Angular JS is an open source framework built over JavaScript. It was built by the developers at Google. This framework was used to overcome obstacles encountered while working with Single Page applications. Also, testing was considered as a key aspect while building the framework. It was ensured that the framework could be easily tested. The initial release of the framework was in October 2010.

Features of Angular 2

Following are the key features of Angular 2 −

* **Components** − The earlier version of Angular had a focus of Controllers but now has changed the focus to having components over controllers. Components help to build the applications into many modules. This helps in better maintaining the application over a period of time.
* **TypeScript** − The newer version of Angular is based on TypeScript. This is a superset of JavaScript and is maintained by Microsoft.
* **Services** − Services are a set of code that can be shared by different components of an application. So for example if you had a data component that picked data from a database, you could have it as a shared service that could be used across multiple applications.

In addition, Angular 2 has better event-handling capabilities, powerful templates, and better support for mobile devices.

Components of Angular 2

Angular 2 has the following components −

* **Modules** − This is used to break up the application into logical pieces of code. Each piece of code or module is designed to perform a single task.
* **Component** − This can be used to bring the modules together.
* **Templates** − This is used to define the views of an Angular JS application.
* **Metadata** − This can be used to add more data to an Angular JS class.
* **Service** − This is used to create components which can be shared across the entire application

To start working with Angular 2, you need to get the following key components installed.

* **Npm** − This is known as the node package manager that is used to work with the open source repositories. Angular JS as a framework has dependencies on other components. And **npm** can be used to download these dependencies and attach them to your project.
* **Git** − This is the source code software that can be used to get the sample application from the **github** angular site.
* **Editor** − There are many editors that can be used for Angular JS development such as Visual Studio code and WebStorm. In our tutorial, we will use Visual Studio code which comes free of cost from Microsoft.

**Modules**

Modules are used in Angular JS to put logical boundaries in your application. Hence, instead of coding everything into one application, you can instead build everything into separate modules to separate the functionality of your application. Let’s inspect the code which gets added to the demo application.

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

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

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

@NgModule ({

imports: [ BrowserModule ],

declarations: [ AppComponent ],

bootstrap: [ AppComponent ]

})

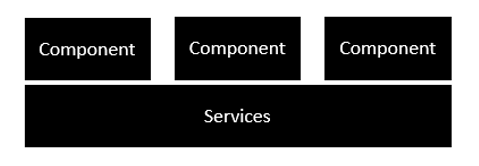
export class AppModule { }

* The import statement is used to import functionality from the existing modules. Thus, the first 3 statements are used to import the NgModule, BrowserModule and AppComponent modules into this module.
* The NgModule decorator is used to later on define the imports, declarations, and bootstrapping options.
* The BrowserModule is required by default for any web based angular application.
* The bootstrap option tells Angular which Component to bootstrap in the application.

A module is made up of the following parts −

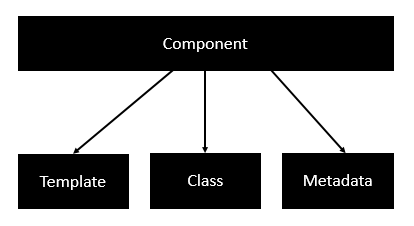
* **Bootstrap array** − This is used to tell Angular JS which components need to be loaded so that its functionality can be accessed in the application. Once you include the component in the bootstrap array, you need to declare them so that they can be used across other components in the Angular JS application.
* **Export array** − This is used to export components, directives, and pipes which can then be used in other modules.
* **Import array** − Just like the export array, the import array can be used to import the functionality from other Angular JS modules.

The following screenshot shows the anatomy of an Angular 2 application. Each application consists of Components. Each component is a logical boundary of functionality for the application. You need to have layered services, which are used to share the functionality across components.



Following is the anatomy of a Component. A component consists of −

* **Class** − This is like a C++ or Java class which consists of properties and methods.
* **Metadata** − This is used to decorate the class and extend the functionality of the class.
* **Template** − This is used to define the HTML view which is displayed in the application.



Following is an example of a component.

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

@Component ({

selector: 'my-app',

templateUrl: 'app/app.component.html'

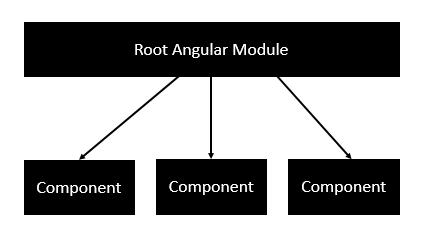
})

export class AppComponent {

appTitle: string = 'Welcome';

}

Each application is made up of modules. Each Angular 2 application needs to have one Angular Root Module. Each Angular Root module can then have multiple components to separate the functionality.



Following is an example of a root module.

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

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

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

@NgModule ({

imports: [ BrowserModule ],

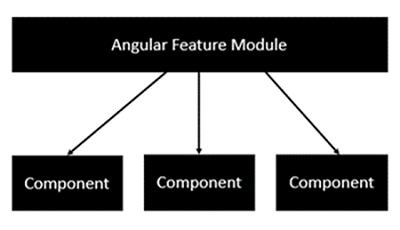
declarations: [ AppComponent ],

bootstrap: [ AppComponent ]

})

export class AppModule { }

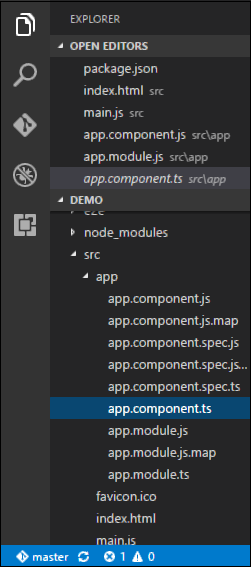
Each application is made up of feature modules where each module has a separate feature of the application. Each Angular feature module can then have multiple components to separate the functionality.



Components are a logical piece of code for Angular JS application. A Component consists of the following −

* **Template** − This is used to render the view for the application. This contains the HTML that needs to be rendered in the application. This part also includes the binding and directives.
* **Class** − This is like a class defined in any language such as C. This contains properties and methods. This has the code which is used to support the view. It is defined in TypeScript.
* **Metadata** − This has the extra data defined for the Angular class. It is defined with a decorator.

Let’s now go to the app.component.ts file and create our first Angular component.



Let’s add the following code to the file and look at each aspect in detail.

## Class

The class decorator. The class is defined in TypeScript. The class normally has the following syntax in TypeScript.

### Syntax

class classname {

Propertyname: PropertyType = Value

}

### Parameters

* **Classname** − This is the name to be given to the class.
* **Propertyname** − This is the name to be given to the property.
* **PropertyType** − Since TypeScript is strongly typed, you need to give a type to the property.
* **Value** − This is the value to be given to the property.

### Example

export class AppComponent {

appTitle: string = 'Welcome';

}

In the example, the following things need to be noted −

* We are defining a class called AppComponent.
* The export keyword is used so that the component can be used in other modules in the Angular JS application.
* appTitle is the name of the property.
* The property is given the type of string.
* The property is given a value of ‘Welcome’.

## Template

This is the view which needs to be rendered in the application.

### Syntax

Template: '

<HTML code>

class properties

'

### Parameters

* **HTML Code** − This is the HTML code which needs to be rendered in the application.
* **Class properties** − These are the properties of the class which can be referenced in the template.

### Example

template: '

<div>

<h1>{{appTitle}}</h1>

<div>To Tutorials Point</div>

</div>

'

In the example, the following things need to be noted −

* We are defining the HTML code which will be rendered in our application
* We are also referencing the appTitle property from our class.

## Metadata

This is used to decorate Angular JS class with additional information.

Let’s take a look at the completed code with our class, template, and metadata.

### Example

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

@Component ({

selector: 'my-app',

template: ` <div>

<h1>{{appTitle}}</h1>

<div>To Tutorials Point</div>

</div> `,

})

export class AppComponent {

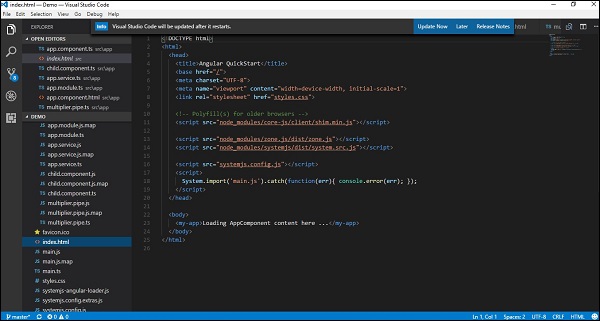
appTitle: string = 'Welcome';

}

In the above example, the following things need to be noted −

* We are using the import keyword to import the ‘Component’ decorator from the angular/core module.
* We are then using the decorator to define a component.
* The component has a selector called ‘my-app’. This is nothing but our custom html tag which can be used in our main html page.

Now, let’s go to our index.html file in our code.



Let’s make sure that the body tag now contains a reference to our custom tag in the component. Thus in the above case, we need to make sure that the body tag contains the following code −

In the chapter on Components, we have already seen an example of the following template.

template: '

<div>

<h1>{{appTitle}}</h1>

<div>To Tutorials Point</div>

</div>

'

This is known as an **inline template**. There are other ways to define a template and that can be done via the templateURL command. The simplest way to use this in the component is as follows.

Syntax

templateURL:

viewname.component.html

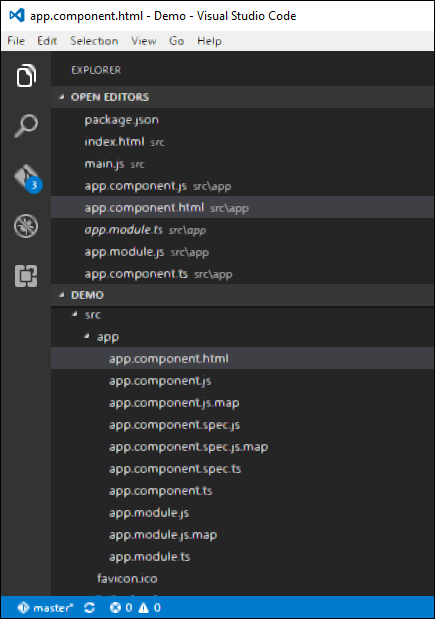
Parameters

* **viewname** − This is the name of the app component module.

After the viewname, the component needs to be added to the file name.

Following are the steps to define an inline template.

**Step 1** − Create a file called app.component.html. This will contain the html code for the view.



**Step 2** − Add the following code in the above created file.

<div>{{appTitle}} Tutorialspoint </div>

This defines a simple div tag and references the appTitle property from the app.component class.

**Step 3** − In the app.component.ts file, add the following code.

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

@Component ({

selector: 'my-app',

templateUrl: 'app/app.component.html'

})

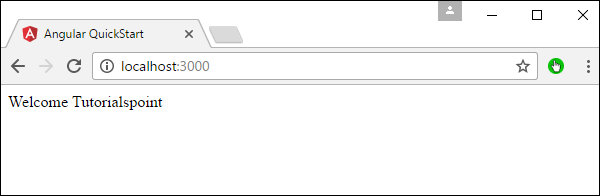
export class AppComponent {

appTitle: string = 'Welcome';

}

From the above code, the only change that can be noted is from the templateURL, which gives the link to the app.component.html file which is located in the app folder.

**Step 4** − Run the code in the browser, you will get the following output.



From the output, it can be seen that the template file (app.component.html) file is being called accordingly.

A **directive** is a custom HTML element that is used to extend the power of HTML. Angular 2 has the following directives that get called as part of the BrowserModule module.

* ngif
* ngFor

If you view the app.module.ts file, you will see the following code and the BrowserModule module defined. By defining this module, you will have access to the 2 directives.

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

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

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

@NgModule ({

imports: [ BrowserModule ],

declarations: [ AppComponent ],

bootstrap: [ AppComponent ]

})

export class AppModule { }

Now let’s look at each directive in detail.

## ngIf

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

### Syntax

\*ngIf = 'expression'

If the expression evaluates to true then the corresponding gets added, else the elements are not added.

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

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

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

@Component ({

selector: 'my-app',

templateUrl: 'app/app.component.html'

})

export class AppComponent {

appTitle: string = 'Welcome';

appStatus: boolean = true;

}

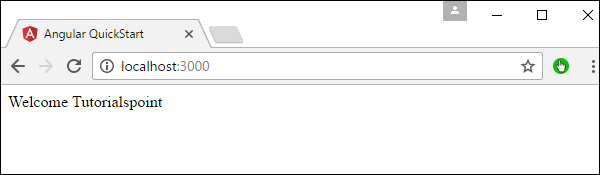
**Step 2** − Now in the app.component.html file, add the following code.

<div \*ngIf = 'appStatus'>{{appTitle}} Tutorialspoint </div>

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

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

### Output



## ngFor

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

### Syntax

\*ngFor = 'let variable of variablelist'

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

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

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

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

@Component ({

selector: 'my-app',

templateUrl: 'app/app.component.html'

})

export class AppComponent {

appTitle: string = 'Welcome';

appList: any[] = [ {

"ID": "1",

"Name" : "One"

},

{

"ID": "2",

"Name" : "Two"

} ];

}

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

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

<div \*ngFor = 'let lst of appList'>

<ul>

<li>{{lst.ID}}</li>

<li>{{lst.Name}}</li>

</ul>

</div>

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

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

# Angular 2 - User Input

In Angular 2, you can make the use of DOM element structure of HTML to change the values of the elements at run time. Let’s look at some in detail.

## The Input Tag

In the app.component.ts file place the following code.

import {

Component

} from '@angular/core';

@Component ({

selector: 'my-app',

template: '

<div>

<input [value] = "name" (input) = "name = $event.target.value">

{{name}}

</div>

'

})

export class AppComponent { }

Following things need to be noted about the above code.

* **[value] = ”username”** − This is used to bind the expression username to the input element’s value property.
* **(input) = ”expression”** − This a declarative way of binding an expression to the input element’s input event.
* **username = $event.target.value** − The expression that gets executed when the input event is fired.
* **$event** − Is an expression exposed in event bindings by Angular, which has the value of the event’s payload.

Once you save all the code changes and refresh the browser, you will get the following output.

You can now type anything and the same input will reflect in the text next to the Input control.

### COMPONENTS

In Angular 2, “everything is a component.” Components are the main way we build and specify elements and logic on the page, through both custom elements and attributes that add functionality to our existing components.

### A SIMPLE COMPONENT

Here’s a simple [Component](https://angular.io/docs/ts/latest/api/core/index/Component-decorator.html) that renders our name, and a button that triggers a method to print our name to the console:

##### **APP.COMPONENT.TS**

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

@Component({

selector: 'my-component',

template: `

<div>Hello my name is {{name}}.

<button (click)="sayMyName()">Say my name</button>

</div>

`

})

export class MyComponent {

name: string;

constructor() {

this.name = 'Max'

}

sayMyName() {

console.log('My name is', this.name)

}

}

When we use the <my-component></my-component> tag in our HTML, this component will be created, our constructor called, and rendered.

### INPUTS

Components are the core of an Angular 2 app but most developers need to know how to pass data into components to dynamically configure them.

#### @Input

To define an input for a component, we use the @Input decorator.

For example, our <user-profile> component needs a user argument to render information about that user:

<user-profile [user]="currentUser"></user-profile>

So, we add an @Input binding to user:

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

@Component({

selector: 'user-profile',

template: '<div>{{user.name}}</div>'

})

export class UserProfile {

@Input() user;

constructor() {}

}

### OUTPUTS

If you want to bind to particular event, you can use the new [Event syntax](http://learnangular2.com/events) in Angular 2, but what if you need your own custom event?

To create a custom event, we can use the new @Output decorator. Take the following component:

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

@Component({

selector: 'user-profile',

template: '<div>Hi, my name is {{user.name}}</div>'

})

export class UserProfile {

constructor() {}

}

Let’s import Output and EventEmitter and create our new event

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

@Component({

selector: 'user-profile',

template: '<div>Hi, my name is {{user.name}}</div>'

})

export class UserProfile {

@Output() userUpdated = new EventEmitter();

constructor() {

// Update user

// ...

this.userUpdated.emit(this.user);

}

}

Now when we used this component in its parent component, we can bind the event that user-profileemits

<user-profile (userUpdated)="handleUserUpdated($event)"></user-profile>

export class SettingsPage {

constructor(){}

handleUserUpdated(user) {

// Handle the event

}

}

### TEMPLATES

Templates are very similar to templates in Angular 1, though there are many small syntactical changes that make it more clear what is happening.

## A SIMPLE TEMPLATE

Let’s start with a very simple template that shows our name and our favorite thing:

<div>

Hello my name is {{name}} and I like {{thing}} quite a lot.

</div>

## {}: RENDERING

To render a value, we can use the standard double-curly syntax:

My name is {{name}}

Pipes, previously known as “Filters,” transform a value into a new value, like localizing a string or converting a floating point value into a currency representation:

## []: BINDING PROPERTIES

To resolve and bind a variable to a component, use the [] syntax. If we have this.currentVolume in our component, we will pass this through to our component and the values will stay in sync:

<video-control [volume]="currentVolume"></video-control>

## (): HANDLING EVENTS

To listen for an event on a component, we use the () syntax

<my-component (click)="onClick($event)"></my-component>

## [()]: TWO-WAY DATA BINDING

To keep a binding up to date given user input and other events, use the [()] syntax. Think of it as a combination of handling an event and binding a property:

<input [(ngModel)]="myName">

The this.myName value of your component will stay in sync with the input value.

## \*: THE ASTERISK

\* indicates that this directive treats this component as a template and will not draw it as-is. For example, ngFor takes our <my-component> and stamps it out for each item in items, but it never renders our initial <my-component> since it’s a template:

<my-component \*ngFor="#item of items">

</my-component>

Other similar directives that work on templates rather than rendered components are \*ngIf and \*ngSwitch.

### ANGULAR 2 EVENTS

Events in Angular 2 use the parentheses notation in templates, and trigger methods in a component’s class. For example, assume we have this component class:

@Component(...)

class MyComponent {

clicked(event) {

}

}

And this template:

<button (click)="clicked()">Click</button>

Our clicked() method will be called when the button is clicked.

## DELEGATION

Events in Angular 2 behave like normal DOM events. They can bubble up and propagate down. Nothing special to do here!

## EVENT OBJECT

To capture the event object, pass $event as a parameter in the event callback from the template:

<button (click)="clicked($event)"></button>

This is an easy way to modify the event, such as calling preventDefault:

@Component(...)

class MyComponent {

clicked(event) {

event.preventDefault();

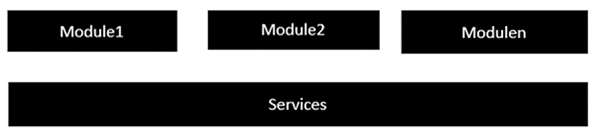
}

}

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# Angular 2 - Services

A service is used when a common functionality needs to be provided to various modules. For example, we could have a database functionality that could be reused among various modules. And hence you could create a service that could have the database functionality.



The following key steps need to be carried out when creating a service.

**Step 1** − Create a separate class which has the injectable decorator. The injectable decorator allows the functionality of this class to be injected and used in any Angular JS module.

@Injectable()

export class classname {

}

**Step 2** − Next in your appComponent module or the module in which you want to use the service, you need to define it as a provider in the @Component decorator.

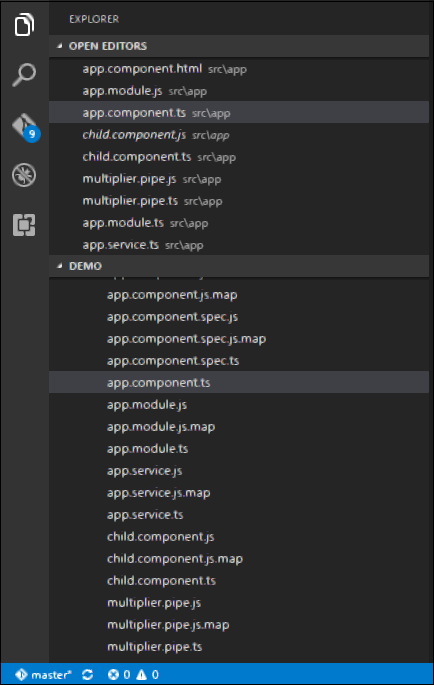
@Component ({

providers : [classname]

})

Let�s look at an example on how to achieve this. Following are the steps involved.

**Step 1** − Create a **ts** file for the service called app.service.ts.



**Step 2** − Place the following code in the file created above.

import {

Injectable

} from '@angular/core';

@Injectable()

export class appService {

getApp(): string {

return "Hello world";

}

}

Following points need to be noted about the above program.

* The Injectable decorator is imported from the angular/core module.
* We are creating a class called appService that is decorated with the Injectable decorator.
* We are creating a simple function called getApp, which returns a simple string called �Hello world�.

**Step 3** − In the app.component.ts file, place the following code.

import {

Component

} from '@angular/core';

import {

appService

} from './app.service';

@Component ({

selector: 'demo-app',

template: '<div>{{value}}</div>',

providers: [appService]

})

export class AppComponent {

value: string = "";

constructor(private \_appService: appService) { }

ngOnInit(): void {

this.value = this.\_appService.getApp();

}

}

Following points need to be noted about the above program.

* First, we import our appService module in the appComponent module.
* Then, we register the service as a provider in this module.
* In the constructor, we define a variable called \_appService of the type appService so that it can be called anywhere in the appComponent module.
* As an example, in the ngOnInit lifecyclehook, we called the getApp function of the service and assign the output to the value property of the AppComponent class.

Once you save all the code changes and refresh the browser, you will get the following output.

