**Angular 4** released in March 2017 proves to be a major breakthrough and is the latest release from the Angular team after Angular2.

Angular 4 is almost the same as Angular 2. It has a backward compatibility with Angular 2. Projects developed in Angular 2 will work without any issues with Angular 4.

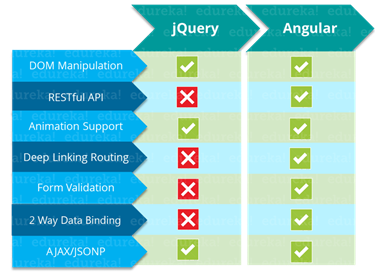
## ****Evolution of Angular****

Angular is a JavaScript based open-source framework for building client-side web applications. So, let us first understand Javascript. JavaScript runs on the client side of the web, which can be used to design or program how the web pages behave on the occurrence of an event. Typically, JavaScript is used for interface interactions, slideshows and other interactive components. JavaScript evolved quickly and has also been used for server-side programming (like in Node.js), game development, etc.

JavaScript deals with the dynamic content, which is an important aspect of web development. Dynamic content refers to constantly changing content and it adapts to specific users. For example, JavaScript can be used to determine whether or not to render the mobile version of the website by checking the device, which is accessing the website.

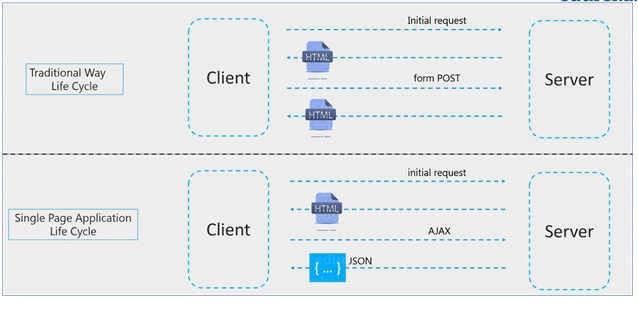
This encouraged web developers to start creating their own custom JavaScript libraries for reducing the number of code lines and implementing complex functionalities easily. **jQuery** is a fast, small, and feature-rich JavaScript library, which makes things like HTML document traversal and manipulation, event handling, animation, and Ajax much simpler with an easy-to-use API. jQuery became the most popular one because it was easy to use and extremely powerful.

Since jQuery has no real structure, the developer has full freedom to build projects as they see fit. However, the lack of structure also means it’s easier to fall into the trap of “spaghetti code,” which can lead to confusion in larger projects with no clear design direction or code maintainability. For these situations, a framework like Angular can be a big help.



Angular is a client-side JavaScript framework that was specifically designed to help developers build SPAs (Single Page Applications) in accordance with best practices for web development. By providing a structured environment for building SPAs, the risk of producing “spaghetti code” is highly reduced. So, you must be wondering what is SPA?

***Single-page application*** (or ***SPA***) are applications that are accessed via a web browser like other websites but offer more dynamic interactions resembling native mobile and desktop apps. The most notable difference between a regular website and SPA is the reduced amount of page refreshes. SPAs have a heavier usage of AJAX- a way to communicate with back-end servers without doing a full page refresh to get data loaded into our application. As a result, the process of rendering pages happens mostly on the client-side.

**Figure:** Angular Tutorial – Single Page Application

For example, if you go through Gmail, you will notice that while opening mail from the inbox will only fetch the email and display it in place of the e-mail list. The rest of the components like sidebar, navigation bar etc. are not reloaded. It only refreshes the DOM (Document Object Model) for the required section. So, this reduces the overhead loading of the website.

So, now as we know what is Javascript & Jquery and how angular came into the picture. Moving ahead in Angular Tutorial, we will look through the features of angular and understand how to work with Angular.

**Differences between Angular and AngularJS**

* The architecture 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. We will be looking at it in a while.
* 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.
* **Modularity** – Angular follows modularity. Similar functionalities are kept together in same modules. This gives Angular a lighter & faster core.

Angular recommends the use of Microsoft’s TypeScript language, which introduces the following features:

* Class-based Object Oriented Programming
* Static Typing

TypeScript is a superset of ECMAScript 6 (ES6) and is backward compatible with ECMAScript 5. Angular also includes the benefits of ES6:

* + Iterators
  + For/Of loops
  + Reflection
  + Improved dependency injection – bindings make it possible for dependencies to be named
  + Dynamic loading
  + Asynchronous template compilation
  + Simpler Routing
  + Replacing controllers and $scope with components and directives – a component is a directive with a template
* Support reactive programming using RxJS

Moving ahead in this Angular tutorial, let’s understand the features of Angular.

**Cross Platform**

* **Progressive web apps**

It uses modern web platform capabilities to deliver an app-like experience. It gives high performance, offline, and zero-step installation. So, working with Angular is pretty much easy.

* **Native**

You can build native mobile apps with strategies using Ionic Framework, NativeScript, and React Native.

* **Desktop**

Create desktop-installed apps across Mac, Windows, and Linux using the same Angular methods you’ve learned for the web plus.

**Speed and Performance**

* **Code generation**

Angular turns your templates into code that’s highly optimized for JavaScript virtual machines, giving you all the benefits of hand-written code with the productivity of a framework.

* **Universal**

You can use any technology with Angular for serving the application like node.js, .NET, PHP and other servers.

* **Code splitting**

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

**Productivity**

* **Templates**

Quickly create UI views with simple and powerful template syntax.

* **Angular CLI**

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

* **IDEs**

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

**Full Development Story**

* **Testing**

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

Moving on to the most important part of our Angular tutorial, let us discuss about the building blocks of Angular.

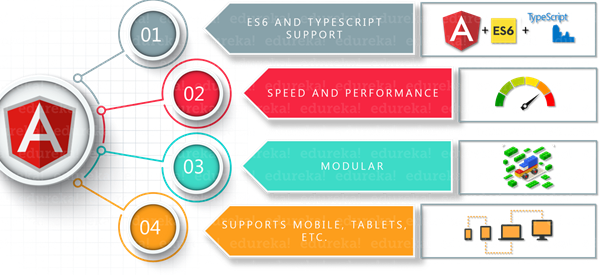
**Building Blocks of Angular**

The main building blocks of Angular are:

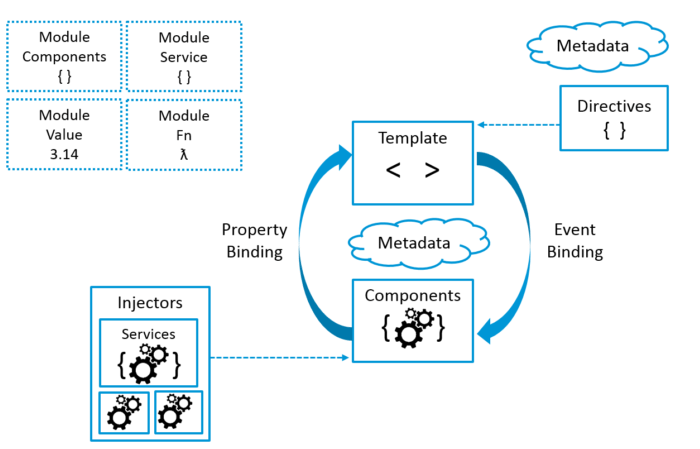
* Modules
* Components
* Templates
* Metadata
* Data binding
* Directives
* Services
* Dependency injection

## ****Angular Tutorial****

Angular is a TypeScript-based open-source front-end web application platform led by the Angular Team at Google and by a community of individuals and corporations. Angular is a complete rewrite from the same team that built AngularJS. But let me tell you that Angular is completely different from AngularJS. Let us understand the differences between Angular and AngularJS.



**Angular Architecture**

****

**Modules**

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

***Decorators***are functions that modify JavaScript classes. Decorators are basically used for attaching metadata to classes so that, it knows the configuration of those classes and how they should work. *NgModule*is a decorator function that takes metadata object whose properties describe the module. The properties are:

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

Let us take a look how the root module (i.e. ***src/app/app.module.ts***) looks like:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | import { NgModule } from '@angular/core';  import { BrowserModule } from '@angular/platform-browser';    @NgModule({  imports:[ BrowserModule ],  providers: [ BookList ],  declarations: [ AppComponent ],  exports: [],  bootstrap: [ AppComponent ]  })    export class AppModule { } |

A root module generally doesn’t *export* it’s class because as root module is the one which imports other modules  & components to use them. We bootstrap the *AppModule*in a *main.ts* file, where we specify the bootstrap module and inside the bootstrap module, contains the bootstrap component.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | import { enableProdMode } from '@angular/core';  import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';  import { AppModule } from './app/app.module';  import { environment } from './environments/environment';    if (environment.production) {  enableProdMode();  }    platformBrowserDynamic().bootstrapModule(AppModule); |

**Angular libraries**

Angular gives us a collection of JavaScript modules (library modules) which provide various functionalities. Each Angular library has *@angular* prefix, like *@angular/core, @angular/compiler, @angular/compiler-cli, @angular/http, @angular/router*. You can install them using the **npm** package manager and import parts of them with JavaScript import statements.

|  |  |
| --- | --- |
| 1 | import { Component } from '@angular/core'; |

In the above example, Angular’s Component decorator is imported from the @angular/core library.

Now moving ahead in Angular Tutorial, let us understand the next and one of the important building block of Angular, i.e. Component.

**Components**

A *component* controls one or more section on the screen called a *view*. For example, if you are building a movie list application, you can have components like App Component (*the bootstrapped component)*, Movielist Component, Movie Description Component, etc.

Inside the component, you define a component’s application logic i.e. how does it support the view—inside a class. The class interacts with the view through an API of properties and methods.

Every app has a main component which is bootstrapped inside the main module, i.e AppComponent.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | import { Component } from '@angular/core';    @Component({  selector:'app-root',  templateUrl:'./app.component.html',  styleUrls: ['./app.component.css']  })    export class AppComponent{  title = 'app works!';  } |

So, after taking a look at the bootstrapped component. Now let us take a look at one more component i.e. Movies Component as to give you more idea about the component.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import { Component, OnInit } from '@angular/core';    @Component({  selector: 'app-movies',  templateUrl: './movies.component.html',  styleUrls: ['./movies.component.css']  })    export class MoviesComponent implements OnInit {    movies: any[];    ngOnInit() {      getMovies()({      this.movies = movies;      })    }  } |

Here, first we import the components and dependencies that we require in our component using import keyword. Then, we attach the metadata of the component using @Component decorator. The first property is the selector, which targets the keyword and dumps the data within selected tag. You can attach the view using two ways: either you can attach template property and specify the template here or you can use templateUrl and provide the path of the file which contains the template. We will be discussing about the template in detail later. The third property is styleUrls which gives the path of the css style sheet.

Then after the metadata, we specify the logic of the component which resides inside the class. We specify the constructor and inside it, we specify the variables and method, which needs to be initialized when the class is created. Then we also have **ngOnInit**method that is called when a component is instantiated.

Moving to next building block in our Angular tutorial, i.e. Templates.

**Templates**

You associate component’s view with its companion **template**. A template is nothing but a form of HTML tags that tells Angular about how to render the component. A template looks like regular HTML, except for a few differences. Here is a template for our Movie Component as we discussed above:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | <app-navbar></app-navbar>      <div class ="container">  <flash-messages></flash-messages>  <router-outlet></router-outlet>  </div> |

Here we have custom tags like <app-navbar>.

**Metadata**

Metadata tells Angular how to process a class. To tell Angular that MovieList Component is a component, **metadata** is attached to the class. In TypeScript, you attach metadata by using a **decorator**. In the below code, you can see metadata attached to the Movie Component:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | import { Component, OnInit } from '@angular/core';    @Component({  selector: 'app-movies',  templateUrl: './movies.component.html',  styleUrls: ['./movies.component.css']  }) |

Here is the @Component decorator, which identifies the class immediately below it as a component class. The @Component decorator takes the required configuration object which Angular needs to create and present the component and its view.

The most important configurations of @Component decorator are:

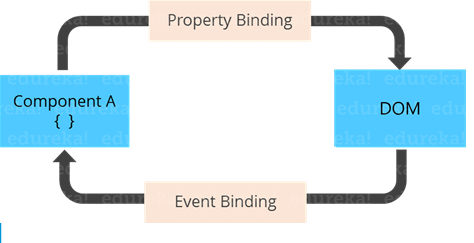
* *selector*: Selector tells Angular to create and insert an instance of this component where it finds <app-movies> tag*.*For example, if an app’s HTML contains <app-movies></app-movies>, then Angular inserts an instance of the MovieListComponent view between those tags.
* *templateUrl*: It contains the path of this component’s HTML template.
* *providers*: An array of **dependency injection providers** for services that the component requires. This is one way to tell Angular that the component’s constructor requires a *MovieService* to get the list of movies to display.

The metadata in the @Component tells Angular where to get the major building blocks you specify for the component.*The template, metadata, and component together describe a view.*The architectural takeaway is that you must add metadata to your code, so that Angular knows what to do.

[**VIEW UPCOMING ANGULAR BATCHES**](https://www.edureka.co/angular-training)

**Data Binding**

If you are not using a framework, you have to push data values into the HTML controls and turn user responses into some actions and value updates. Writing such push/pull logic is tedious, error-prone, and a nightmare to read. Angular supports **data binding**, a mechanism for coordinating parts of a template with parts of a component. You should add binding markup to the template HTML to tell Angular how to connect both sides.



**Figure:***Angular Tutorial – Data Binding*

Each form has a direction — to the DOM, from the DOM, or in both directions.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | <li> {{movie.name}}</li>      <movie-detail [movie]="selectedMovie"></movie-detail>      <li (click)="selectMovie(Movie)"></li> |

* The {{movie.name}} *interpolation* displays the component’s name property value within the <li> element.
* The [movie] *property binding* passes the value of selectedMovie from the parent MovieListComponent to the movie property of the child MovieDetailComponent.
* The(click) *event binding* calls the component’s selectMovie method when the user clicks a movies’s name.

**Two-way data binding** is an important part as it combines property and event binding in a single notation, using the ngModel directive. Here’s an example from the MovieDetailComponent template:

|  |  |
| --- | --- |
| 1 | <input [(ngModel)]="movie.name"> |

In two-way binding, a data property value flows to the input box from the component as with property binding. The user’s changes also flow back to the component, resetting the property to the latest value, as with event binding. Angular processes all data bindings once per JavaScript event cycle, from the root of the application component tree through all child components.

Data binding plays an important role in communication between a template and its component. Data binding is also important for communication between parent and child components.

Moving further in this Angular Tutorial, we will discuss about next building block, i.e. directive.

**Directives**

Angular templates are *dynamic*. When Angular renders them, it transforms the DOM according to the instructions given by **directives**. A directive is a class with a @Directive decorator. A component is a *directive-with-a-template*; a @Component decorator is actually a @Directive decorator extended with template-oriented features.

While **a component is technically a directive**, components are so distinctive and central to Angular applications that this architectural overview separates components from directives.

Two *other* kinds of directives exist: ***structural***and ***attribute***directives.

Directive tends to appear within an element tag as attributes do, sometimes by name but more often as the target of an assignment or a binding.

**Structural** directives alter layout by adding, removing, and replacing elements in DOM.

This example template uses two built-in structural directives:

|  |  |
| --- | --- |
| 1  2  3  4 | <li \*ngFor="let movie of movies"></li>      <movie-detail \*ngIf="selectedMovie"></movie-detail> |

* \*ngFor tells Angular to retrieve one <li> per movie in the movies
* \*ngIf includes the MovieDetail component only if a selected movie exists.

**Attribute** directives alter the appearance or behavior of an existing element. In templates, they look like regular HTML attributes. The ngModel directive, which implements two-way data binding, is an example of an attribute directive. ngModel modifies the behavior of an existing element by setting its display value property and responding to change events.

|  |  |
| --- | --- |
| 1 | <input [(ngModel)]="movie.name"> |

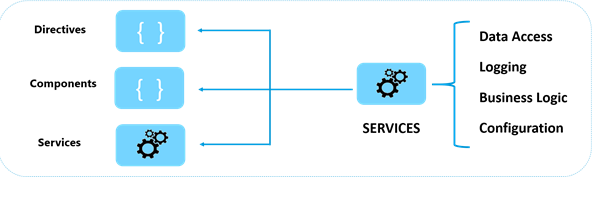
Angular has a few more directives that either alter the layout structure (for example, ngSwitch) or modify aspects of DOM elements and components (for example, ngStyle and ngClass).

You can also write your own directives, i.e. *Custom Directive*.

**Services**

*Service* is a broad category encompassing any value, function, or feature that your application needs. A service is typically a class with a well-defined purpose. Anything can be a service. Examples include:

* logging service
* data service
* message bus
* tax calculator
* application configuration



**Figure:***Angular Tutorial – Services*

Angular has no definition of a service. There is no service base class, and no place to register a service. Yet services are fundamental to any Angular application. Components are the consumers of services.

Here’s an example of a service class where we are using Google’s Firebase as database and importing the movie-list:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39 | import { Injectable } from '@angular/core';  import { AngularFire, FirebaseListObservable, FirebaseObjectObservable } from 'angularfire2';  import { Movie } from '../interfaces/movie'    @Injectable()    export class FirebaseService {  movies: FirebaseListObservable<any[]>;  movie: FirebaseObjectObservable<any>;  titles: FirebaseListObservable<any[]>;  term: string = 'Incep';    constructor(privateaf: AngularFire) { }    getMovies()  {      this.movies = this.af.database.list('/movies') asFirebaseListObservable<Movie[]>;      returnthis.movies;  }    getMovieDetails(id)  {      this.movie = this.af.database.object('/movies/'+id) asFirebaseObjectObservable<Movie>;      return this.movie;  }    searchMovies()  {      this.movies = this.af.database.list('/movies',{      query: {          orderByChild:'title',          startAt:this.term,          endAt:this.term + "\uf8ff",          value:'once'  }  } ) as FirebaseListObservable<Movie[]>;  return this.movies;  }  } |

 Services are everywhere. Component classes don’t fetch data from the server, validate user input, or log directly to the console. They delegate such tasks to services.

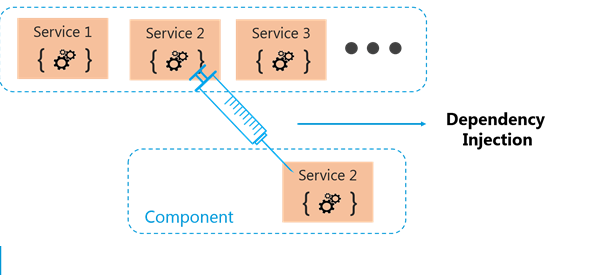
A component’s job is to enable the user experience and nothing more. It mediates between the view (rendered by the template) and the application logic. A good component presents properties and methods for data binding. Angular does help you *follow* these principles by making it easy to factor your application logic into services and make those services available to components through *dependency injection*.

So, now let us understand dependency injection, in this Angular Tutorial blog.

**Dependency Injection**

*Dependency injection* is a way to supply a new instance of a class with the fully-formed dependencies it requires. Most dependencies are services. Angular uses dependency injection to provide new components with the services they need. Angular can tell which services a component needs by looking at the types of its constructor parameters.

When Angular creates a component, it first asks an **injector** for the services that the component requires.



**Figure:***Angular Tutorial – Dependency Injection*

An injector maintains a container of service instances that it has previously created. If a requested service instance is not in the container, the injector makes one and adds it to the container before returning the service to Angular. When all requested services have been resolved and returned, Angular can call the component’s constructor with those services as arguments. This is *dependency injection*.

In the below example you can see, we are registering a **provider** of the Firebase Service with the injector. A provider is something that can create or return a service, typically the service class itself. You can register providers in modules or in components.

In general, add providers to the root module so that the same instance of a service is available everywhere as shown in the code below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | @NgModule({  declarations: [  AppComponent,  HomeComponent,  MoviesComponent,  NavbarComponent,  MovieDetailsComponent,  AboutComponent,  BoldTextDirective,  MovieSearchComponent,  ],    imports: [  BrowserModule,  FormsModule,  HttpModule,  RouterModule.forRoot(routes),  AngularFireModule.initializeApp(firebaseConfig,firebaseAuthConfig),  FlashMessagesModule  ],    providers: [FirebaseService],  bootstrap: [AppComponent]  })    export class AppModule { } |

Alternatively, you can register a service at a component level in the providers property of the @Component decorator:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | import { Component } from '@angular/core';  import { FirebaseService } from './services/firebase.service';    @Component({  selector:'app-root',  templateUrl:'./app.component.html',  styleUrls: ['./app.component.css'],  providers: [FirebaseService]  })  export class AppComponent {  title = 'app works!';  } |

Registering at a component level means you get a new instance of the service with each new instance of that component.

Concluding about dependency injection, you can say that:

* Dependency injection is wired into the Angular framework and used everywhere.
* The *injector* is the main mechanism.
  + An injector maintains a *container* of service instances that it created.
  + An injector can create a new service instance from a *provider*.
* A *provider* is a recipe for creating a service.
* Register *providers* with injectors.

**Features:**

ngIf

Angular2 supported only the **if** condition. However, Angular 4 supports the **if else** condition as well. Let us see how it works using the ng-template.

<span \*ngIf="isavailable; else condition1">Condition is valid.</span>

<ng-template #condition1>Condition is invalid</ng-template>

as keyword in for loop

With the help of **as** keyword you can store the value as shown below −

<div \*ngFor="let i of months | slice:0:5 as total">

Months: {{i}} Total: {{total.length}}

</div>

The variable total stores the output of the slice using the **as** keyword.

Animation Package

Animation in Angular 4 is available as a separate package and needs to be imported from @angular/animations. In Angular2, it was available with @**angular/core**. It is still kept the same for its backward compatibility aspect.

Template

**Angular 4** uses **<ng-template>** as the tag instead of **<template>;** the latter was used in Angular2. The reason Angular 4 changed **<template>** to **<ng-template>** is because of the name conflict of the **<template>** tag with the html **<template>** standard tag. It will deprecate completely going ahead. This is one of the major changes in Angular 4.

TypeScript 2.2

Angular 4 is updated to a recent version of TypeScript, which is 2.2. This helps improve the speed and gives better type checking in the project.

Pipe Title Case

Angular 4 has added a new pipe title case, which changes the first letter of each word into uppercase.

<div>

<h2>{{ 'Angular 4 titlecase' | titlecase }}</h2>

</div>

The above line of code generates the following output – **Angular 4 Titlecase**.

Http Search Parameters

Search parameters to the http get api is simplified. We do not need to call **URLSearchParams** for the same as was being done in Angular2.

Smaller and Faster Apps

Angular 4 applications are smaller and faster when compared to Angular2. It uses the TypeScript version 2.2, the latest version which makes the final compilation small in size.

In this chapter, we will discuss the Environment Setup required for Angular 4.

To install Angular 4, we require the following −

* Nodejs
* Npm
* Angular CLI
* IDE for writing your code

Nodejs has to be greater than 4 and npm has to be greater than 3.

### Nodejs

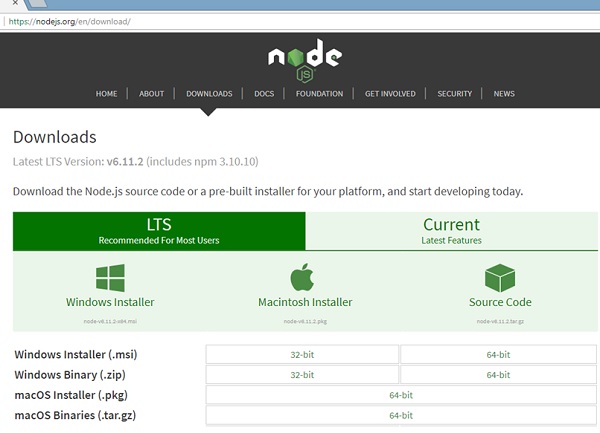
### To check if nodejs is installed on your system, type node –v in the terminal. This will help you see the version of nodejs currently installed on your system.

C:\>node –v

v6.11.0

If it does not print anything, install nodejs on your system. To install nodejs, go the homepage <https://nodejs.org/en/download/> of nodejs and install the package based on your OS.

The homepage of nodejs will look like the following −

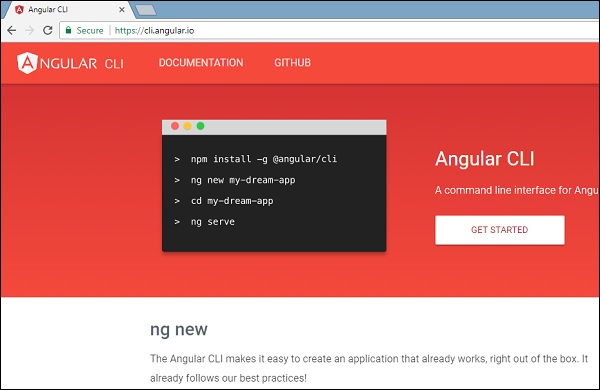


Based on your OS, install the required package. Once nodejs is installed, npm will also get installed along with it. To check if npm is installed or not, type npm –v in the terminal. It should display the version of the npm.

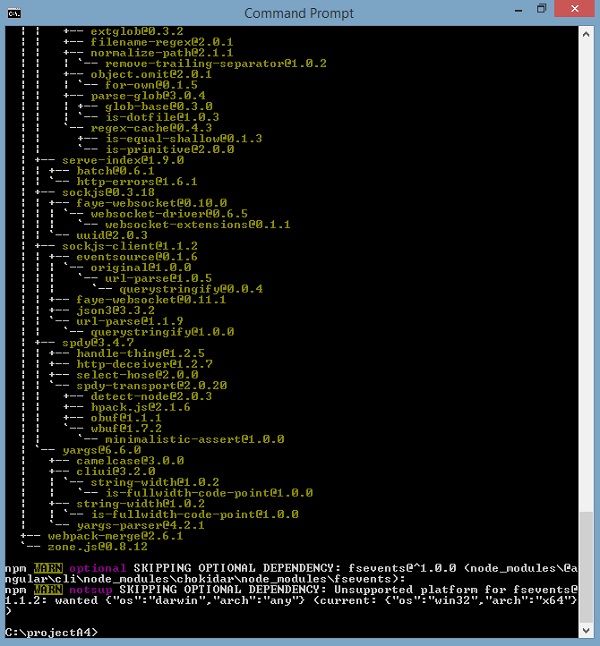
C:\>npm –v

5.3.0

Angular 4 installations are very simple with the help of angular CLI. Visit the homepage <https://cli.angular.io/> of angular to get the reference of the command.



Type **npm install –g @angular/cli**, to install angular cli on your system.



You will get the above installation in your terminal, once Angular CLI is installed. You can use any IDE of your choice, i.e., WebStorm, Atom, Visual Studio Code, etc.

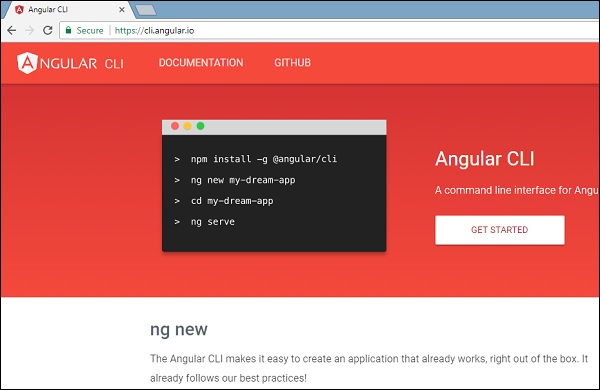
The details of the project setup is explained in the next chapter.

AngularJS is based on the model view controller, whereas Angular 2 is based on the components structure. Angular 4 works on the same structure as Angular2 but is faster when compared to Angular2.

Angular4 uses TypeScript 2.2 version whereas Angular 2 uses TypeScript version 1.8. This brings a lot of difference in the performance.

To install Angular 4, the Angular team came up with Angular CLI which eases the installation. You need to run through a few commands to install Angular 4.

Go to this site [https://cli.angular.io](https://cli.angular.io/) to install Angular CLI.

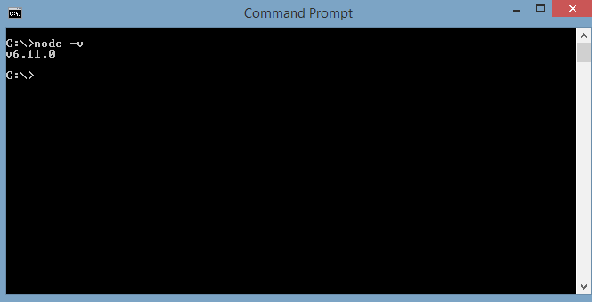


To get started with the installation, we first need to make sure we have nodejs and npm installed with the latest version. The npm package gets installed along with nodejs.

Go to the nodejs site <https://nodejs.org/en/>.

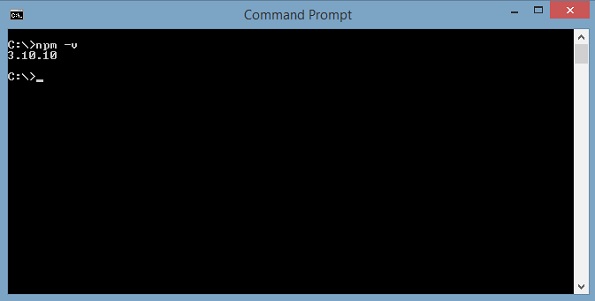


The latest version of Nodejs v6.11.0 is recommended for users. Users who already have nodejs greater than 4 can skip the above process. Once nodejs is installed, you can check the version of node in the command line using the command, node **–v**, as shown below −



The command prompt shows v6.11.0. Once nodejs is installed, npm will also get installed along with it.

To check the version of npm, type command **npm –v** in the terminal. It will display the version of npm as shown below.



The version of npm is 3.10.10. Now that we have nodejs and npm installed, let us run the angular cli commands to install Angular 4. You will see the following commands on the webpage −

npm install -g @angular/cli //command to install angular 4

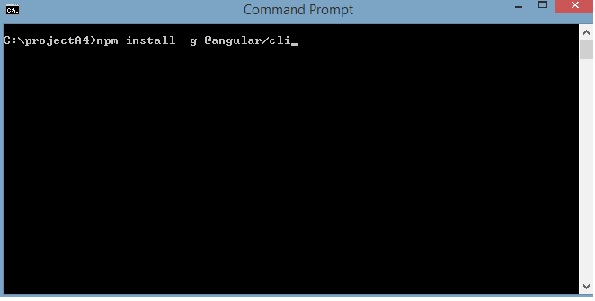
ng new Angular 4-app // name of the project

cd my-dream-app

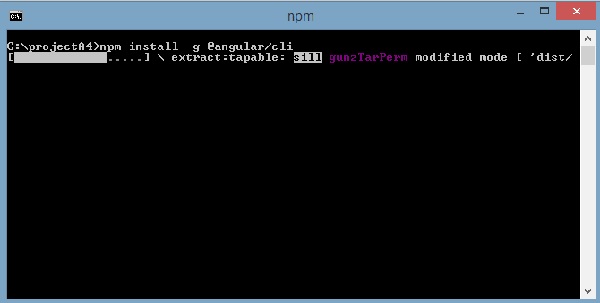
ng serve

Let us start with the first command in the command line and see how it works.

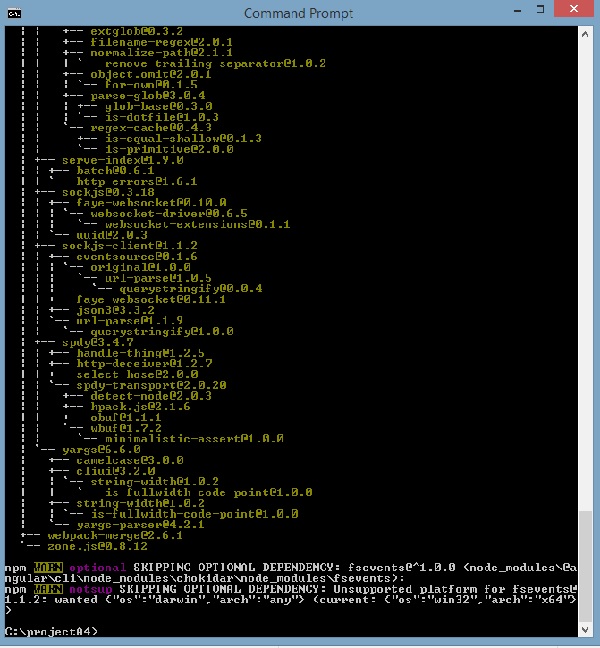
To start with, we will create an empty directory wherein, we will run the Angular CLI command.



Enter the above command to install Angular 4. The installation process will start and will take a few minutes to complete.



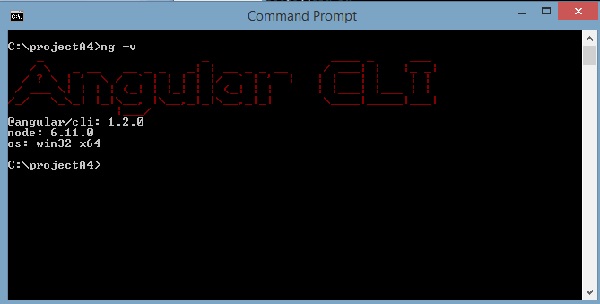
Once the above command to install is complete, the following Command Prompt appears −



We have created an empty folder **ProjectA4** and installed the Angular CLI command. We have also used **-g** to install Angular CLI globally. Now, you can create your Angular 4 project in any directory or folder and you don’t have to install Angular CLI project wise, as it is installed on your system globally and you can make use of it from any directory.

Let us now check whether Angular CLI is installed or not. To check the installation, run the following command in the terminal −

ng -v



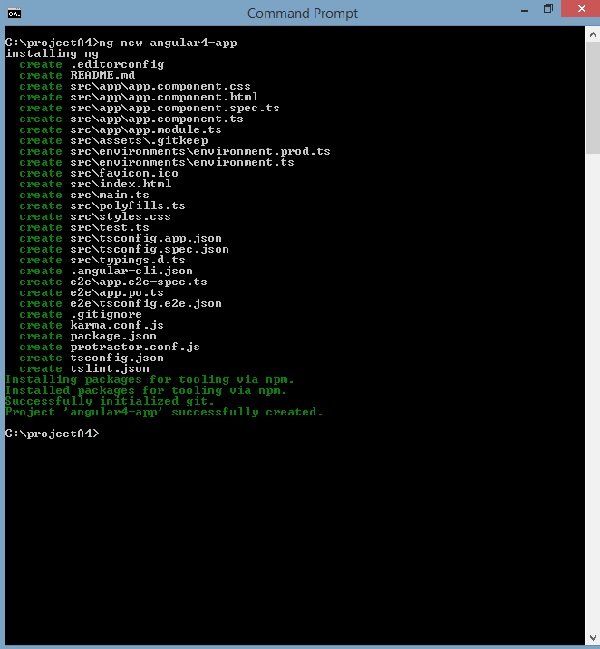
We get the @angular/cli version, which is at present 1.2.0. The node version running is 6.11.0 and also the OS details. The above details tell us that we have installed angular cli successfully and now we are ready to commence with our project.

We have now installed Angular 4. Let us now create our first project in Angular 4. To create a project in Angular 4, we will use the following command −

ng new projectname

We will name the project **ng new Angular 4-app**.

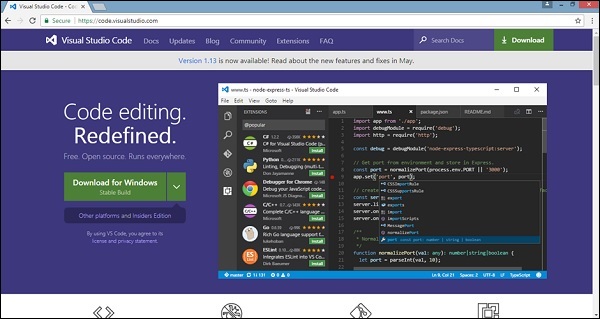
Let us now run the above command in the command line.



The project **Angular 4-app** is created successfully. It installs all the required packages necessary for our project to run in Angular 4. Let us now switch to the project created, which is in the directory **Angular 4-app**. Change the directory in the command line - **cd Angular 4-app**.

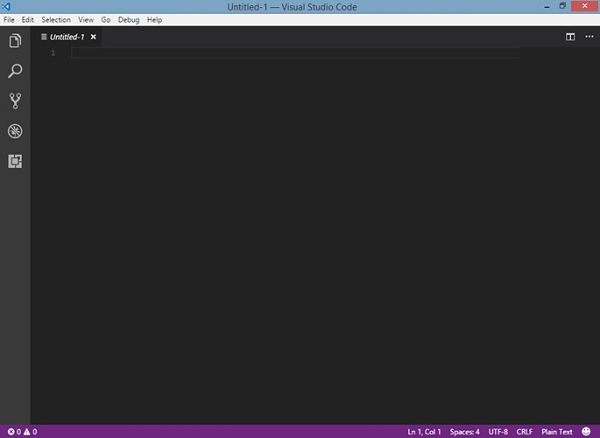
We will use Visual Studio Code IDE for working with Angular 4; you can use any IDE, i.e., Atom, WebStorm, etc.

To download Visual Studio Code, go to <https://code.visualstudio.com/> and click **Download for Windows**.

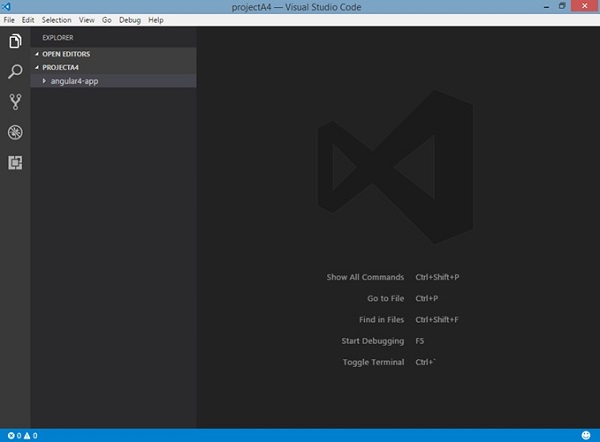


Click **Download for Windows** for installing the IDE and run the setup to start using IDE.

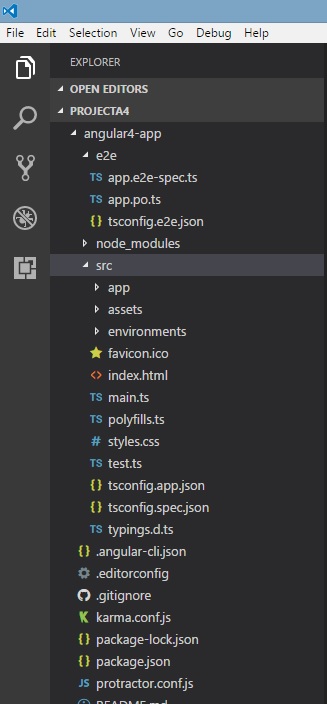
The Editor looks as follows −



We have not started any project in it. Let us now take the project we have created using angular-cli.



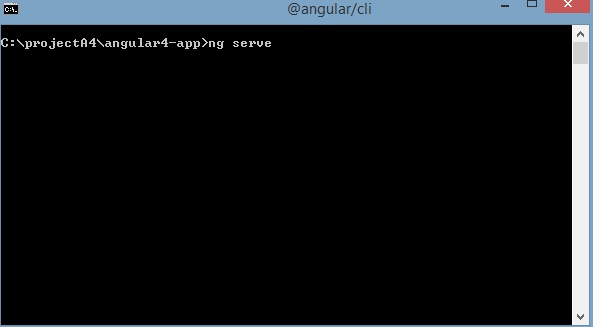
We will consider the **Angular 4-app** project. Let us open the Angular 4-app and see how the folder structure looks like.

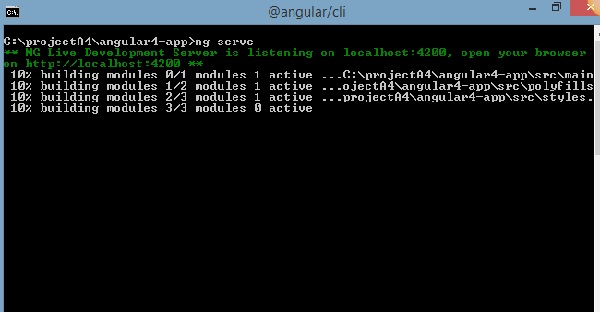


Now that we have the file structure for our project, let us compile our project with the following command −

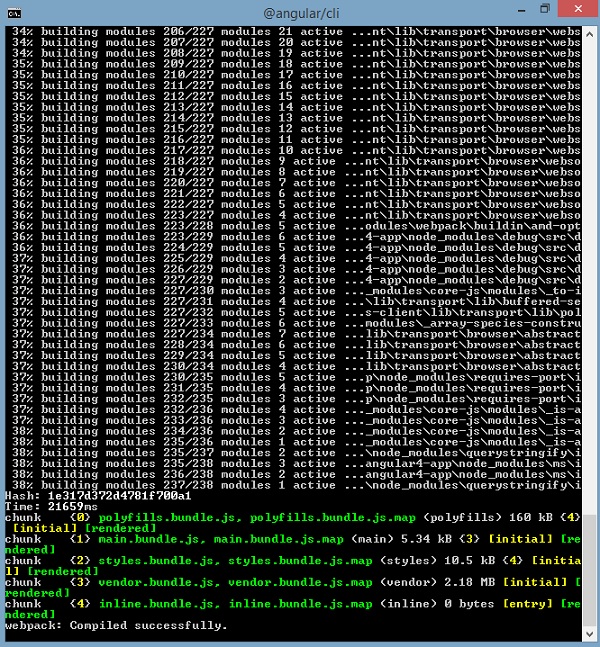
ng serve

The **ng serve** command builds the application and starts the web server.

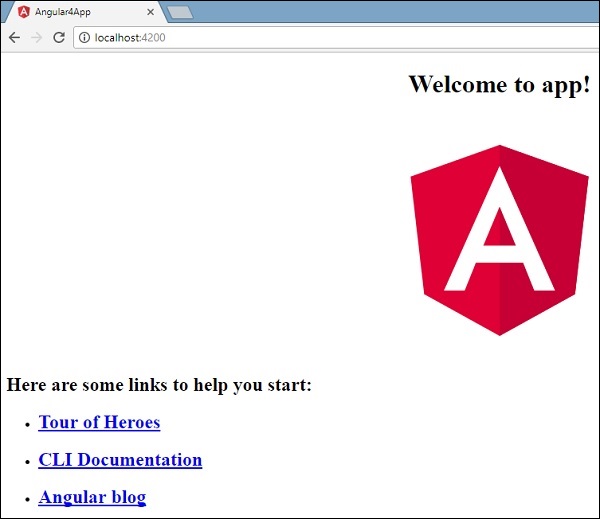




The web server starts on port 4200. Type the url **http://localhost:4200/** in the browser and see the output. Once the project is compiled, you will receive the following output −

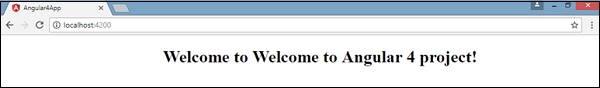


Once you run **http://localhost:4200/** in the browser, you will be directed to the following screen −



Let us now make some changes to display the following content −

**“Welcome to Angular 4 project”**



We have made changes in the files – **app.component.html** and **app.component.ts**. We will discuss more about this in our subsequent chapters.

Let us complete the project setup. If you see we have used port 4200, which is the default port that angular–cli makes use of while compiling. You can change the port if you wish using the following command −

ng serve --host 0.0.0.0 –port 4205

The Angular 4 app folder has the following **folder structure** −

* **e2e** − end to end test folder. Mainly e2e is used for integration testing and helps ensure the application works fine.
* **node\_modules** − The npm package installed is node\_modules. You can open the folder and see the packages available.
* **src** − This folder is where we will work on the project using Angular 4.

The Angular 4 app folder has the following **file structure** −

* **.angular-cli.json** − It basically holds the project name, version of cli, etc.
* **.editorconfig** − This is the config file for the editor.
* **.gitignore** − A .gitignore file should be committed into the repository, in order to share the ignore rules with any other users that clone the repository.
* **karma.conf.js** − This is used for unit testing via the protractor. All the information required for the project is provided in karma.conf.js file.
* **package.json** − The package.json file tells which libraries will be installed into node\_modules when you run npm install.

At present, if you open the file in the editor, you will get the following modules added in it.

"@angular/animations": "^4.0.0",

"@angular/common": "^4.0.0",

"@angular/compiler": "^4.0.0",

"@angular/core": "^4.0.0",

"@angular/forms": "^4.0.0",

"@angular/http": "^4.0.0",

"@angular/platform-browser": "^4.0.0",

"@angular/platform-browser-dynamic": "^4.0.0",

"@angular/router": "^4.0.0",

In case you need to add more libraries, you can add those over here and run the npm install command.

* **protractor.conf.js** − This is the testing configuration required for the application.
* **tsconfig.json** − This basically contains the compiler options required during compilation.
* **tslint.json** − This is the config file with rules to be considered while compiling.

The **src folder** is the main folder, which **internally has a different file structure**.

app

It contains the files described below. These files are installed by angular-cli by default.

* **app.module.ts** − If you open the file, you will see that the code has reference to different libraries, which are imported. Angular-cli has used these default libraries for the import – angular/core, platform-browser. The names itself explain the usage of the libraries.

They are imported and saved into variables such as **declarations, imports, providers**, and **bootstrap**.

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

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

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

@NgModule({

declarations: [

AppComponent

],

imports: [

BrowserModule

],

providers: [],

bootstrap: [AppComponent]

})

export class AppModule { }

**declarations** − In declarations, the reference to the components is stored. The Appcomponent is the default component that is created whenever a new project is initiated. We will learn about creating new components in a different section.

**imports** − This will have the modules imported as shown above. At present, BrowserModule is part of the imports which is imported from @angular/platform-browser.

**providers** − This will have reference to the services created. The service will be discussed in a subsequent chapter.

**bootstrap** − This has reference to the default component created, i.e., AppComponent.

* **app.component.css** − You can write your css structure over here. Right now, we have added the background color to the div as shown below.

.divdetails{

background-color: #ccc;

}

* **app.component.html** − The html code will be available in this file.

<!--The content below is only a placeholder and can be replaced.-->

<div class = "divdetails">

<div style = "text-align:center">

<h1>

Welcome to {{title}}!

</h1>

<img width = "300" src = "data:image/svg+xml;base64,PD94bWwgdmVyc2lvbj0iMS4wIiBlbmNv

ZGluZz0idXRmLTgiPz4NCjwhLS0gR2VuZXJhdG9yOiBBZG9iZSBJbGx1c3RyYXRvciAxOS4xLjAsIFNWRyBFe

HBvcnQgUGx1Zy1JbiAuIFNWRyBWZXJzaW9uOiA2LjAwIEJ1aWxkIDApICAtLT4NCjxzdmcgdmVyc2lvbj0iMS4

xIiBpZD0iTGF5ZXJfMSIgeG1sbnM9Imh0dHA6Ly93d3cudzMub3JnLzIwMDAvc3ZnIiB4bWxuczp4bGluaz0iaH

R0cDovL3d3dy53My5vcmcvMTk5OS94bGluayIgeD0iMHB4IiB5PSIwcHgiDQoJIHZpZXdCb3g9IjAgMCAyNTAg

MjUwIiBzdHlsZT0iZW5hYmxlLWJhY2tncm91bmQ6bmV3IDAgMCAyNTAgMjUwOyIgeG1sOnNwYWNlPSJwcmVzZXJ2

ZSI+DQo8c3R5bGUgdHlwZT0idGV4dC9jc3MiPg0KCS5zdDB7ZmlsbDojREQwMDMxO30NCgkuc3Qxe2ZpbGw6I0M

zMDAyRjt9DQoJLnN0MntmaWxsOiNGRkZGRkY7fQ0KPC9zdHlsZT4NCjxnPg0KCTxwb2x5Z29uIGNsYXNzPSJzdD

AiIHBvaW50cz0iMTI1LDMwIDEyNSwzMCAxMjUsMzAgMzEuOSw2My4yIDQ2LjEsMTg2LjMgMTI1LDIzMCAxMjUsMj

MwIDEyNSwyMzAgMjAzLjksMTg2LjMgMjE4LjEsNjMuMiAJIi8+DQoJPHBvbHlnb24gY2xhc3M9InN0MSIgcG9pbn

RzPSIxMjUsMzAgMTI1LDUyLjIgMTI1LDUyLjEgMTI1LDE1My40IDEyNSwxNTMuNCAxMjUsMjMwIDEyNSwyMzAgMj

AzLjksMTg2LjMgMjE4LjEsNjMuMiAxMjUsMzAgCSIvPg0KCTxwYXRoIGNsYXNzPSJzdDIiIGQ9Ik0xMjUsNTIuMU

w2Ni44LDE4Mi42aDBoMjEuN2gwbDExLjctMjkuMmg0OS40bDExLjcsMjkuMmgwaDIxLjdoMEwxMjUsNTIuMUwxMj

UsNTIuMUwxMjUsNTIuMUwxMjUsNTIuMQ0KCQlMMTI1LDUyLjF6IE0xNDIsMTM1LjRIMTA4bDE3LTQwLjlMMTQyLD

EzNS40eiIvPg0KPC9nPg0KPC9zdmc+DQo=">

</div>

<h2>Here are some links to help you start: </h2>

<ul>

<li>

<h2>

<a target = "\_blank" href="https://angular.io/tutorial">Tour of Heroes</a>

</h2>

</li>

<li>

<h2>

<a target = "\_blank" href = "https://github.com/angular/angular-cli/wiki">

CLI Documentation

</a>

</h2>

</li>

<li>

<h2>

<a target="\_blank" href="http://angularjs.blogspot.ca/">Angular blog</a>

</h2>

</li>

</ul>

</div>

This is the default html code currently available with the project creation.

* **app.component.spec.ts** − These are automatically generated files which contain unit tests for source component.
* **app.component.ts** − The class for the component is defined over here. You can do the processing of the html structure in the .ts file. The processing will include activities such as connecting to the database, interacting with other components, routing, services, etc.

The structure of the file is as follows −

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

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent {

title = 'app';

}

Assets

You can save your images, js files in this folder.

Environment

This folder has the details for the production or the dev environment. The folder contains two files.

* environment.prod.ts
* environment.ts

Both the files have details of whether the final file should be compiled in the production environment or the dev environment.

The additional file structure of Angular 4 app folder includes the following −

favicon.ico

This is a file that is usually found in the root directory of a website.

index.html

This is the file which is displayed in the browser.

<!doctype html>

<html lang = "en">

<head>

<meta charset = "utf-8">

<title>HTTP Search Param</title>

<base href = "/">

<link href = "https://fonts.googleapis.com/icon?family=Material+Icons" rel="stylesheet">

<link href = "https://fonts.googleapis.com/css?family=Roboto|Roboto+Mono" rel="stylesheet">

<link href = "styles.c7c7b8bf22964ff954d3.bundle.css" rel="stylesheet">

<meta name = "viewport" content="width=device-width, initial-scale=1">

<link rel = "icon" type="image/x-icon" href="favicon.ico">

</head>

<body>

<app-root></app-root>

</body>

</html>

The body has **<app-root></app-root>**. This is the selector which is used in **app.component.ts** file and will display the details from app.component.html file.

main.ts

main.ts is the file from where we start our project development. It starts with importing the basic module which we need. Right now if you see angular/core, angular/platform-browser-dynamic, app.module and environment is imported by default during angular-cli installation and project setup.

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

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

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

import { environment } from './environments/environment';

if (environment.production) {

enableProdMode();

}

platformBrowserDynamic().bootstrapModule(AppModule);

The **platformBrowserDynamic().bootstrapModule(AppModule)** has the parent module reference **AppModule**. Hence, when it executes in the browser, the file that is called is index.html. Index.html internally refers to main.ts which calls the parent module, i.e., AppModule when the following code executes −

platformBrowserDynamic().bootstrapModule(AppModule);

When AppModule is called, it calls app.module.ts which further calls the AppComponent based on the boostrap as follows −

bootstrap: [AppComponent]

In app.component.ts, there is a **selector: app-root** which is used in the index.html file. This will display the contents present in app.component.html.

The following will be displayed in the browser −



polyfill.ts

This is mainly used for backward compatibility.

styles.css

This is the style file required for the project.

test.ts

Here, the unit test cases for testing the project will be handled.

tsconfig.app.json

This is used during compilation, it has the config details that need to be used to run the application.

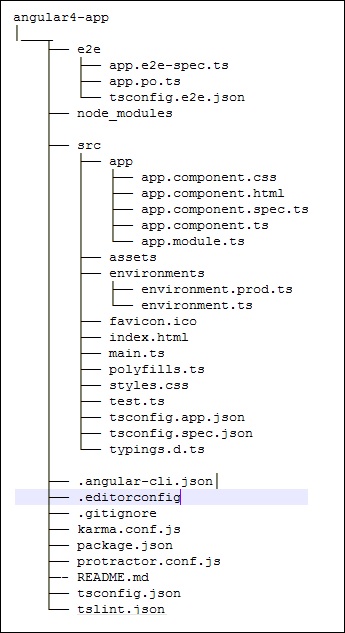
tsconfig.spec.json

This helps maintain the details for testing.

typings.d.ts

It is used to manage the TypeScript definition.

The final file structure looks as follows −



Major part of the development with Angular 4 is done in the components. Components are basically classes that interact with the .html file of the component, which gets displayed on the browser. We have seen the file structure in one of our previous chapters. The file structure has the app component and it consists of the following files −

* **app.component.css**
* **app.component.html**
* **app.component.spec.ts**
* **app.component.ts**
* **app.module.ts**

The above files were created by default when we created new project using the angular-cli command.

If you open up the **app.module.ts** file, it has some libraries which are imported and also a declarative which is assigned the appcomponent as follows −

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

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

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

@NgModule({

declarations: [

AppComponent

],

imports: [

BrowserModule

],

providers: [],

bootstrap: [AppComponent]

})

export class AppModule { }

The declarations include the AppComponent variable, which we have already imported. This becomes the parent component.

Now, angular-cli has a command to create your own component. However, the app component which is created by default will always remain the parent and the next components created will form the child components.

Let us now run the command to create the component.

ng g component new-cmp

When you run the above command in the command line, you will receive the following output −

C:\projectA4\Angular 4-app>ng g component new-cmp

installing component

create src\app\new-cmp\new-cmp.component.css

create src\app\new-cmp\new-cmp.component.html

create src\app\new-cmp\new-cmp.component.spec.ts

create src\app\new-cmp\new-cmp.component.ts

update src\app\app.module.ts

Now, if we go and check the file structure, we will get the new-cmp new folder created under the src/app folder.

The following files are created in the new-cmp folder −

* new-cmp.component.css − css file for the new component is created.
* new-cmp.component.html − html file is created.
* new-cmp.component.spec.ts − this can be used for unit testing.
* new-cmp.component.ts − here, we can define the module, properties, etc.

Changes are added to the app.module.ts file as follows −

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

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

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

import { NewCmpComponent } from './new-cmp/new-cmp.component';

// includes the new-cmp component we created

@NgModule({

declarations: [

AppComponent,

NewCmpComponent // here it is added in declarations and will behave as a child component

],

imports: [

BrowserModule

],

providers: [],

bootstrap: [AppComponent] //for bootstrap the AppComponent the main app component is given.

})

export class AppModule { }

The **new-cmp.component.ts** file is generated as follows −

import { Component, OnInit } from '@angular/core'; // here angular/core is imported .

@Component({

// this is a declarator which starts with @ sign. The component word marked in bold needs to be the same.

selector: 'app-new-cmp', //

templateUrl: './new-cmp.component.html',

// reference to the html file created in the new component.

styleUrls: ['./new-cmp.component.css'] // reference to the style file.

})

export class NewCmpComponent implements OnInit {

constructor() { }

ngOnInit() {}

}

If you see the above new-cmp.component.ts file, it creates a new class called NewCmpComponent, which implements OnInit.In, which has a constructor and a method called ngOnInit(). ngOnInit is called by default when the class is executed.

Let us check how the flow works. Now, the app component, which is created by default becomes the parent component. Any component added later becomes the child component.

When we hit the url in the **http://localhost:4200/** browser, it first executes the index.html file which is shown below −

<!doctype html>

<html lang = "en">

<head>

<meta charset = "utf-8">

<title>Angular 4App</title>

<base href = "/">

<meta name="viewport" content="width = device-width, initial-scale = 1">

<link rel = "icon" type = "image/x-icon" href = "favicon.ico">

</head>

<body>

<app-root></app-root>

</body>

</html>

The above is the normal html file and we do not see anything that is printed in the browser. Take a look at the tag in the body section.

<app-root></app-root>

This is the root tag created by the Angular by default. This tag has the reference in the **main.ts** file.

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

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

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

import { environment } from './environments/environment';

if (environment.production) {

enableProdMode();

}

platformBrowserDynamic().bootstrapModule(AppModule);

AppModule is imported from the app of the main parent module, and the same is given to the bootstrap Module, which makes the appmodule load.

Let us now see the **app.module.ts** file −

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

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

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

import { NewCmpComponent } from './new-cmp/new-cmp.component';

@NgModule({

declarations: [

AppComponent,

NewCmpComponent

],

imports: [

BrowserModule

],

providers: [],

bootstrap: [AppComponent]

})

export class AppModule { }

Here, the AppComponent is the name given, i.e., the variable to store the reference of the **app. Component.ts** and the same is given to the bootstrap. Let us now see the **app.component.ts** file.

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

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent {

title = 'Angular 4 Project!';

}

Angular core is imported and referred as the Component and the same is used in the Declarator as −

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

In the declarator reference to the selector, **templateUrl** and **styleUrl** are given. The selector here is nothing but the tag which is placed in the index.html file that we saw above.

The class AppComponent has a variable called title, which is displayed in the browser.

The **@Component** uses the templateUrl called app.component.html which is as follows −

<!--The content below is only a placeholder and can be replaced.-->

<div style="text-align:center">

<h1>

Welcome to {{title}}.

</h1>

</div>

It has just the html code and the variable title in curly brackets. It gets replaced with the value, which is present in the **app.component.ts** file. This is called binding. We will discuss the concept of binding in a subsequent chapter.

Now that we have created a new component called **new-cmp**. The same gets included in the **app.module.ts** file, when the command is run for creating a new component.

**app.module.ts** has a reference to the new component created.

Let us now check the new files created in new-cmp.

new-cmp.component.ts

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

@Component({

selector: 'app-new-cmp',

templateUrl: './new-cmp.component.html',

styleUrls: ['./new-cmp.component.css']

})

export class NewCmpComponent implements OnInit {

constructor() { }

ngOnInit() {}

}

Here, we have to import the core too. The reference of the component is used in the declarator.

The declarator has the selector called **app-new-cmp** and the **templateUrl**and **styleUrl**.

The .html called **new-cmp.component.html** is as follows −

<p>

new-cmp works!

</p>

As seen above, we have the html code, i.e., the p tag. The style file is empty as we do not need any styling at present. But when we run the project, we do not see anything related to the new component getting displayed in the browser. Let us now add something and the same can be seen in the browser later.

The selector, i.e., **app-new-cmp** needs to be added in the **app.component .html** file as follows −

<!--The content below is only a placeholder and can be replaced.-->

<div style="text-align:center">

<h1>

Welcome to {{title}}.

</h1>

</div>

<app-new-cmp></app-new-cmp>

When the **<app-new-cmp></app-new-cmp>** tag is added, all that is present in the .html file of the new component created will get displayed on the browser along with the parent component data.

Let us see the **new component .html** file and the **new-cmp.component.ts**file.

new-cmp.component.ts

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

@Component({

selector: 'app-new-cmp',

templateUrl: './new-cmp.component.html',

styleUrls: ['./new-cmp.component.css']

})

export class NewCmpComponent implements OnInit {

newcomponent = "Entered in new component created";

constructor() {}

ngOnInit() { }

}

In the class, we have added one variable called new component and the value is “**Entered in new component created**”.

The above variable is bound in the **.new-cmp.component.html** file as follows −

<p>

{{newcomponent}}

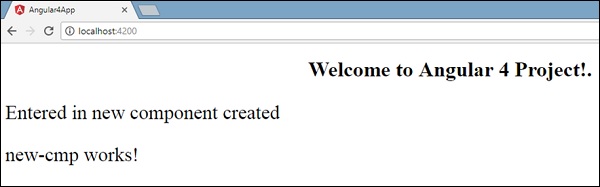
</p>

<p>

new-cmp works!

</p>

Now since we have included the **<app-new-cmp></app-new-cmp>**selector in the **app. component .html** which is the .html of the parent component, the content present in the new component .html file (new-cmp.component.html) gets displayed on the browser as follows −



Similarly, we can create components and link the same using the selector in the **app.component.html** file as per our requirements.

**Module** in Angular refers to a place where you can group the components, directives, pipes, and services, which are related to the application.

In case you are developing a website, the header, footer, left, center and the right section become part of a module.

To define module, we can use the **NgModule**. When you create a new project using the Angular –cli command, the ngmodule is created in the app.module.ts file by default and it looks as follows −

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

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

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

@NgModule({

declarations: [

AppComponent

],

imports: [

BrowserModule

],

providers: [],

bootstrap: [AppComponent]

})

export class AppModule { }

The NgModule needs to be imported as follows −

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

The structure for the ngmodule is as shown below −

@NgModule({

declarations: [

AppComponent

],

imports: [

BrowserModule

],

providers: [],

bootstrap: [AppComponent]

})

It starts with **@NgModule** and contains an object which has declarations, import s, providers and bootstrap.

### Declaration

It is an array of components created. If any new component gets created, it will be imported first and the reference will be included in declarations as shown below −

declarations: [

AppComponent,

NewCmpComponent

]

### Import

It is an array of modules required to be used in the application. It can also be used by the components in the Declaration array. For example, right now in the @NgModule we see the Browser Module imported. In case your application needs forms, you can include the module as follows −

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

The import in the **@NgModule** will be like the following −

imports: [

BrowserModule,

FormsModule

]

### Providers

This will include the services created.

### Bootstrap

This includes the main app component for starting the execution.

# **Angular 4 - Data Binding**

Data Binding is available right from AngularJS, Angular 2 and is now available in Angular 4 as well. We use curly braces for data binding - {{}}; this process is called interpolation. We have already seen in our previous examples how we declared the value to the variable title and the same is printed in the browser.

The variable in the **app.component.html** file is referred as {{title}} and the value of title is initialized in the **app.component.ts** file and in **app.component.html**, the value is displayed.

Let us now create a dropdown of months in the browser. To do that , we have created an array of months in **app.component.ts** as follows −

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

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent {

title = 'Angular 4 Project!';

// declared array of months.

months = ["January", "Feburary", "March", "April", "May",

"June", "July", "August", "September",

"October", "November", "December"];

}

The month’s array that is shown above is to be displayed in a dropdown in the browser. For this, we will use the following line of code −

<!--The content below is only a placeholder and can be replaced. -->

<div style="text-align:center">

<h1>

Welcome to {{title}}.

</h1>

</div>

<div> Months :

<select>

<option \*ngFor="let i of months">{{i}}</option>

</select>

</div>

We have created the normal select tag with option. In option, we have used the **for loop**. The **for loop** is used to iterate over the months’ array, which in turn will create the option tag with the value present in the months.

The syntax **for** in Angular is **\*ngFor = “let I of months”** and to get the value of months we are displaying it in {{i}}.

The two curly brackets help with data binding. You declare the variables in your **app.component.ts** file and the same will be replaced using the curly brackets.

Let us see the output of the above month’s array in the browser



The variable that is set in the **app.component.ts** can be bound with the **app.component.html** using the curly brackets; for example, **{{}}**.

Let us now display the data in the browser based on condition. Here, we have added a variable and assigned the value as true. Using the if statement, we can hide/show the content to be displayed.

### Example

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

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent {

title = 'Angular 4 Project!';

//array of months.

months = ["January", "February", "March", "April",

"May", "June", "July", "August", "September",

"October", "November", "December"];

isavailable = true; //variable is set to true

}

<!--The content below is only a placeholder and can be replaced.-->

<div style = "text-align:center">

<h1>

Welcome to {{title}}.

</h1>

</div>

<div> Months :

<select>

<option \*ngFor = "let i of months">{{i}}</option>

</select>

</div>

<br/>

<div>

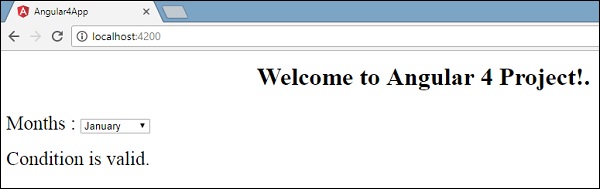
<span \*ngIf = "isavailable">Condition is valid.</span>

//over here based on if condition the text condition is valid is displayed.

If the value of isavailable is set to false it will not display the text.

</div>

### Output



Let us try the above example using the **IF THEN ELSE** condition.

### Example

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

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent {

title = 'Angular 4 Project!';

//array of months.

months = ["January", "February", "March", "April",

"May", "June", "July", "August", "September",

"October", "November", "December"];

isavailable = false;

}

In this case, we have made the **isavailable** variable as false. To print the **else** condition, we will have to create the **ng-template** as follows −

<ng-template #condition1>Condition is invalid</ng-template>

The full code looks like this −

<!--The content below is only a placeholder and can be replaced.-->

<div style="text-align:center">

<h1>

Welcome to {{title}}.

</h1>

</div>

<div> Months :

<select>

<option \*ngFor="let i of months">{{i}}</option>

</select>

</div>

<br/>

<div>

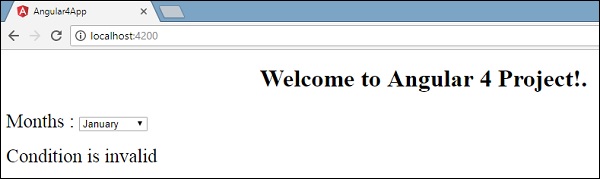
<span \*ngIf="isavailable; else condition1">Condition is valid.</span>

<ng-template #condition1>Condition is invalid</ng-template>

</div>

**If** is used with the else condition and the variable used is **condition1**. The same is assigned as an **id** to the **ng-template**, and when the available variable is set to false the text **Condition is invalid** is displayed.

The following screenshot shows the display in the browser.



Let us now use the **if then else** condition.

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

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent {

title = 'Angular 4 Project!';

//array of months.

months = ["January", "February", "March", "April",

"May", "June", "July", "August", "September",

"October", "November", "December"];

isavailable = true;

}

Now, we will make the variable **isavailable** as true. In the html, the condition is written in the following way −

<!--The content below is only a placeholder and can be replaced.-->

<div style="text-align:center">

<h1>

Welcome to {{title}}.

</h1>

</div>

<div> Months :

<select>

<option \*ngFor="let i of months">{{i}}</option>

</select>

</div>

<br/>

<div>

<span \*ngIf="isavailable; then condition1 else condition2">Condition is valid.</span>

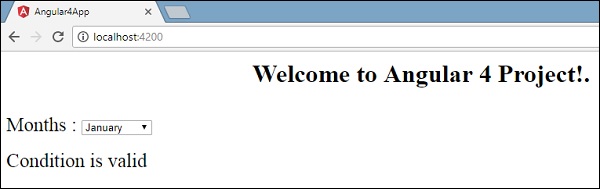
<ng-template #condition1>Condition is valid</ng-template>

<ng-template #condition2>Condition is invalid</ng-template>

</div>

If the variable is true, then **condition1**, else **condition2**. Now, two templates are created with id **#condition1** and **#condition2**.

The display in the browser is as follows −



# **Angular 4 - Event Binding**

In this chapter, we will discuss how Event Binding works in Angular 4. When a user interacts with an application in the form of a keyboard movement, a mouse click, or a mouseover, it generates an event. These events need to be handled to perform some kind of action. This is where event binding comes into picture.

Let us consider an example to understand this better.

### app.component.html

<!--The content below is only a placeholder and can be replaced.-->

<div style = "text-align:center">

<h1>

Welcome to {{title}}.

</h1>

</div>

<div> Months :

<select>

<option \*ngFor = "let i of months">{{i}}</option>

</select>

</div>

<br/>

<div>

<span \*ngIf = "isavailable; then condition1 else condition2">

Condition is valid.

</span>

<ng-template #condition1>Condition is valid</ng-template>

<ng-template #condition2>Condition is invalid</ng-template>

</div>

<button (click)="myClickFunction($event)">

Click Me

</button>

In the **app.component.html** file, we have defined a button and added a function to it using the click event.

Following is the syntax to define a button and add a function to it.

(click)="myClickFunction($event)"

The function is defined in the **.ts** file: **app.component.ts**

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

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent {

title = 'Angular 4 Project!';

//array of months.

months = ["January", "Feburary", "March", "April",

"May", "June", "July", "August", "September",

"October", "November", "December"];

isavailable = true;

myClickFunction(event) {

//just added console.log which will display the event details in browser on click of the button.

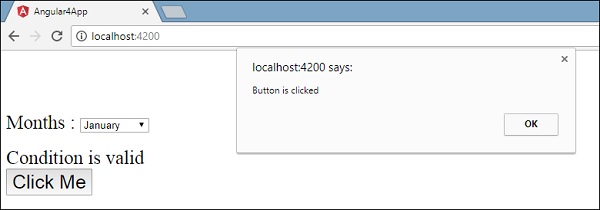
alert("Button is clicked");

console.log(event);

}

}

Upon clicking the button, the control will come to the function **myClickFunction** and a dialog box will appear, which displays **the Button is clicked** as shown in the following screenshot −



Let us now add the change event to the dropdown.

The following line of code will help you add the change event to the dropdown −

<!--The content below is only a placeholder and can be replaced.-->

<div style = "text-align:center">

<h1>

Welcome to {{title}}.

</h1>

</div>

<div> Months :

<select (change) = "changemonths($event)">

<option \*ngFor = "let i of months">{{i}}</option>

</select>

</div>

<br/>

<div>

<span \*ngIf = "isavailable; then condition1 else condition2">

Condition is valid.

</span>

<ng-template #condition1>Condition is valid</ng-template>

<ng-template #condition2>Condition is invalid</ng-template>

</div>

<button (click) = "myClickFunction($event)">Click Me</button>

The function is declared in the **app.component.ts** file −

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

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent {

title = 'Angular 4 Project!';

//array of months.

months = ["January", "Feburary", "March", "April",

"May", "June", "July", "August", "September",

"October", "November", "December"];

isavailable = true;

myClickFunction(event) {

alert("Button is clicked");

console.log(event);

}

changemonths(event) {

console.log("Changed month from the Dropdown");

console.log(event);

}

}

The console message “**Changed month from the Dropdown**” is displayed in the console along with the event.



Let us add an alert message in **app.component.ts** when the value from the dropdown is changed as shown below −

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

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent {

title = 'Angular 4 Project!';

//array of months.

months = ["January", "February", "March", "April",

"May", "June", "July", "August", "September",

"October", "November", "December"];

isavailable = true;

myClickFunction(event) {

//just added console.log which will display the event details in browser

on click of the button.

alert("Button is clicked");

console.log(event);

}

changemonths(event) {

alert("Changed month from the Dropdown");

}

}

When the value in dropdown is changed, a dialog box will appear and the following message will be displayed - “**Changed month from the Dropdown**”.

