provided by Angular.

DESIGN CONSIDERATIONS

In Angular 4.0, Services can be used to interact with data or implement business logic as per requirement. It can be used to provide connection to database or define functions that operate on data like CRUD operations. One can also design the service to provide a remote access to a web resource. It can be used to provide common integration needs like invoking a API of another application. Services can also be an excellent choice to design communication between components of your Angular application. You can build a service that provides a common event handling platform. Last but not the least, one can centralize common business functions used across the application as services. There are various approaches one can take to design a service. While designing services, think common functionality that every aspect of your application uses and transform them into services.

CREATE A SERVICE

So we can create a custom service as per our requirement. For create a service, we need to follow the below steps –

1. First we need to create a type script file with proper naming.
2. Now Create a type script class with the proper name which will represent the service after a while.
3. Use `@Injectable` decorator at the beginning of the class name which imported from the `@angular/core` packages. Basically the meaning of `@Injectable` is that custom service and any of its dependency can be automatically injected by the other components.
4. Though for design readability Angular recommends that you always define the `@Injectable` decorator whenever you create any service.
5. Now use the `Export` keyword against the class objects so that this service can be injectable or reused any other components.
6. One more important thing is that when we create our custom service available to the whole application through the use of providers meta data. This providers meta data must be defined in the `app.module.ts`(main application module file) file. If you provide the service in a main module file then it is visible to whole application. If you provide it in any component then only that component can use the service. By providing the service at the module level, Angular creates only one instance of CustomService class which can be used by all the components in an application.

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

    alert(message: string) {
        alert(message);
    }

    constructor() { }

}
```

@INJECTABLE

`@Injectable` is actually is a decorator. Decorator are a proposed extension in JavaScript. In short decorator provide the facility to programmer to modify or use methods, classes, properties and parameters. Injectables are just normal classes (normal objects) and as such, they have no special lifecycle. When an object of your class is created, the class's constructor is called, so that's what your "OnInit" would be. As for the destruction, a service does not really get destroyed. The only thing that might happen is that it gets garbage collected once there is no longer a reference to it, which likely happens after the dependency injector is removed itself. But you generally have no control over it, and there is no concept of a deconstructor in JavaScript.

`@Injectable()` lets Angular know that a *class* can be used with the dependency injector. `@Injectable()` is not *strictly* required if the class has *other* Angular decorators on it or does not have any dependencies.

What is important is that any class that is going to be injected with Angular *is decorated*. However, best practice is to decorate injectables with `@Injectable()`, as it makes more sense to the reader.

```
@Injectable()
export class SampleService {
    constructor() {
        console.log('Sample service is created');
    }
}
```

Just like most of the other server and client side frameworks, Angular 4 also has the concept of **Dependency Injection or DI**. Instances of services or classes can be injected using the constructor. Angular provides the concept of providers meta data to provide a service or any class as a candidate for automatic injection, at runtime, by defining that service or a class in a constructor of the dependent class.

DEPEWHAT IS DEPENDENCY INJECTION

Actually dependency injection is an important and usefull application design pattern. Angular 4.0 has its own dependency injection framework. Basically, it is a coding pattern in which classes receives its dependencies from external sources rather than creating them himself. In software engineering, dependency injection is a technique whereby one object supplies the dependencies of another object. A dependency is an object that can be used (a service). An injection is the passing of a dependency to a dependent object (a client) that would use it. Dependency Injection (DI) is a software design pattern that deals with how components get hold of their dependencies. The AngularJS injector subsystem is in charge of creating components, resolving their dependencies, and providing them to other components as requested.

DEPENDENCY INJECTION IN ANGULAR 4.0

Dependency injection has always been one of Angular's biggest features and selling points. It allows us to inject dependencies in different components across our applications, without needing to know, how those dependencies are created, or what dependencies they need themselves. However, it turns out that the current dependency injection system in Angular 1.x has some problems that need to be solved in Angular 2.x, in order to build the next generation framework. Below picture demonstrate the angular 2.0 dependency injection process.

Actually from the beginning, Dependency Injection is one of the biggest key features and selling points in Angular Js. It allows us to inject dependencies in different components across our applications, without needing to know, how those dependencies are created, or what dependencies they need themselves. However, it turns out that the current dependency injection system in Angular 1.x has some problems that need to be solved in Angular 2.x, in order to build the next generation framework. In this article, we're going to explore the new dependency injection system for future generations. So before going to create the custom service in example, we first understand what dependency injection is, and what is the advantages or disadvantages in Angular.

DEPENDENCY INJECTION AS A DESIGN PATTERN

In ng-conf 2014, Vojta Jina gives a great speech on the dependency injection and also described the story about the new dependency injection in the AngularJS. Basically we can use dependency injection in two way – as a design pattern and as a framework. So first we will discuss about dependency injection as a pattern. We first look at the below code and then analyzing the problems it introduces –

```
class car
{
    constructor()
    {
        this.engine = new Engine();
        this.tires = Tires.getInstance();
        this.color = app.get('colors');
    }
}
```

Nothing special here. We have a class Car that has a constructor in which we set up everything we need in order to construct a car object once needed. But what is the Problem? Well, as we can see, the constructor not only assigns needed dependencies to internal properties, it also knows how those object are created. For example the engine object is created using the Engine constructor, Tires seems to be a singleton interface and the doors are requested via a global object that acts as a **service locator**.

This leads to code that is hard to maintain and even harder to test. Just imagine you'd like to test this class. How would you replace Engine with a MockEngine dependency in that code? When writing tests, we want to test different scenarios that our code is used in, hence each scenario needs its own configuration. If we want to write testable code, we need to write reusable code. Our code should work in any environment as long as all dependencies are satisfied. Which brings us to the conclusion that **testable code is reusable code** and vice versa.

So how can we write this code better and make it more testable? It's super easy and you probably already know what to do. We change our code to this:

```
class Car {
    constructor(engine, tires, colors) {
        this.engine = engine;
        this.tires = tires;
        this.colors = colors;
    }
}
```

All we did is we moved the dependency creation out of the constructor and extended the constructor function to expect all needed dependencies. There are no concrete implementations anymore in this code, we literally moved the responsibility of creating those dependencies to a higher level. If we want to create a car object now, all we have to do is to pass all needed dependencies to the constructor:

```
var car = new car(new Engine(), new Tires(), new Colors);
```

How cool is that? The dependencies are now decoupled from our class, which allows us to pass in mocked dependencies in case we're writing tests:

```
var car = new car(new MockEngine(), new MockTires(), new MockColors);
```

And guess what, **this is dependency injection**. To be a bit more specific, this particular pattern is also called **constructor injection**. There are two other injection patterns, setter injection and interface injection, but we won't cover them in this article.

Okay cool, now we use DI, but when comes a DI **system** into play? As mentioned before, we literally moved the responsibility of dependency creation to a higher level. And this is exactly what our new problem is. Who takes care of assembling all those dependencies for us? It's us.

```
function main() {  
    var engine = new Engine();  
    var tires = new Tires();  
    var colos = new Colors();  
    var car = new Card(engine,tires,colors);  
    car.drive();  
}
```

We need to maintain a main function now. Doing that manually can be quite hairy, especially when the application gets bigger and bigger. Wouldn't it be nice if we could do something like this?

```
function main() {  
    var injector = new Injector(.....);  
    var car = injector.get(Car);  
    car.drive();  
}
```

DEPENDENCY INJECTION AS A FRAMEWORK

This is where dependency injection as a framework comes in. As we all know, Angular 4 has it's own DI system which allows us to annotate services and other components and let the injector find out, what dependencies need to be instantiated. This is all cool but it turns out, that the existing DI has some problem though:

- **Internal cache** - Dependencies are served as singletons. Whenever we ask for a service, it is created only once per application lifecycle. Creating factory machinery is quite hairy.

- **Namespace collision** - There can only be one token of a “type” in an application. If we have a car service, and there’s a third-party extension that also introduces a service with the same name, we have a problem.
- **Built into the framework** - Angular 4’s DI is baked right into the framework. There’s no way for us to use it decoupled as a standalone system.

Dependency Injection basically consists three things –

1. **Injector** – The Injector object that exposes APIs to us to create instances of dependencies
2. **Provider** – A Provider is like a commander that tells the injector how to create an instance of a dependency. A provider takes a token and maps that to a factory function that creates an objects.
3. **Dependency** – A Dependency is the type of which an object should be created.

So as per the above discussion, dependency injection is responsible for the below issues –

- Registering a class, function or value. These items, in the context of dependency injection, are called “providers” because they result in something. For example, a class is used to provide or result in an instance – see below for more details on provider types
- Resolving dependencies between providers – for example, if one provider requires another provider
- Making the provider’s result available in code when we ask for it. This process of making the provider result available to a block of code is called “injecting it.” The code that injects the provider results is, logically enough, called an “injector.”
- Maintaining a hierarchy of injectors so that if a component asks for a provider result from a provider not available in its injector, DI searches up the hierarchy of injectors

WHAT ARE PROVIDERS?

So now one question arises after the above discussion that what are these providers that injectors registers in each level? Basically it is very simple – a provider is a resource or JavaScript thing or class that Angular uses to provide something we want to use :

- A class provider generates or provides an instance of a class
- A factory provider generates or provides whatever returns when we run specified function
- A value provider does not need to take up action to provide the result, it just returns value.

Sample of a Class

```
export class testClass {  
    public message:string = 'Hello from Service Class';  
    public count:number;  
    constructor(){  
        this.count=1;  
    }  
}
```

Okay, that's the class; now let's instruct Angular to use it to register a class provider so we can ask the dependency injection system to give us an instance to use in our code. We'll create a component, that will serve as the root component for our application. Adding the testClass provider to this component is straightforward:

- Import testClass
- Add it to the @Component providers property
- Add an argument of type "testClass" to the constructor

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

```
@Component({  
  module : module.id,  
  selector : 'test-prog',  
  template : '<h1>Hello {{_message}}</h1>',  
  providers : [testClass]  
})  
export class TestComponent{  
  private _message:string="";  
  constructor(private _testClass : testClass){  
    this._message = this._testClass.message;  
  }  
}
```

Under the covers, when Angular instantiates the component, the DI system creates an injector for the component which registers the testClass provider. Angular then sees the testClass type specified in the constructor's argument list and looks up the newly registered testClass provider and uses it to generate an instance which it assigns to "_testClass". The process of looking up the testClass provider and generating an instance to assign to "_testClass" is all Angular. It takes advantage of the TypeScript syntax to know what type to search for but Angular's injector does the work of looking up and returning the testClass instance.

Code of app.component.student.ts

```
import { Component, OnInit, ViewChild } from '@angular/core';
import { StudentService } from '../app.service.student';

@Component({
  moduleId: module.id,
  selector: 'student-details',
  templateUrl: 'app.component.student.html'
})

export class StudentComponent implements OnInit {

  private _model: any = {};
  private _source: Array<any>;

  constructor(private _service: StudentService) {
    this._source = this._service.returnStudentData();
  }

  ngOnInit(): void {
  }

  private submit(): void {
    if (this.validate()) {
      this._service.addStudentData(this._model);
      this.reset();
    }
  }

  private reset(): void {
    this._model = {};
  }

  private validate(): boolean {
    let status: boolean = true;
    if (typeof (this._model.name) === "undefined") {
      alert('Name is Blank');
      status = false;
      return;
    }
    else if (typeof (this._model.age) === "undefined") {
      alert('Age is Blank');
      status = false;
      return;
    }
  }
}
```



```

    }
    else if (typeof (this._model.city) === "undefined") {
        alert('City is Blank');
        status = false;
        return;
    }
    else if (typeof (this._model.dob) === "undefined") {
        alert('dob is Blank');
        status = false;
        return;
    }
    return status;
}
}

```

Code of app.component.student.html

```

<div>
  <h2>Student Form</h2>
  <table style="width:80%;">
    <tr>
      <td>Student Name</td>
      <td><input type="text" [(ngModel)]="_model.name" /></td>
    </tr>
    <tr>
      <td>Age</td>
      <td><input type="number" [(ngModel)]="_model.age" /></td>
    </tr>
    <tr>
      <td>City</td>
      <td><input type="text" [(ngModel)]="_model.city" /></td>
    </tr>
    <tr>
      <td>Student DOB</td>
      <td><input type="date" [(ngModel)]="_model.dob" /></td>
    </tr>
    <tr>
      <td></td>
      <td>
        <input type="button" value="Submit" (click)="submit()" />
      </td>
    </tr>
  </table>
  <input type="button" value="Reset" (click)="reset()" />
</div>

```

```

        </td>
    </tr>
</table>
<h3>Student Details</h3>
<div class="ibox-content">
    <div class="ibox-table">
        <div class="table-responsive">
            <table class="responsive-table table-striped table-bordered
table-hover">
                <thead>
                    <tr>
                        <th style="width:40%;">
                            <span>Student's Name</span>
                        </th>
                        <th style="width:15%;">
                            <span>Age</span>
                        </th>
                        <th style="width:25%;">
                            <span>City</span>
                        </th>
                        <th style="width:20%;">
                            <span>Date of Birth</span>
                        </th>
                    </tr>
                </thead>
                <tbody>
                    <tr *ngFor="let item of _source; let i=index">
                        <td><span>{{item.name}}</span></td>
                        <td><span>{{item.age}}</span></td>
                        <td><span>{{item.city}}</span></td>
                        <td><span>{{item.dob}}</span></td>
                    </tr>
                </tbody>
            </table>
        </div>
    </div>
</div>
</div>

```

Code of app.service.student.ts

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

@Injectable()
export class StudentService {
    private _studentList: Array<any> = [];

    constructor() {
        this._studentList = [{name:'Amit Roy', age:20, city:'Kolkata', dob:'01-01-1997'}];
    }

    returnStudentData(): Array<any> {
        return this._studentList;
    }

    addStudentData(item: any): void {
        this._studentList.push(item);
    }
}
```

Code of app.demo.module.ts

```
import { NgModule, NO_ERRORS_SCHEMA } from '@angular/core';
import { APP_BASE_HREF } from '@angular/common';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule } from "@angular/forms";
import { HttpClientModule } from '@angular/http';
import { RouterModule } from '@angular/router';

import { HomeComponent } from './SampleCode/app.component.home';
import { StudentService } from './samplecode/app.service.student';
import { StudentComponent } from './samplecode/app.component.student';

@NgModule({
    imports: [BrowserModule, FormsModule, HttpClientModule],
    declarations: [HomeComponent, StudentComponent],
    bootstrap: [HomeComponent],
    schemas: [NO_ERRORS_SCHEMA],
    providers: [{ provide: StudentService, useClass: StudentService }]
})

export class DemoModule { }
```

Output

Student Form

Student Name

Age

City

Student DOB

dd-mm-yyyy

Submit

Reset

Student Details

| Student's Name | Age | City | Date of Birth |
|----------------|-----|---------|---------------|
| Amit Roy | 20 | Kolkata | 01-01-1997 |
| Amit | 20 | Kolkata | 1992-07-10 |

CHAPTER 9 : AJAX EVENT HANDLING

Angular 4.0 introduces many innovative concept like performance improvements, the Component Routing, sharpened Dependency Injection (DI), lazy loading, async templating, mobile development with Native Script; all linked with a solid tooling and excellent testing support. Making HTTP requests in Angular 4.0 apps looks somewhat different than what we're used to from Angular 1.x, a key difference being that Angular 4's Http returns observables.

It is very much clear to us that Angular 4.0 always look and feels different than the Angular 1.x. In case of Http API calling, the same scenario occurred. The `$http` service which angular 1.x provides us works very nicely in most of the cases. Angular 4.0 Http requires that we need to learn some new concept or mechanism, including how to work with observables.

Reactive Extensions for JavaScript (RxJS) is a reactive streams library that allows you to work with Observables. RxJS combines Observables, Operators and Schedulers so we can subscribe to streams and react to changes using composable operations.

DIFFERENCES BETWEEN ANGULAR 1.X \$HTTP AND ANGULAR 2 HTTP

Angular 4's Http API calling again provides a fairly straightforward way of handling requests. For starters, HTTP calls in Angular 4 by default return observables through RxJS, whereas `$http` in Angular 1.x returns promises. Using observable streams gives us the benefit of greater flexibility when it comes to handling the responses coming from HTTP requests. For example, we have the potential of tapping into useful RxJS operators like `retry` so that a failed HTTP request is automatically re-sent, which is useful for cases where users have poor or intermittent network communication.

In Angular 4, Http is accessed as an injectable class from `angular4/http` and, just like other classes, we import it when we want to use it in our components. Angular 4 also comes with a set of injectable providers for Http, which are imported via `HTTP_PROVIDERS`. With these we get providers such as `RequestOptions` and `ResponseOptions`, which allow us to modify requests and responses by extending the base class for each. In Angular 1.x, we would do this by providing a `transformRequest` or `transformResponse` function to our `$httpOptions`.

OBSERVABLES VS PROMISES

When used with **Http**, both implementations provide an easy API for handling requests, but there are some key differences that make **Observables** a superior alternative:

- Promises only accept one value unless we compose multiple promises (Eg: `$q.all`).
- Promises can't be cancelled.

As we already discussed that in Angular 4.0, there are many new features introduced in Angular 4.0. An exciting new feature used with Angular is the Observable. This isn't an Angular specific feature, but rather a proposed standard for managing async data that will be included in the release of ES7. Observables open up a continuous channel of communication in which multiple values of data can be emitted over time. From this we get a pattern of dealing with data by using array-like operations to parse, modify and maintain data. Angular uses observables extensively - you'll see them in the HTTP service and the event system.

In version 4.0, angular mainly introduces reactive programming concept based on the observables for dealing the asynchronous processing of data. In angular 1.x, we basically used promises to handle the asynchronous processing. But still in the Angular 4.0, we can still use the promises for the same purpose. The main concept of reactive programming is the observable element which related to the entity that can be observed. Basically, at normal look, promises and observables seem to very similar to each other. Both of them allow us to execute asynchronous processing, register callbacks for both successful and error responses and also notify or inform us when the result is there.

HOW TO DEFINE OBSERVABLES

Before going to details example of observables, first we need to understand how to define an observables objects. To create an observable, we can use the create method of the Observable object. A function must be provided as parameter with the code to initialize the observable processing. A function can be returned by this function to cancel the observable.

```
var observable = Observable.create((observer) => {  
  
    setTimeout(() => {  
  
        observer.next('some event');  
  
    }, 500);  
  
});
```

Similar to promises, observables can produce several notifications using the different methods from the observer:

- **next** – Emit an event. This can be called several times.
- **error** – Throw an error. This can be called once and will break the stream. This means that the error callback will be immediately called and no more events or completion can be received.
- **complete** – Mark the observable as completed. After this, no more events or errors will be handled and provided to corresponding callbacks.

Observables allow us to register callbacks for previously described notifications. The subscribe method tackles this issue. It accepts three callbacks as parameters:

- The onNext callback that will be called when an event is triggered.
- The onError callback that will be called when an error is thrown.
- The onCompleted callback that will be called when the observable completes.

OBSERVABLE SPECIFICITIES

Since observables and promises have many similarities. In spite of that, observables provide us some new specifications like below –

- **Lazy** - An observable is only enabled when a first observer subscribes. This is a significant difference compared to promises. Due to this, processing provided to initialize a promise is always executed even if no listener is registered. This means that promises don't wait for subscribers to be ready to receive and handle the response. When creating the promise, the initialization processing is always immediately called. Observables are lazy so we have to subscribe a callback to let them execute their initialization callback.
- **Execute Several Times** - Another particularity of observables is that they can trigger several times unlike promises which can't be used after they were resolved or rejected.
- **Canceling Observables** - Another characteristic of observables is that they can be cancelled. For this, we can simply return a function within the initialization call of the Observable.create function. We can refactor our initial code to make it possible to cancel the timeout function.
- **Error Handling** - If something unexpected arises we can raise an error on the Observable stream and use the function reserved for handling errors in our subscribe routine to see what happened.

OBSERVABLES VS PROMISES

Both Promises and Observables provide us with abstractions that help us deal with the asynchronous nature of our applications. However, there are important differences between the two:

As seen in the example above, Observables can define both the setup and teardown aspects of asynchronous behavior.

OBSERVABLES ARE CANCELLABLE.

Moreover, Observables can be retried using one of the retry operators provided by the API, such as `retry` and `retryWhen`. On the other hand, Promises require the caller to have access to the original function that returned the promise in order to have a retry capability.

OBSERVABLE HTTP EVENTS

A common operation in any web application is getting or posting data to a server. Angular applications do this with the `Http` library, which previously used Promises to operate in an asynchronous manner. The updated `Http` library now incorporates Observables for triggering events and getting new data.

OBSERVABLES ARRAY OPERATIONS

In addition to simply iterating over an asynchronous collection, we can perform other operations such as filter or map and many more as defined in the RxJS API. This is what bridges an Observable with the iterable pattern, and lets us conceptualize them as collections.

Here are two really useful array operations - map and filter. What exactly do these do?

1. **map** will create a new array with the results of calling a provided function on every element in this array. In the below example we used it to create a new result set by iterating through each item and appending the word 'Mr' abbreviation in front of every employee's name.
2. **filter** will create a new array with all elements that pass the test implemented by a provided function. Here we have used it to create a new result set by excluding any user whose salary is below 20000. Now when our subscribe callback gets invoked, the data it receives will be a list of JSON objects whose id properties are greater than or equal to 20000 salary.

Note the chaining function style, and the optional static typing that comes with TypeScript, that we used in this example. Most importantly functions like filter return an Observable, as in Observables beget other Observables, similarly to promises. In order to use map and filter in a chaining sequence we have flattened the results of our Observable using flatMap. Since filter accepts an Observable, and not an array, we have to convert our array of JSON objects from data.json() to an Observablestream. This is done with flatMap

The Angular 4 http module @angular/http exposes a Http service that our application can use to access web services over HTTP. We'll use this utility in our PeopleService service. We start by importing it together with all types involved in doing an http request:

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

These are all types and method required to make and handle an HTTP request to a web service:

Http :- The Angular 4 http service that provides the API to make HTTP requests with methods corresponding to HTTP verbs like get, post, put, etc

Response - which represents a response from an HTTP service and follows the fetch API specification

Observable which is the async pattern used in Angular 4. The concept of observable comes from the observer design pattern as an object that notifies an interested party of observers when something interesting happens. In RxJs it has been generalized to manage sequences of data or events, to become composable with other observables and to provide a lot of utility functions known as operators that let you achieve amazing stuff.

Angular comes with its own HTTP library which we can use to call out to external APIs.

When we make calls to an external server, we want our user to continue to be able to interact with the page. That is, we don't want our page to freeze until the HTTP request returns from the external server. To achieve this effect, our HTTP requests are asynchronous.

Dealing with asynchronous code is, historically, more tricky than dealing with synchronous code. In JavaScript, there are generally three approaches to dealing with async code:

1. Callbacks
2. Promises
3. Observables

HTTP POST

Let's imagine we received the *username* and *password* from a form the user submitted. We would call *authenticate* to log in the user. Once the user is logged in we will proceed to store the token so we can include it in following requests.

```
http.post(url: string, body: string, options?: RequestOptionsArgs) : Observable<Response>
```

The *http.post* signature above reads as follows. We need to provide a url and a body, both strings and then optionally an options object. In our example we are passing the modified *headers* property. *http.post* returns an *Observable*, we use *map* to extract the JSON object from the response and *subscribe*. This will setup our stream as soon as it emits the result.

DIFFERENCES BETWEEN \$HTTP AND ANGULAR/HTTP

Angular 4 Http by default returns an *Observable* opposed to a *Promise* (\$q module) in \$http. This allows us to use more flexible and powerful RxJS operators like *switchMap* (*flatMapLatest* in version 4), *retry*, *buffer*, *debounce*, *merge* or *zip*. By using *Observables* we improve readability and maintenance of our application as they can respond gracefully to more complex scenarios involving multiple emitted values opposed to only a one-off single value.

Code of app.component.employeeList.ts

```
import { Component, OnInit, ViewChild, AfterViewInit } from '@angular/core';
import { Http, Response, Headers } from '@angular/http';
import 'rxjs/Rx';
```

```
@Component({
  moduleId: module.id,
  selector: 'employee-list',
  templateUrl: 'app.component.employeeList.html'
})
```

```
export class EmployeeListComponent implements OnInit {
```

```
  private data: Array<any> = [];
  private showDetails: boolean = true;
  private showEmployee: boolean = false;
  private editEmployee: boolean = false;
  private _selectedData: any;
```

```
private _deletedData:any;

constructor(private http: Http) {
}

ngOnInit(): void {
}

ngAfterViewInit(): void {
    this.loadData();
}

private loadData(): void {
    let self = this;
    this.http.request('http://localhost:81/SampleAPI/employee/getemployee')
        .subscribe((res: Response) => {
            self.data = res.json();
        });
}

private addEmployee(): void {
    this.showDetails = false;
    this.showEmployee = true;
}

private onHide(args: boolean): void {
    this.showDetails = !args;
    this.showEmployee = args;
    this.editEmployee = args;
    this.loadData();
}

private onUpdateData(item:any):void{
    this._selectedData = item;
    this._selectedData.DOB = new Date(this._selectedData.DOB);
    this._selectedData.DOJ = new Date(this._selectedData.DOJ);
    this.showDetails = false;
    this.editEmployee = true;
}

private onDeleteData(item:any):void{
    this._deletedData = item;
    if(confirm("Do you want to Delete Record Permanently?")){
        let self = this;
    }
}
```

```

let headers = new Headers();
headers.append('Content-Type', 'application/json; charset=utf-8');

this.http.post("http://localhost:81/SampleAPI/employee/DeleteEmployee",
this._deletedData, { headers: headers })
    .subscribe((res: Response) => {
        self.loadData();
    });
    }
}
}

```

Code of app.component.employeeelist.html

```

<div class="container">
    <h3>HTTP Module Sample - Add and Fetch Data</h3>
    <div class="panel panel-default" *ngIf="showDetails">
        <div class="panel-body">
            <table class="table table-striped table-bordered">
                <thead>
                    <tr>
                        <th>Sr1 No</th>
                        <th>Alias</th>
                        <th>Employee Name</th>
                        <th>Date of Birth</th>
                        <th>Join Date</th>
                        <th>Department</th>
                        <th>Designation</th>
                        <th>Salary</th>
                        <th></th>
                    </tr>
                </thead>
                <tbody>
                    <tr *ngFor="let item of data">
                        <td>{{item.Id}}</td>
                        <td>{{item.Code}}</td>
                        <td>{{item.Name}}</td>
                        <td>{{item.DOB | date : 'shortDate'}}</td>
                        <td>{{item.DOJ | date : 'mediumDate'}}</td>
                        <td>{{item.Department}}</td>
                        <td>{{item.Designation}}</td>
                        <td>{{item.Salary | currency: 'INR': true}}</td>
                        <td>
                            <a (click)="onUpdateData(item);">

```

```

                <i class="fa fa-edit" style="font-
size:18px"></i>
                </a>
                <a (click)="onDeleteData(item);">
                    <i class="fa fa-remove" style="font-
size:18px"></i>
                </a>
            </td>
        </tr>
    </tbody>
</table>
<p>
    <button class="btn btn-primary" (click)="addEmployee()">
        Add Employee
    </button>
</p>
</div>
</div>
<div class="panel panel-default" *ngIf="showEmployee">
    <employee-add (onHide)="onHide($event);"></employee-add>
</div>
<div class="panel panel-default" *ngIf="editEmployee">
    <employee-update [source]="_selectedData"
(onHide)="onHide($event);"></employee-update>
</div>
</div>

```

Code of app.component.employeeadd.ts

```

import { Component, OnInit, EventEmitter, Output } from '@angular/core';
import { Http, Response, Headers } from '@angular/http';
import 'rxjs/Rx';

@Component({
    moduleId: module.id,
    selector: 'employee-add',
    templateUrl: 'app.component.employeeadd.html'
})

export class AddEmployeeComponent implements OnInit {

    private _model: any = {};
    @Output() private onHide: EventEmitter<boolean> = new
EventEmitter<boolean>();

```

```
constructor(private http: Http) {
}

ngOnInit(): void {

}

private onCancel(): void {
    this._model = {};
    this.onHide.emit(false);
}

private submit(): void {
    debugger;
    if (this.validate()) {
        let self = this;
        let headers = new Headers();
        headers.append('Content-Type', 'application/json; charset=utf-8');

        this.http.post("http://localhost:81/SampleAPI/employee/AddEmployee",
            this._model, { headers: headers })
            .subscribe((res: Response) => {
                self.onCancel();
            });
    }
}

private reset(): void {
    this._model = {};
}

private validate(): boolean {
    let status: boolean = true;
    if (typeof (this._model.code) === "undefined") {
        alert('Alias is Blank');
        status = false;
        return;
    }
    else if (typeof (this._model.name) === "undefined") {
        alert('Name is Blank');
        status = false;
        return;
    }
    else if (typeof (this._model.dob) === "undefined") {
```

```

        alert('dob is Blank');
        status = false;
        return;
    }
    else if (typeof (this._model.doj) === "undefined") {
        alert('DOJ is Blank');
        status = false;
        return;
    }
    else if (typeof (this._model.department) === "undefined") {
        alert('Department is Blank');
        status = false;
        return;
    }
    else if (typeof (this._model.designation) === "undefined") {
        alert('Designation is Blank');
        status = false;
        return;
    }
    else if (typeof (this._model.salary) === "undefined") {
        alert('Salary is Blank');
        status = false;
        return;
    }
    return status;
}
}

```

Code of app.component.employeeadd.html

```

<div class="panel panel-default">
    <h4>Provide Employee Details</h4>
    <table class="table" style="width:60%;">
        <tr>
            <td>Employee Code</td>
            <td><input type="text" [(ngModel)]="_model.code" /></td>
        </tr>
        <tr>
            <td>Employee Name</td>
            <td><input type="text" [(ngModel)]="_model.name" /></td>
        </tr>
        <tr>
            <td>Date of Birth</td>
            <td><input type="date" [(ngModel)]="_model.dob" /></td>
        </tr>
    </table>
</div>

```

```
<tr>  
    <td>Date of Join</td>  
    <td><input type="date" [(ngModel)]="_model.doj" /></td>  
</tr>  
<tr>  
    <td>Department</td>  
    <td><input type="text" [(ngModel)]="_model.department" /></td>  
</tr>  
<tr>  
    <td>Designation</td>  
    <td><input type="text" [(ngModel)]="_model.designation" /></td>  
</tr>  
<tr>  
    <td>Salary</td>  
    <td><input type="number" [(ngModel)]="_model.salary" /></td>  
</tr>  
<tr>  
    <td></td>  
    <td>  
        <input type="button" value="Submit" (click)="submit()" />  
        &nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&~  
        <input type="button" value="Cancel" (click)="onCancel()" />  
    </td>  
</tr>  
</table>  
</div>
```

Output 1

HTTP Module Sample - Add and Fetch Data

| Srl No | Alias | Employee Name | Date of Birth | Join Date | Department | Designation | Salary |
|--------|-------|---------------|---------------|-------------|------------|-------------|------------|
| 1. | A001 | RABIN | 6/10/1980 | Sep 1, 2006 | ACCOUNTS | CLERK | ₹15,000.00 |
| 2. | A002 | SUJIT | 12/22/1986 | Oct 4, 2010 | SALES | MANAGER | ₹35,000.00 |
| 3 | A003 | KAMALESH | 6/3/1982 | May 7, 2006 | ACCOUNTS | CLERK | ₹16,000.00 |

Add Employee

Output 2

HTTP Module Sample - Add and Fetch Data

Provide Employee Details

| | |
|---|---|
| Employee Code | <input type="text" value="B001"/> |
| Employee Name | <input type="text" value="LALIT SHARMA"/> |
| Date of Birth | <input type="text" value="15-02-1990"/> |
| Date of Join | <input type="text" value="17-11-2017"/> |
| Department | <input type="text" value="MARKETING"/> |
| Designation | <input type="text" value="MANAGER"/> |
| Salary | <input type="text" value="40000"/> |
| <input type="button" value="Submit"/> <input type="button" value="Cancel"/> | |

Output 3

HTTP Module Sample - Add and Fetch Data

| Srl No | Alias | Employee Name | Date of Birth | Join Date | Department | Designation | Salary |
|--------|-------|---------------|---------------|--------------|------------|-------------|------------|
| 1 | A001 | RABIN | 6/10/1980 | Sep 1, 2006 | ACCOUNTS | CLERK | ₹15,000.00 |
| 2 | A002 | SUIT | 12/22/1986 | Oct 4, 2010 | SALES | MANAGER | ₹35,000.00 |
| 3 | A003 | KAMALESH | 6/3/1982 | May 7, 2006 | ACCOUNTS | CLERK | ₹16,000.00 |
| 4 | B001 | LALIT SHARMA | 2/15/1999 | Nov 17, 2017 | MARKETING | MANAGER | ₹40,000.00 |

CHAPTER 10 : ROUTING

Angular 4.0 brings many improved modules to the Angular framework including a new router called the Component Router. The component router is totally configurable and feature packed router. Features included are standard view routing, nested child routes, named routes and route parameters.

WHY ROUTING ?

Routing allows us to specify some aspects of the application's state in the URL. Unlike with server-side front-end solutions, this is optional - we can build the full application without ever changing the URL. Adding routing, however, allows the user to go straight into certain aspects of the application. This is very convenient as it can keep your application linkable and bookmarkable and allow users to share links with others.

Routing allows you to:

- Maintain the state of the application
- Implement modular applications
- Implement the application based on the roles (certain roles have access to certain URLs)

ROUTE DEFINITION OBJECTS

The Routes type is an array of routes that defines the routing for the application. This is where we can set up the expected paths, the components we want to use and what we want our application to understand them as.

Each route can have different attributes; some of the common attributes are:

- path - URL to be shown in the browser when application is on the specific route
- component - component to be rendered when the application is on the specific route
- redirectTo - redirect route if needed; each route can have either component or redirect attribute defined in the route (covered later in this chapter)
- pathMatch - optional property that defaults to 'prefix'; determines whether to match full URLs or just the beginning. When defining a route with empty path string set pathMatch to 'full', otherwise it will match all paths.
- children - array of route definitions objects representing the child routes of this route (covered later in this chapter)

To use Routes, create an array of [route configurations](#).

```
const routes: Routes = [  
  { path: 'component-one', component: ComponentOne },  
  { path: 'component-two', component: ComponentTwo }  
];
```

ROUTERMODULE

RouterModule.forRoot takes the Routes array as an argument and returns a *configured* router module. The following sample shows how we import this module in an app.routes.ts file.

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

const routes: Routes = [
  { path: 'component-one', component: ComponentOne },
  { path: 'component-two', component: ComponentTwo }
];

export const routing = RouterModule.forRoot(routes);
```

We then import our routing configuration in the root of our application.

```
import { routing } from './app.routes';

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

REDIRECTING THE ROUTER TO ANOTHER ROUTE

When your application starts, it navigates to the empty route by default. We can configure the router to redirect to a named route by default:

```
export const routes: Routes = [  
  { path: '', redirectTo: 'component-one', pathMatch: 'full' },  
  { path: 'component-one', component: ComponentOne },  
  { path: 'component-two', component: ComponentTwo }  
];
```

The `pathMatch` property, which is required for redirects, tells the router how it should match the URL provided in order to redirect to the specified route. Since `pathMatch: full` is provided, the router will redirect to `component-one` if the entire URL matches the empty path (`''`).

When starting the application, it will now automatically navigate to the route for `component-one`.

ROUTERLINK

Add links to routes using the `RouterLink` directive. For example the following code defines a link to the route at path `component-one`.

```
<a routerLink="/component-one">Component One</a>
```

Alternatively, you can navigate to a route by calling the `navigate` function on the router:

```
this.router.navigate(['component-one']);
```

DYNAMICALLY ADDING ROUTE COMPONENTS

Rather than define each route's component separately, use `RouterOutlet` which serves as a component placeholder; Angular dynamically adds the component for the route being activated into the `<router-outlet></router-outlet>` element.

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

In the above example, the component corresponding to the route specified will be placed after the `<router-outlet></router-outlet>` element when the link is clicked.

WHAT IS NESTED CHILD ROUTES?

Child/Nested routing is a powerful new feature in the new Angular router. We can think of our application as a tree structure, components nested in more components. We can think the same way with our routes and URLs.

So we have the following routes, `/` and `/about`. Maybe our about page is extensive and there are a couple of different views we would like to display as well. The URLs would look something like `/about` and `/about/item`. The first route would be the default about page but the more route would offer another view with more details.

DEFINING CHILD ROUTES

When some routes may only be accessible and viewed within other routes it may be appropriate to create them as child routes.

For example: The product details page may have a tabbed navigation section that shows the product overview by default. When the user clicks the "Technical Specs" tab the section shows the specs instead.

If the user clicks on the product with ID 3, we want to show the product details page with the overview:

```
localhost:3000/product-details/3/overview
```

When the user clicks "Technical Specs":

```
localhost:3000/product-details/3/specs
```

overview and specs are child routes of product-details/:id. They are only reachable within product details.

PASSING OPTIONAL PARAMETERS

Query parameters allow you to pass optional parameters to a route such as pagination information.

For example, on a route with a paginated list, the URL might look like the following to indicate that we've loaded the second page:

```
localhost:3000/product-list?page=2
```

The key difference between query parameters and route parameters is that route parameters are essential to determining route, whereas query parameters are optional

PASSING QUERY PARAMETERS

Use the [queryParams] directive along with [routerLink] to pass query parameters. For example:

```
<a [routerLink]="['product-list']" [queryParams]='{ page: 99 }">Go to Page 99</a>
```

Alternatively, we can navigate programmatically using the Router service:

```
goToPage(pageNum) {  
  this.router.navigate(['/product-list'], { queryParams: { page: pageNum } });  
}
```

READING QUERY PARAMETERS

Similar to reading route parameters, the Router service returns an Observable we can subscribe to to read the query parameters:

```
import { Component } from '@angular/core';  
import { ActivatedRoute, Router } from '@angular/router';
```

```

@Component({
  selector: 'product-list',
  template: `<!-- Show product list -->`
})
export default class ProductList {
  constructor(
    private route: ActivatedRoute,
    private router: Router) {}

  ngOnInit() {
    this.sub = this.route
      .queryParams
      .subscribe(params => {
        // Defaults to 0 if no query param provided.
        this.page = +params['page'] || 0;
      });
  }

  ngOnDestroy() {
    this.sub.unsubscribe();
  }

  nextPage() {
    this.router.navigate(['product-list'], { queryParams: { page: this.page +
1 } });
  }
}

```

Code of app.staticroute.module.ts

```

import { NgModule, NO_ERRORS_SCHEMA } from '@angular/core';
import { APP_BASE_HREF } from '@angular/common';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule } from "@angular/forms";

import { routing, appRoutingProviders } from
'./day4/route/route_config/static/app.routes';

```

```
import { HomeComponent } from
'./day4/route/route_config/static/app.component.homepage';
import { HomeComponent } from
'./day4/route/route_config/static/app.component.home';
import { FirstProgComponent } from './day1/component/app.component.firstprog';
import { HelloWorldComponent } from
'./day1/component/app.component.helloworld';
import { TemplateUrlStyleComponent } from
'./day1/component/app.component.templatestyle';

@NgModule({
  imports: [BrowserModule, FormsModule, routing],
  declarations: [HomePageComponent, HomeComponent, HelloWorldComponent,
FirstProgComponent, TemplateUrlStyleComponent],
  bootstrap: [HomePageComponent],
  schemas: [NO_ERRORS_SCHEMA],
  providers: [{ provide: APP_BASE_HREF, useValue: '/' }],
  appRoutingProviders],
})

export class StaticRouteModule { }
```

Code of app.staticroute.module.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <!--<base href="/"-->
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
  <meta charset="utf-8">
  <title>Angular 2 - Console</title>
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta name="description" content="">
  <meta name="keywords" content="">
  <meta name="author" content="">

  <link href="resource/css/bootstrap.min.css" rel="stylesheet">
  <link rel="stylesheet" href="resource/css/font-awesome.min.css">
  <link rel="stylesheet" href="resource/css/jquery-ui.css">
  <link href="resource/css/style.css" rel="stylesheet">
</head>
<body>
```

```

<home-page></home-page>
<footer>
  <div class="container">
    <div class="row">
      <div class="col-md-12">
        <p class="copy">Copyright &copy; 2017-2018 | <a
href="http://www.c-sharpcorner.com/members/debasis-saha">Debasis Saha</a> </p>
      </div>
    </div>
  </div>
</footer>
<script src="resource/js/jquery.js"></script>
<script src="resource/js/bootstrap.min.js"></script>
<script src="resource/js/jquery-ui.min.js"></script>
<script src="resource/js/jquery.slimscroll.min.js"></script>
<script src="resource/js/custom.js"></script>

<script src="node_modules/core-js/client/shim.min.js"></script>
<script src="node_modules/zone.js/dist/zone.js"></script>
<script src="node_modules/systemjs/dist/system.src.js"></script>
<script src="systemjs.config.js"></script>
<script>
  System.import('main.js').catch(function (err) { console.error(err);
});
</script>

<!-- Set the base href, demo only! In your app: <base href="/"> -->
<script>document.write('<base href="' + document.location + '"
/>');</script>
</body>
</html>

```

Code of app.component.home.ts

```

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

@Component({
  moduleId: module.id,
  selector: 'home',
  template: 'Angular Samples Home Page'
})

export class HomeComponent implements OnInit {
  constructor() {

```

```
    }

    ngOnInit(): void {
    }

}
```

Code of app.component.homepage.ts

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

@Component({
  moduleId: module.id,
  selector: 'home-page',
  templateUrl: 'app.component.homepage.html'
})

export class HomePageComponent implements OnInit {

  constructor() {
  }

  ngOnInit(): void {
  }

}
```

Code of app.routes.ts

```
import { Routes, RouterModule } from '@angular/router';
import { HomeComponent } from './app.component.home';
import { FirstProgComponent } from
'../../../../day1/component/app.component.firstprog';
import { HelloWorldComponent } from
'../../../../day1/component/app.component.helloworld';
import { TemplateUrlStyleComponent } from
'../../../../day1/component/app.component.templatestyle';

export const routes: Routes = [
  { path: '', redirectTo: 'home', pathMatch: 'full' },
  { path: 'home', component: HomeComponent },
  { path: 'hello', component: HelloWorldComponent },
  { path: 'firstprog', component: FirstProgComponent },
  { path: 'template', component: TemplateUrlStyleComponent }
```



```
];  
  
export const appRoutingProviders: any[] = [];  
  
export const routing = RouterModule.forRoot(routes);
```

Code of app.component.firstprog.ts

```
import { Component } from "@angular/core";  
  
@Component({  
  selector: "first-prog",  
  template: "<h1>First Programme in Angular 4.0. Welcome to Day 1  
Session</h1> "  
})  
  
export class FirstProgComponent {  
  constructor() {  
  }  
}
```

Code of app.component.helloworld.ts

```
import { Component } from '@angular/core';  
  
@Component({  
  selector: 'hello-world',  
  template: '<h1><b>Angular 4!</b>Hello World</h1>'  
})  
  
export class HelloWorldComponent {  
  constructor() {  
  }  
  
  ngOnInit() {  
    alert("Page Init Method Fired!!")  
  }  
}
```

Code of app.component.templatestyle.ts

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

@Component({
  moduleId: module.id,
  selector: 'template-style',
  templateUrl: 'app.component.template.html',
  styles: ['h1{color:red;font-weight:bold}','h2{color:blue}']
})

export class TemplateUrlStyleComponent {
  constructor() {

  }
}
```

Code of app.component.template.html

```
<div>
  <h1>Angular 4 Component with Template</h1>
  <br />
  <h2>C# Corner</h2>
</div>
```

CHAPTER 11 : UNIT TESTING

In this Angular 4.0 series, we have already discussed about different types of basic and advanced concept or features of AngularJS 4.0 like data binding, directives, pipes, service, route, http modules, pipes etc. Now in this article, we will discuss about one of the main advantage of Angular 4.0 i.e. Unit Testing. Because Angular Framework has been designed with testability as a primary objective.

Nowadays, JavaScript has become the de facto programming language to build and empower frontend/web applications. We can use JavaScript to develop simple or complex applications. However, applications in production are often vulnerable to bugs caused by design inconsistencies, logical implementation errors, and similar issues. For this reason, it is usually difficult to predict how applications will behave in real-time environments, which leads to unexpected behavior, non-availability of applications, or outages for short or long durations. This generates lack of confidence and dissatisfaction among application users. Also, high cost is often associated with fixing the production bugs. Therefore, there is a need to develop applications that are of a high quality and that offer high availability.

Test-Driven-Development is an engineering process in which the developer writes an initial automated test case that defines a feature, then writes the minimum amount of code to pass the test and eventually refactors the code to acceptable standards.

Now when we are talked about testing in Angular, we are basically point out two different types of Testing.

UNIT TESTING

A unit test is used to test individual components of the system. That's why unit testing is basically called **Isolated Testing**. It is the best practice of testing small isolated pieces of code. If our unit testing depends on some external resources like database, network, apis then it is not a unit test.

An integration test is a test which tests the system as a whole, and how it will run in production. Unit tests should only verify the behavior of a specific unit of code. If the unit's behavior is modified, then the unit test would be updated as well. Unit tests should not make assumptions about the behavior of other parts of your codebase or your dependencies. When other parts of your codebase are modified, your unit tests should not fail. (Any failure indicates a test that relies on other components and is therefore not a unit test.) Unit tests are cheap to maintain and should only be updated when the individual units are modified. For TDD in Angular, a unit is most commonly defined as a class, pipe, component, or service. It is important to keep units relatively small. This helps you write small tests which are "self-documenting", where they are easy to read and understand.

FUNCTIONAL TESTING

Functional testing is defined as the testing the complete functionality or functional flow of an application. In case of web applications, this means interacting the different UI of the web applications as it is running by the user in the browser in real life. This is also called **End To End Testing (E2E Testing)**.

THE TESTING TOOL CHAIN

Our testing toolchain will consist of the following tools:

- Jasmine
- Karma
- Phantom-js
- Istanbul
- Sinon
- Chai

In this article, we use Jasmine and Karma for perform unit test in Angular 4.

JASMINE

Jasmine is the most popular JavaScript testing framework in the Angular community. This testing framework supports a software development practice which is called as **Behavior Driven Development** or **BDD**. This is one the main features of **Test Driven Development(TDD)**. This is the core framework that we will write our unit tests with. Basically Jasmine & BDD basically try to describes a test method case in a human readable pattern so that any user include any non-technical personal can identified what is going on. Jasmine tests are written using JavaScript functions, which makes writing tests a nice extension of writing application code. There are several Jasmine functions in the example, which I have described below.

| Name | Descriptions |
|------------|---|
| describe | Groups a number of related tests (this is optional, but it helps organize test code). Basically a test suits need to called this function with two parameter – a string (for test name or test description) and a function. |
| beforeEach | Executes a function before each test (this is often used for the arrange part of a test) |
| it | Executes a function to form a test (the act part of the test) |
| expect | Identifies the result from the test (part of the assert stage) |
| toEqual | Compares the result from the test to the expected value (the other part of the assert) |
| beforeAll | This function is called once , <i>before</i> all the specs in describe test suite are run |

The basic sequence to pay attention to is that the it function executes a test function so that the expect and toEqual functions can be used to assess the result. The toEqual function is only one way that Jasmine can evaluate the result of a test. I have listed the other available functions as below.

| Name | Descriptions |
|---|--|
| <code>expect(x).toEqual(val)</code> | Asserts that x has the same value as val (but not necessarily the same object) |
| <code>expect(x).toBe(obj)</code> | Asserts that x and obj are the same object |
| <code>expect(x).toMatch(regexp)</code> | Asserts that x matches the specified regular expression |
| <code>expect(x).toBeDefined()</code> | Asserts that x has been defined |
| <code>expect(x).toBeUndefined()</code> | Asserts that x has not been defined |
| <code>expect(x).toBeNull()</code> | Asserts that x is null |
| <code>expect(x).toBeTruthy()</code> | Asserts that x is true or evaluates to true |
| <code>expect(x).toBeFalsy()</code> | Asserts that x is false or evaluates to false |
| <code>expect(x).toContain(y)</code> | Asserts that x is a string that contains y |
| <code>expect(x).toBeGreaterThan(y)</code> | Asserts that x is greater than y |
| <code>expect(fn).toThrow(string);</code> | Asserts to capture any throw of the given function |
| <code>expect(fn).toThrowError(string);</code> | Asserts to capture any exception error |

KARMA

Karma is a test automation tool for controlling the execution of our tests and what browser to perform them under. It also allows us to generate various reports on the results. For one or two tests this may seem like overkill, but as an application grows larger and the number of units to test grows, it is important to organize, execute and report on tests in an efficient manner. Karma is library agnostic so we could use other testing frameworks in combination with other tools (like code coverage reports, spy testing, e2e, etc.).

In order to test our Angular application we must create an environment for it to run in. We could use a browser like Chrome or Firefox to accomplish this (Karma supports in-browser testing), or we could use a browser-less environment to test our application, which can offer us greater control over automating certain tasks and managing our testing workflow. PhantomJS provides a JavaScript API that allows us to create a headless DOM instance which can be used to bootstrap our Angular application. Then, using that DOM instance that is running our Angular application, we can run our tests.

Karma is basically a tool which lets us spawn browsers and run the all jasmine tests inside of them which are executed from the command line. This result of the tests are also displayed in the command line. Karma also watch our development file changes and re-run the test automatically.

Istanbul is used by Karma to generate code coverage reports, which tells us the overall percentage of our application being tested. This is a great way to track which components/services/pipes/etc. have tests written and which don't. We can get some useful insight into how much of the application is being tested and where.

WHY UNIT TESTING REQUIRED ?

- Basically Guards or protects the existing code which can be broken due to any changes.

- Integrate automatic build process to automate the pipeline of resource publish at any time.
- Clarifies what the code does both when used as intended and when faced with deviant conditions. They serve as a form of documentation for your code.
- Reveals mistakes in design and implementation. When a part of the application seems hard to test, the root cause is often a design flaw, something to cure now rather than later when it becomes expensive to fix.
- It allows us to test the interaction of a directives or components with its' template url.
- It allows us to easily track the change detection
- It also allows us to use and test Angular Dependency Injection framework.

FILENAME CONVENTIONS

Each unit test is put into its own separate file. The Angular team recommends putting unit test scripts alongside the files they are testing and using a .spec filename extension to mark it as a testing script (this is a Jasmine convention). So if you had a component `/app/components/mycomponent.ts`, then your unit test for this component would be in `/app/components/mycomponent.spec.ts`. This is a matter of personal preference; you can put your testing scripts wherever you like, though keeping them close to your source files makes them easier to find and gives those who aren't familiar with the source code an idea of how that particular piece of code should work.

ANGULAR TEST BED

Angular Test Bed (ATB) is one of the higher level Angular Only testing framework which allows us to pass data to test environment and can easily test behaviors that depends on Angular Framework.

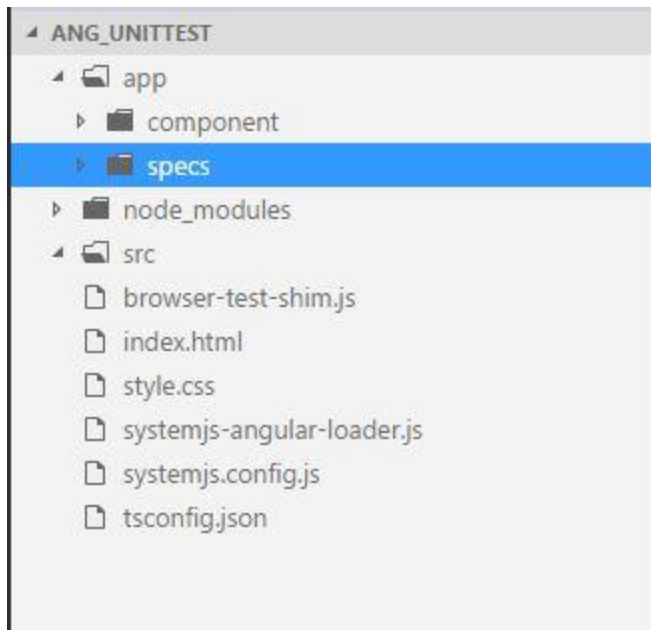
WHEN TO USE ANGULAR TEST BED

Actually, we need to use the Angular Test Bed process for perform unit testing as per the below reasons ,

1. This process allows us to test the interaction of a component or directives with its templates
2. It also allows us to easily test change detection mechanism of the angular
3. It also allows us to test the Dependency Injection process
4. It allows us to run test using NgModule configuration which we use in our application
5. It allows us to test user interactions including clicks and input field operation.

OUR FIRST UNIT TEST

For starting the unit test, first, we need to configure the environment for the unit Test. For configuring the Unit Test Project, we need to create a folder structure just as below.



- **App** Folder contains both Angular and Unit Test Specs
- **Component** folder contains all the components which we need to unit test
- **Specs** folder contains all the Angular test bed unit test models

Code of tsconfig.json

```
{
  "compilerOptions": {
    "target": "es5",
    "module": "commonjs",
    "moduleResolution": "node",
    "sourceMap": true,
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "lib": [
      "es2015",
      "dom"
    ],
    "noImplicitAny": true,
    "suppressImplicitAnyIndexErrors": true,
    "typeRoots": [
      "../node_modules/@types/"
    ]
  },
  "compileOnSave": true,
  "exclude": [
    "node_modules/*",
    "**/*-aot.ts"
  ]
}
```

```
]
}
```

Code of systemjs.config.js file

```
(function (global) {
  System.config({
    // DEMO ONLY! REAL CODE SHOULD NOT TRANSPILE IN THE BROWSER
    transpiler: 'ts',
    typescriptOptions: {
      // Copy of compiler options in standard tsconfig.json
      "target": "es5",
      "module": "commonjs",
      "moduleResolution": "node",
      "sourceMap": true,
      "emitDecoratorMetadata": true,
      "experimentalDecorators": true,
      "lib": ["es2015", "dom"],
      "noImplicitAny": true,
      "suppressImplicitAnyIndexErrors": true
    },
    meta: {
      'typescript': {
        "exports": "ts"
      }
    },
    paths: {
      // paths serve as alias
      'npm:': 'https://unpkg.com/'
    },
    // map tells the System loader where to look for things
    map: {
      // our app is within the app folder
      'app': 'app',

      // angular bundles
      '@angular/animations':
'npm:@angular/animations/bundles/animations.umd.js',
      '@angular/animations/browser':
'npm:@angular/animations/bundles/animations-browser.umd.js',
      '@angular/core': 'npm:@angular/core/bundles/core.umd.js',
      '@angular/common': 'npm:@angular/common/bundles/common.umd.js',
      '@angular/common/http': 'npm:@angular/common/bundles/common-
http.umd.js',
      '@angular/compiler': 'npm:@angular/compiler/bundles/compiler.umd.js',
```



```

        '@angular/platform-browser': 'npm:@angular/platform-
browser/bundles/platform-browser.umd.js',
        '@angular/platform-browser/animations': 'npm:@angular/platform-
browser/bundles/platform-browser-animations.umd.js',
        '@angular/platform-browser-dynamic': 'npm:@angular/platform-browser-
dynamic/bundles/platform-browser-dynamic.umd.js',
        '@angular/http': 'npm:@angular/http/bundles/http.umd.js',
        '@angular/router': 'npm:@angular/router/bundles/router.umd.js',
        '@angular/router/upgrade': 'npm:@angular/router/bundles/router-
upgrade.umd.js',
        '@angular/forms': 'npm:@angular/forms/bundles/forms.umd.js',
        '@angular/upgrade': 'npm:@angular/upgrade/bundles/upgrade.umd.js',
        '@angular/upgrade/static': 'npm:@angular/upgrade/bundles/upgrade-
static.umd.js',

        // other libraries
        'rxjs': 'npm:rxjs@5.5.3',
        'rxjs/operators': 'npm:rxjs@5.5.3/operators/index.js',
        'tslib': 'npm:tslib/tslib.js',
        'angular-in-memory-web-api': 'npm:angular-in-memory-web-
api@0.4/bundles/in-memory-web-api.umd.js',
        'ts': 'npm:plugin-
typescript@5.2.7/lib/plugin.js',
        'typescript': 'npm:typescript@2.4.2/lib/typescript.js',

    },
    // packages tells the System loader how to load when no filename and/or
no extension
    packages: {
        app: {
            main: './main.ts',
            defaultExtension: 'ts',
            meta: {
                './*.ts': {
                    loader: 'systemjs-angular-loader.js'
                }
            }
        },
        rxjs: {
            defaultExtension: 'js'
        }
    }
});

})(this);

```

Code of systemjs-AngularJS-loader.js

```

var templateUrlRegex = /templateUrl\s*:(\s*['"](.*)['"]\s*)/gm;
var stylesRegex = /styleUrls\s*:(\s*\[[^\]]*\])/g;
var stringRegex = /(['"])(?:[^\\"\\]|\\.)*\1/g;

module.exports.translate = function(load){
    if (load.source.indexOf('moduleId') !== -1) return load;

    var url = document.createElement('a');
    url.href = load.address;

    var basePathParts = url.pathname.split('/');

    basePathParts.pop();
    var basePath = basePathParts.join('/');

    var baseHref = document.createElement('a');
    baseHref.href = this.baseURL;
    baseHref = baseHref.pathname;

    if (!baseHref.startsWith('/base/')) { // it is not karma
        basePath = basePath.replace(baseHref, '');
    }

    load.source = load.source
        .replace(templateUrlRegex, function(match, quote, url){
            var resolvedUrl = url;

            if (url.startsWith('.')) {
                resolvedUrl = basePath + url.substr(1);
            }

            return 'templateUrl: \'' + resolvedUrl + '\'';
        })
        .replace(stylesRegex, function(match, relativeUrls) {
            var urls = [];

            while ((match = stringRegex.exec(relativeUrls)) !== null) {
                if (match[2].startsWith('.')) {
                    urls.push('\' + basePath + match[2].substr(1) + '\'');
                } else {
                    urls.push('\' + match[2] + '\'');
                }
            }
        })
    }

```

```
        return "styleUrls: [" + urls.join(', ') + "];";
    });

    return load;
};
```

Code of browser-test-shim.js

```
(function () {

    Error.stackTraceLimit = 0; // "No stacktrace" is usually best for app
    testing.

    // Uncomment to get full stacktrace output. Sometimes helpful, usually not.
    // Error.stackTraceLimit = Infinity; //

    jasmine.DEFAULT_TIMEOUT_INTERVAL = 3000;

    var baseUrl = document.baseURI;
    baseUrl = baseUrl + baseUrl[baseUrl.length-1] ? '' : '/';

    System.config({
        baseUrl: baseUrl,
        // Extend usual application package list with test folder
        packages: { 'testing': { main: 'index.js', defaultExtension: 'js' } },

        // Assume npm: is set in `paths` in systemjs.config
        // Map the angular testing umd bundles
        map: {
            '@angular/core/testing': 'npm:@angular/core/bundles/core-
testing.umd.js',
            '@angular/common/testing': 'npm:@angular/common/bundles/common-
testing.umd.js',
            '@angular/compiler/testing': 'npm:@angular/compiler/bundles/compiler-
testing.umd.js',
            '@angular/platform-browser/testing': 'npm:@angular/platform-
browser/bundles/platform-browser-testing.umd.js',
            '@angular/platform-browser-dynamic/testing': 'npm:@angular/platform-
browser-dynamic/bundles/platform-browser-dynamic-testing.umd.js',
            '@angular/http/testing': 'npm:@angular/http/bundles/http-
testing.umd.js',
            '@angular/router/testing': 'npm:@angular/router/bundles/router-
testing.umd.js',
```

```
'@angular/forms/testing': 'npm:@angular/forms/bundles/forms-  
testing.umd.js',  
  },  
});  
  
System.import('systemjs.config.js')  
  .then(importSystemJsExtras)  
  .then(initTestBed)  
  .then(initTesting);  
  
/** Optional SystemJS configuration extras. Keep going w/o it */  
function importSystemJsExtras(){  
  return System.import('systemjs.config.extras.js')  
    .catch(function(reason) {  
      console.log(  
        'Warning: System.import could not load the optional  
"systemjs.config.extras.js". Did you omit it by accident? Continuing without  
it.'  
      );  
      console.log(reason);  
    });  
}  
  
function initTestBed(){  
  return Promise.all([  
    System.import('@angular/core/testing'),  
    System.import('@angular/platform-browser-dynamic/testing')  
  ])  
  
  .then(function (providers) {  
    var coreTesting = providers[0];  
    var browserTesting = providers[1];  
  
    coreTesting.TestBed.initTestEnvironment(  
      browserTesting.BrowserDynamicTestingModule,  
      browserTesting.platformBrowserDynamicTesting());  
  })  
}  
  
// Import all spec files defined in the html (__spec_files__)  
// and start Jasmine testrunner  
function initTesting () {  
  console.log('loading spec files: ' + __spec_files__.join(', '));  
  return Promise.all(  
    __spec_files__.map(function(spec) {
```

```

        return System.import(spec);
    })
)
// After all imports load, re-execute `window.onload` which
// triggers the Jasmine test-runner start or explain what went wrong
.then(success, console.error.bind(console));

function success () {
    console.log('Spec files loaded; starting Jasmine testrunner');
    window.onload();
}
}

})();

```

Code of Index.html file

```

<!-- Run application specs in a browser -->
<!DOCTYPE html>
<html>
<head>
    <script>document.write('<base href="' + document.location + '" />');</script>
    <title>Sample App Specs</title>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link rel="stylesheet" href="./style.css">
    <link rel="stylesheet"
href="https://cdnjs.cloudflare.com/ajax/libs/jasmine/2.4.1/jasmine.css">

</head>
<body>
    <script src="https://unpkg.com/systemjs@0.19.39/dist/system.src.js"></script>

    <script
src="https://cdnjs.cloudflare.com/ajax/libs/jasmine/2.4.1/jasmine.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/jasmine/2.4.1/jasmine-
html.js"></script>
    <script
src="https://cdnjs.cloudflare.com/ajax/libs/jasmine/2.4.1/boot.js"></script>

    <script src="https://unpkg.com/reflect-metadata@0.1.8"></script>

    <script src="https://unpkg.com/zone.js@0.7.4?main=browser"></script>
    <script src="https://unpkg.com/zone.js/dist/long-stack-trace-
zone.js?main=browser"></script>

```

```

<script src="https://unpkg.com/zone.js/dist/proxy.js?main=browser"></script>
<script src="https://unpkg.com/zone.js/dist/sync-
test.js?main=browser"></script>
<script src="https://unpkg.com/zone.js/dist/jasmine-
patch.js?main=browser"></script>
<!-- <script src="https://unpkg.com/zone.js/dist/async-
test.js?main=browser"></script>
<script src="https://unpkg.com/zone.js/dist/fake-async-
test.js?main=browser"></script> -->

<script>
  var __spec_files__ = [
    'app/specs/calculator.spec'
  ];
</script>
<script src="browser-test-shim.js"></script>
</body>

</html>

```

In the above index.html file, `__spec_files` variables are basically used to store the list of Angular unit test spec file details with a proper folder so that when we run this UI in the browser, it loads spec files one by one and executes them with the help of `browser-test-shims.js` file. In that file, `initTesting()` is basically initializing the Unit Test methods one by one.

Now, perform the unit test. We first need to add a simple TypeScript class which basically performs the basic mathematical operations between two numbers.

Code of calculator.woinput.ts

```

export class CalculatorWithoutInput {
  private _firstNumber:number=10;
  private _secondNumber:number=20;
  private _result : number = 0;

  constructor(){}

  public addNumbers():number{
    this._result = this._firstNumber + this._secondNumber;
    return this._result;
  }

  public subtractNumbers():number{
    this._result = this._firstNumber - this._secondNumber;
    return this._result;
  }
}

```

```

    public multiplyNumbers():number{
        this._result = this._firstNumber * this._secondNumber;
        return this._result;
    }
}

```

Code of calculator.woinput.spec.ts

```

import { CalculatorWithoutInput } from '../component/calculator.woinput';

describe('Calcutlor Without Inputs (Basic Class)', () => {
    let firstNumber :number = 0;
    let secondNumber :number = 0;
    let result : number = 0;

    let objCaculator : CalculatorWithoutInput;

    beforeEach(() => {
        this.objCaculator = new CalculatorWithoutInput();
    });

    afterEach(() => {
        this.objCaculator=null;
        this.firstNumber=0;
        this.secondNumber=0;
        this.result=0;
    });

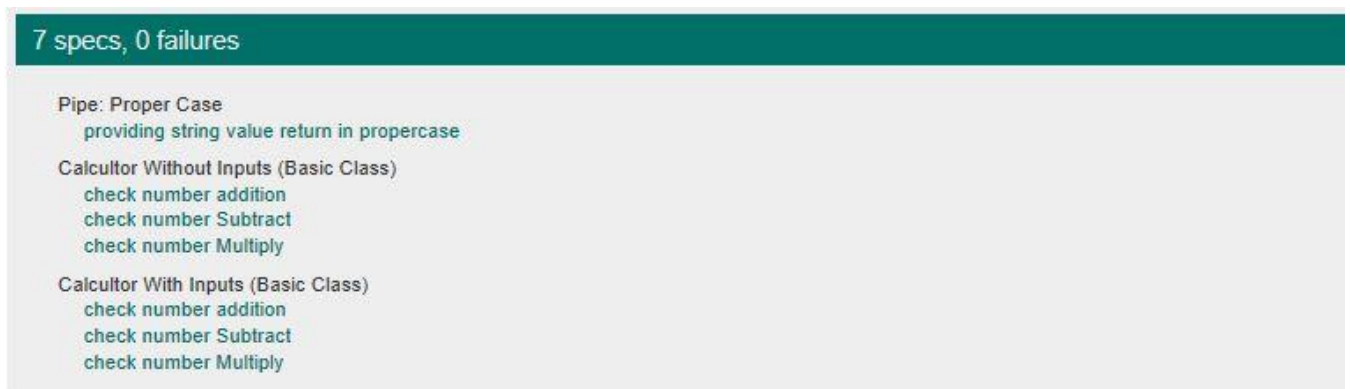
    it('check number addition', () => {
        this.firstNumber=10;
        this.secondNumber=20;
        this.result=this.firstNumber + this.secondNumber;
        expect(this.objCaculator.addNumbers())
            .toEqual(this.result);
    });

    it('check number Subtract', () => {
        this.firstNumber=10;
        this.secondNumber=20;
        this.result=this.firstNumber - this.secondNumber;
        expect(this.objCaculator.subtractNumbers())
            .toEqual(this.result);
    });
});

```

```
it('check number Multiply', () => {
    this.firstNumber=10;
    this.secondNumber=20;
    this.result=this.firstNumber * this.secondNumber;
    expect(this.objCaculator.multiplyNumbers())
        .toEqual(this.result);
});
});
```

Output



TESTING COMPONENTS

Now, for performing unit tests on an Angular component, we first need to develop an Angular component as below.

```
import { Component, Input, Output, EventEmitter } from '@angular/core';

@Component({
    // moduleId: module.id,
    selector: 'student-template-data',
    template: `
        <div class="form-horizontal">
        <h2 class="aligncenter">Student Details (Template)</h2><br />
        <div class="row">
            <div class="col-xs-12 col-sm-2 col-md-2">
                <span>First Name</span>
            </div>
            <div class="col-xs-12 col-sm-4 col-md-4">
                <input type="text" id="txtFName" placeholder="Enter
First Name" #firstName/>
            </div>
            <div class="col-xs-12 col-sm-2 col-md-2">
                <span>Last Name</span>
            </div>
        </div>
        </div>`
})
```



```

        </div>
        <div class="col-xs-12 col-sm-4 col-md-4">
            <input type="text" id="txtLName" placeholder="Enter
Second Name" #lastName/>
        </div>
    </div>
    <br />
    <div class="row">

        <div class="col-xs-12 col-sm-2 col-md-2">
            <span>Email</span>
        </div>
        <div class="col-xs-12 col-sm-4 col-md-4">
            <input type="email" id="txtEmail" #email/>
        </div>
        <div class="col-xs-12 col-sm-2 col-md-2">
        </div>
        <div class="col-xs-12 col-sm-4 col-md-4">
            <input type="button" id="btnSubmit" value="Submit"
class="btn btn-primary" [disabled]="!enabled"

(click)="onSubmit(firstName.value,lastName.value,email.value)"/>
        </div>
    </div>
</div>
,
})

export class StudentTemplateComponent {

    private _model: any = {};

    @Input() enabled:boolean = true;
    @Output() onFormSubmit: EventEmitter<any> = new EventEmitter<any>();

    constructor() {
    }

    private onSubmit(firstName:string, lastName:string, email : string): void {
        this._model.firstName = firstName;
        this._model.lastName = lastName;
        this._model.email = email;
        if (typeof (this._model) === "undefined") {
            alert("Form not Filled Up Properly");
        }
    }
}

```

```
        else {
            alert("Data Is Correct");
            this.onFormSubmit.emit(this._model);
        }
    }

    private onClear(): void {
        this._model = {};
    }
}
```

So, in the above component, we have used the following features.

- Input Properties of a Component
- Output Properties of a Component for Button Event Emit
- Also, we used inline HTML template using Template Selector in the @Component Decorator.

Testing @Input() Decorator

In fact, in Angular 4, every input property is just like a simple variable object value which can be set from outside of the component using selector or using component instance as an object. So normally, we can set the value of input properties as below in our unit test specs.

Testing @Output() Decorator

Testing an Output event is not as simple as an Input method. Because output event is actually an observable object so that we can subscribe to it and get a callback for every event. So for the Output event, we need to raise the event of the specified control in the View. Also, in the output event, we may need to assign values in the input controls of the forms.

Unit Test code of the above mentioned components

```
import { TestBed, ComponentFixture, inject, async } from '@angular/core/testing';
import { StudentTemplateComponent } from '../component/app.component.template';
import { Component, DebugElement } from "@angular/core";
import { By } from "@angular/platform-browser";
```

```
describe('Component: Student Form ', () => {

    let component: StudentTemplateComponent;
    let fixture: ComponentFixture<StudentTemplateComponent>;
    let submitEl: DebugElement;
    let firstNameEl: DebugElement;
    let lastNameEl: DebugElement;
    let emailEl : DebugElement;
```

```
beforeEach(async(() => {
    TestBed.configureTestingModule({
        declarations: [ StudentTemplateComponent ]
    });

    fixture = TestBed.createComponent(StudentTemplateComponent);
    component = fixture.componentInstance;

    submitEl = fixture.debugElement.query(By.css('input[id=btnSubmit]'));
    firstNameEl = fixture.debugElement.query(By.css('input[id=txtFName]'));
    lastNameEl = fixture.debugElement.query(By.css('input[id=txtLName]'));
    emailEl = fixture.debugElement.query(By.css('input[type=email]'));
}));

it('Setting enabled to false disabled the submit button', () => {
    component.enabled = false;
    fixture.detectChanges();
    expect(submitEl.nativeElement.disabled).toBeTruthy();
});

it('Setting enabled to true enables the submit button', () => {
    component.enabled = true;
    fixture.detectChanges();
    expect(submitEl.nativeElement.disabled).toBeFalsy();
});

it('Entering value in input controls and emit output events', () => {
    let user: any;
    firstNameEl.nativeElement.value = "Debasis";
    lastNameEl.nativeElement.value = "Saha";
    emailEl.nativeElement.value = "debasis@yahoo.com";

    // Subscribe to the Observable and store the user in a local variable.
    component.onFormSubmit.subscribe((value) => user = value);

    // This sync emits the event and the subscribe callback gets executed above
    submitEl.triggerEventHandler('click', null);

    // Now we can check to make sure the emitted value is correct
    expect(user.firstName).toBe("Debasis");
    expect(user.lastName).toBe("Saha");
    expect(user.email).toBe("debasis@yahoo.com");
});
});
```

Output

 Jasmine 2.4.1

Options

3 specs, 0 failures

finished in 62.223s

Component: Student Form

- Setting enabled to false disabled the submit button
- Setting enabled to true enables the submit button
- Entering value in input controls and emit output events

Student Details (Template)

First Name

Last Name

Email

TESTING SERVICES

For performing the unit test on Angular service, we will create login authentication service called LoginService and wrap that service into a component called LoginComponent

Code of the app.service.login.ts

```
export class LoginService {
    isAuthenticated(): boolean {
        return !!localStorage.getItem('token');
    }
}
```

Code of the app.component.login.ts

```
import {Component, EventEmitter, Output} from '@angular/core';
import {FormGroup, Validators, FormBuilder} from "@angular/forms";

export class User {
    constructor(public email: string,
                public password: string) {
    }
}

@Component({
    selector: 'login-form',
    template: `
        <form (ngSubmit)="login()"
            [formGroup]="form">
            <label>Email</label>
            <input type="email"
                formControlName="email">
            <label>Password</label>
            <input type="password"
                formControlName="password">
            <button type="submit">Login</button>
        </form>
    `
})
export class LoginComponent {
    @Output() loggedIn = new EventEmitter<User>();
    form: FormGroup;

    constructor(private fb: FormBuilder) {
    }
}
```

```

ngOnInit() {
  this.form = this.fb.group({
    email: ['', [
      Validators.required,
      Validators.pattern("^[^ @]*@[^ @]*$")]],
    password: ['', [
      Validators.required,
      Validators.minLength(8)]],
  });
}

login() {
  console.log(`Login ${this.form.value}`);
  if (this.form.valid) {
    this.loggedIn.emit(
      new User(
        this.form.value.email,
        this.form.value.password
      )
    );
  }
}
}

```

Code of the app.component.login.spec.ts

```

import { TestBed, ComponentFixture } from '@angular/core/testing';
import { ReactiveFormsModule, FormsModule } from "@angular/forms";
import { LoginComponent, User } from "../component/app.component.login";

describe('Component: Login', () => {

  let component: LoginComponent;
  let fixture: ComponentFixture<LoginComponent>;

  beforeEach(() => {

    // refine the test module by declaring the test component
    TestBed.configureTestingModule({
      imports: [ReactiveFormsModule, FormsModule],
      declarations: [LoginComponent]
    });
  });
}

```

```
// create component and test fixture
fixture = TestBed.createComponent(LoginComponent);

// get test component from the fixture
component = fixture.componentInstance;
component.ngOnInit();
});

it('form invalid when empty', () => {
    expect(component.form.valid).toBeFalsy();
});

it('email field validity', () => {
    let errors = {};
    let email = component.form.controls['email'];
    expect(email.valid).toBeFalsy();

    // Email field is required
    errors = email.errors || {};
    expect(errors['required']).toBeTruthy();

    // Set email to something
    email.setValue("test");
    errors = email.errors || {};
    expect(errors['required']).toBeFalsy();
    expect(errors['pattern']).toBeTruthy();

    // Set email to something correct
    email.setValue("test@example.com");
    errors = email.errors || {};
    expect(errors['required']).toBeFalsy();
    expect(errors['pattern']).toBeFalsy();
});

it('password field validity', () => {
    let errors = {};
    let password = component.form.controls['password'];

    // Email field is required
    errors = password.errors || {};
    expect(errors['required']).toBeTruthy();

    // Set email to something
    password.setValue("123456");
    errors = password.errors || {};
```

```
expect(errors['required']).toBeFalsy();
expect(errors['minlength']).toBeTruthy();

// Set email to something correct
password.setValue("123456789");
errors = password.errors || {};
expect(errors['required']).toBeFalsy();
expect(errors['minlength']).toBeFalsy();
});

it('submitting a form emits a user', () => {
  expect(component.form.valid).toBeFalsy();
  component.form.controls['email'].setValue("test@test.com");
  component.form.controls['password'].setValue("123456789");
  expect(component.form.valid).toBeTruthy();

  let user: User;
  // Subscribe to the Observable and store the user in a local variable.
  component.loggedIn.subscribe((value) => user = value);

  // Trigger the login function
  component.login();

  // Now we can check to make sure the emitted value is correct
  expect(user.email).toBe("test@test.com");
  expect(user.password).toBe("123456789");
});
});
```

Output

