ANGULAR

Pre-reqs:

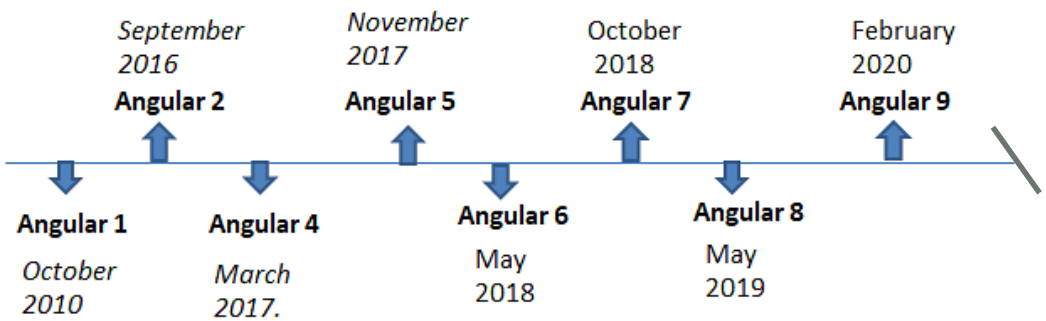
- HTML
- CSS
- Basic JavaScript
- OOP concepts
- Some server side scripting knowledge like ASP.net, JSP/Servlet, PHP etc

Angular

- Softwares:
  - Make sure you install Node 8.9.0 or higher.
  - Your npm version should be 5.6.0 or higher.
  - TypeScript. Make sure you install at least version 2.1 or greater.
  - Highly recommend using the Google Chrome Web Browser to develop Angular apps.
  - install Angular CLI : npm install -g @angular/cli

```
C:\Users\Shrilata>node --version
v10.16.3

C:\Users\Shrilata>npm -v
6.9.0
```

Angular Version History



Angular 10 was released on June 24, 2020

Introduction to Angular

- Angular is **an open source JavaScript client** side framework to build powerful mobile and desktop applications; Created and maintained by Google
  - Especially good for developing reactive Single Page Applications (SPA)
  - Angular is a component based MVC framework. Components are major building blocks of an Angular 2 application
    - What is a component? Well, when written using TypeScript, a component is merely a TypeScript class decorated with @Component() decorator.
  - A SPA is a web application delivered to the browser that doesn't reload the page during use.
    - This means all the code for structure (HTML), presentation (CSS), and behaviour (JavaScript) is retrieved with the initial page load.
    - SPAs offer a faster, more fluid user experience (UX) similar to a native desktop application.
    - Since the page never reloads, this reduces the number of round trips to the server.
- Why Angular?
  - Has been optimized for developer productivity, small payload size and performance.
  - Developed using TypeScript, which is Microsoft's extension of JavaScript that allows use of all ES 2015 (ECMAScript 6) features and adds type checking and object-oriented features like interfaces.

Angular: a platform, not a framework

- A *framework* is usually just the code library used to build an application, whereas a *platform is* more holistic and includes tooling and support beyond a framework.
- Angular comes with a lean core library and makes additional features available as separate packages that can be used as needed.
  - It also has many tools that push it beyond a simple framework, including the following:
    - Dedicated CLI for application development, testing, and deployment
    - Offline rendering capabilities on many back-end server platforms
    - Desktop-, mobile-, and browser-based application execution environments

- What Angular is not
  - A server side framework/technology
  - Javascript library (jQuery, React etc)
  - Design pattern
  - Platform or language (.NET, Java)
  - Plugin or extension

Angular Building Blocks

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

- Angular official site:https://angular.io/

Setting up Local development Environment

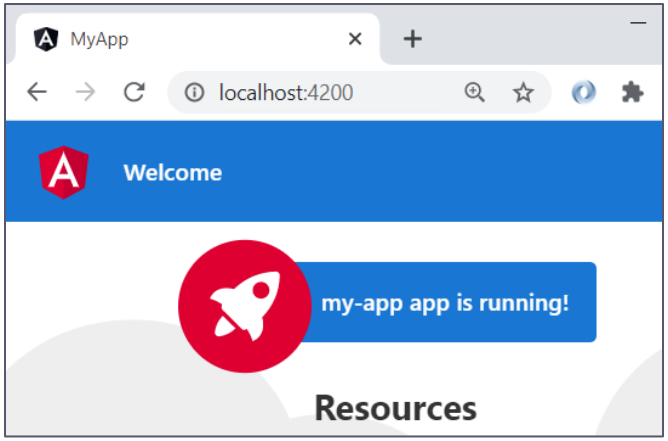
- The Angular CLI creates, manages, builds and test your Angular projects.
  - It is a tool to create projects, generate application and library code, and perform a variety of ongoing development tasks
  - To install the Angular CLI: npm install -g @angular/cli
  - To install Specific Version (Example: 6.1.1) : npm install -g @angular/cli@6.1.1
  - To check if correctly installed: ng --version

```
C:\Users\Administrator>ng --version

Angular CLI: 7.0.3
Node: 8.11.3
OS: win32 x64
Angular:
...

Package                            Version
-----------------------------------------------------------
@angular-devkit/architect          0.10.3
@angular-devkit/core               7.0.3
@angular-devkit/schematics         7.0.3
@schematics/angular                7.0.3
@schematics/update                 0.10.3
rxjs                               6.3.3
typescript                         3.1.3
```

Getting Started with Angular CLI

Generating and serving an Angular project via a development server

- Create and run a new project:
  - ng new my-app
  - cd my-app
  - ng serve or ng serve –open



- Navigate to http://localhost:4200/.
  - The app automatically reloads if you change any of the source files.

- Angular CLI commands:
  - **ng new [name]** creates a new angular application.
    - The Angular CLI installs the necessary npm packages, creates the project files, and populates the project with a simple default app.
  - **ng serve builds** the application and **starts** a web server.
    - Also rebuilds the app as you make changes to your files.
    - ng serve --port 3000 --open

index.html and main.ts

- The index.html is the entry point to your application:
  - Provides the HTML markup for your application with the custom <app-root> element.

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

main.ts

```typescript
import { platformBrowserDynamic }
   from '@angular/platform-browser-dynamic';
import { AppModule } from './app/app.module';
platformBrowserDynamic().bootstrapModule(AppModule)
```

- You need to bootstrap the root module to launch the application.
  - Angular applications provide a bootstrap file, which contains the code required to start the application. The bootstrap file is called **main.ts**
  - You bootstrap the AppModule in the main.ts file.
  - main.ts compiles the application with the JIT compiler and bootstraps the application's main module (AppModule) to run in the browser.

Bootstrapping angular – how does everything work?

cli.json | angular.json | main.ts | app.compon

```
"index": "src/index.html",
"main": "src/main.ts",
```

main.ts

```
import { platformBrowserDynamic }
   from '@angular/platform-browser-dynamic';
import { AppModule } from './app/app.module';
platformBrowserDynamic().bootstrapModule(AppModule)
```

index.html

```
<!doctype html>
<html lang="en">
<head>...</head>
<body>
This is index.html <br>
  <app-root></app-root>
</body>
</html>
```

app.module.ts

```
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 { }
```

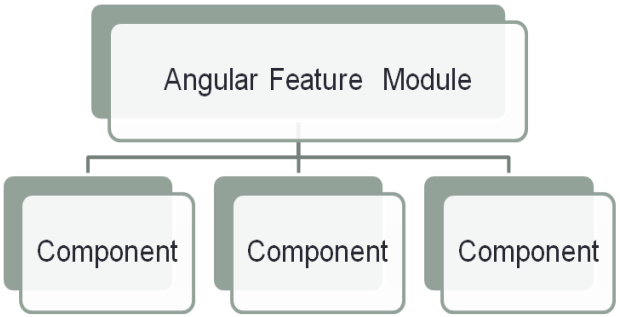
TS app.component.ts ×

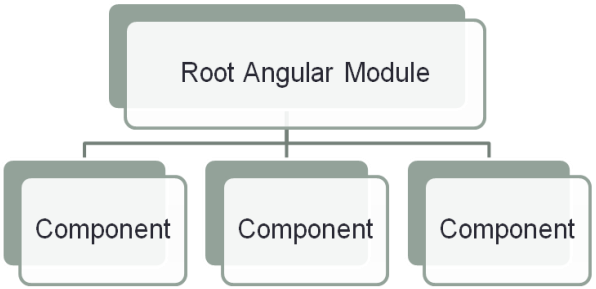
src > app > TS app.component.ts > ...

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

Architecture

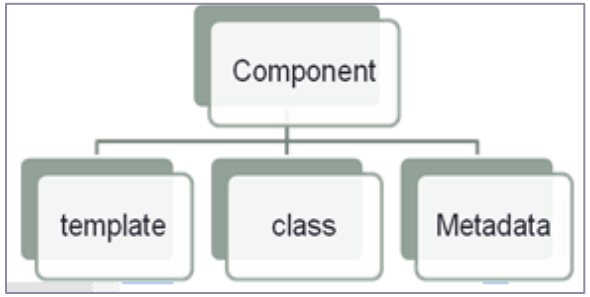
- Angular is designed to be modular
  - When you boot an angular app, you are *not booting component directly*, but rather you create an module that points to the component you want to load.
  - Modules are used in Angular to put logical boundaries in your application.
  - Angular App comprises of several **Modules**, a module typically exports something of a purpose.
    - Every Angular app has at least one Angular module class, the *root module*, conventionally named AppModule.
    - While the *root module* may be the only module in a small application, most apps have many more *feature modules*, each a cohesive block of code dedicated to an application domain, a workflow, or a closely related set of capabilities. An Angular module, whether a *root* or *feature*, is a class with an @NgModule decorator.
    - Angular 2 itself ships in large modules, some of them are '**angular/core**', '**angular/router**' etc
  - Typically a module may export a class which we may be imported in other modules.





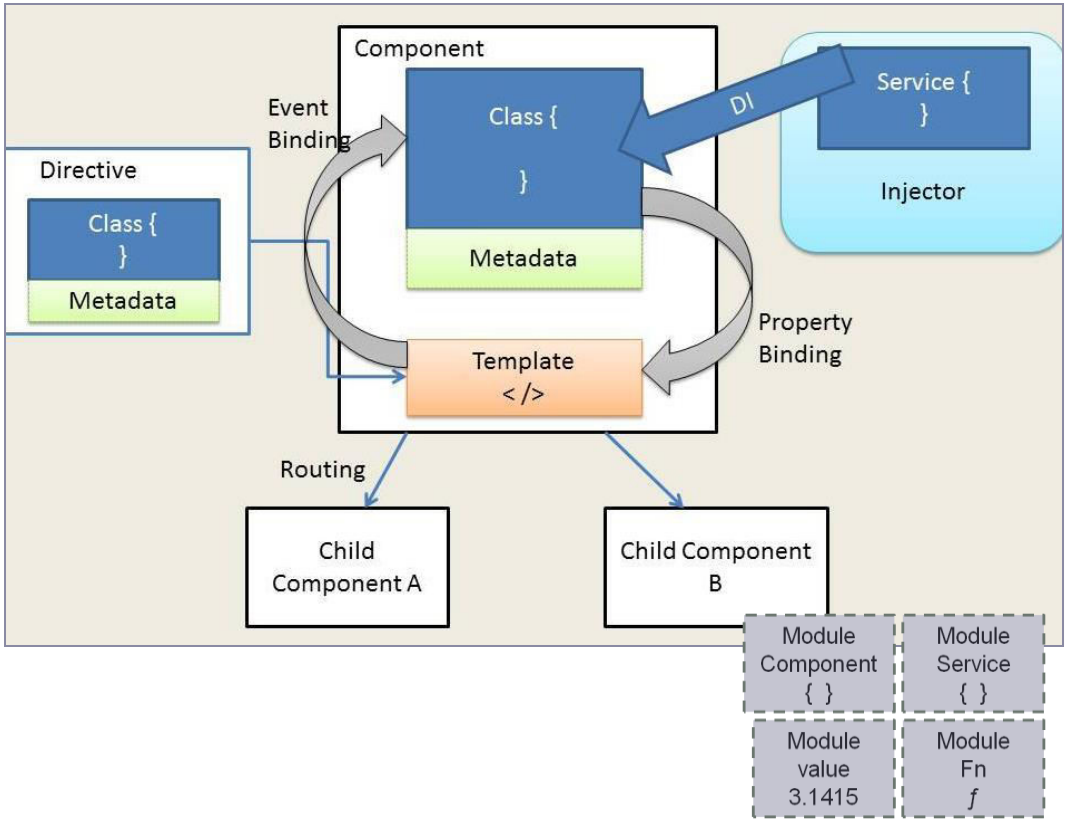
Architecture

- Angular is designed to be modular; An app comprises of several components, which are connected via routing or selectors
- Components may have templates attached to it which a)may display component properties and b)attach events to interact with the properties.
- A component may use a service, to access a particular feature or perform a very specific task; Services must be injected into components before they can be used from within the component (DI)



app.component.ts

```
import { Component }
  from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'app';
}
```



Modules

- A module is a class that is decorated with @NgModule decorator
  - Every application contains at least one module called root module, conventionally called as **AppModule**.
  - NgModule decorator has 4 keys that provides information about module:

```
//app.module.ts
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 { }
```

Decorators are functions that modify JavaScript classes. Angular has many decorators that attach metadata to classes so that it knows what those classes mean and how they should work

Imports : describes which *dependencies this module has.*

Declaration: specifies the components that are defined in this module; may be components, directives & pipes

Bootstrap: tells Angular that when this module is used to bootstrap an app, we need to load the AppComponent component as the top-level component

  - **providers** is used for Dependency Injection

Module

- A module in Angular is a class decorated with @NgModule decorator, whereas a module in JavaScript is a file and all objects defined in the file belongs to that module.
- JavaScript module exports some object using export keyword and they can be imported in other modules using import statement.

- Angular ships a couple of JavaScript modules each beginning with @angular prefix.
  - In order to use objects in those modules, we need to import them using import statement in JavaScript.
  - import { Component } from '@angular/core';
  - import { BrowserModule } from '@angular/platform-browser';

Component

- Components are major building blocks of an app and are composed together to build application.
  - A component controls a patch of screen called a view.
  - Each component is a logical boundary of functionality for the app
  - Every component is a **class** with **its own data and code**.
  - A component may depend on services that are injected using DI
  - The template, metadata, and component together describe a view.
  - Components are decorated with @Component decorator through which we specify template and selector (tag) related to component.

```
import { Component } from '@angular/core';
@Component({
   selector: 'my-app',
   template: `<h1>Hello {{name}}</h1>`,
  //templateUrl:'app/app.component.html'
})
export class AppComponent {
    name = 'Angular';
}
```

Imports the Component object from another module

Uses a decorator to add metadata to the component

Uses a template literal string to write inline HTML

Exports the component object, which was defined as a class

Component

- The template, metadata, and component together describe a view.
  - Metadata: decorates the class and extends the functionality of the class.
  - Template: defines the HTML view which is displayed in the application.
    - Template contains HTML, directives and data binding
  - Properties like **templateUrl** and **providers** can also be used.
    - Provider is an array of DI providers for service that the component requires.

Generating Components, Directives, Pipes and Services

- You can use the ng generate (or just ng g) command to generate Angular components

Scaffold	Usage
Component	ng g component my-new-component
Directive	ng g directive my-new-directive
Pipe	ng g pipe my-new-pipe
Service	ng g service my-new-service
Class	ng g class my-new-class
Guard	ng g guard my-new-guard
Interface	ng g interface my-new-interface
Enum	ng g enum my-new-enum
Module	ng g module my-module

  - angular-cli will add reference to components, directives and pipes automatically in the app.module.ts

Generating Components using CLI : Example-1

ng generate component my-new-component

ng g c my-new-component # using the alias

```
E:\FreeLanceTrg\Angular2\Demos\my-app>ng g c hello-world
  create src/app/hello-world/hello-world.component.html (30 bytes)
  create src/app/hello-world/hello-world.component.spec.ts (657 bytes)
  create src/app/hello-world/hello-world.component.ts (288 bytes)
  create src/app/hello-world/hello-world.component.css (0 bytes)
  update src/app/app.module.ts (416 bytes)
```

hello-world.component.ts

```
1  import { Component, OnInit } from '@angular/core';
2
3  @Component({
4    selector: 'app-hello-world',
5    templateUrl: './hello-world.component.html',
6    styleUrls: ['./hello-world.component.css']
7  })
8  export class HelloWorldComponent implements OnInit {
9    constructor() { }
10   ngOnInit() {}
11 }
```

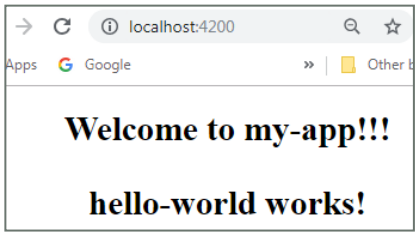
hello-world.component.html

```
1  <p>
2    hello-world works!
3  </p>
```

app.component.html | app.component.ts | app.module.ts | hello-world.c

```
<div style="text-align:center">
  <h1>
    Welcome to {{ title }}!!!
    <app-hello-world></app-hello-world>
  </h1>
```

```
import { HelloWorldComponent } from './hello-world/hello-world.component';

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



Example-2 : Adding data to component

- ng g c user-item

```
import { Component, OnInit } from '@angular/core';
@Component({
  selector: 'app-user-item',
  templateUrl: './user-item.component.html',
  styleUrls: ['./user-item.component.css']
})
export class UserItemComponent implements OnInit {
  uname:string;
  constructor() {
    this.uname = 'Shrilata';
  }

  ngOnInit() { }
}
```

output

Welcome to my-app!!!

hello-world works!

Hello Shrilata

user-item.component.html | app.component.html

```
<p>  Hello {{uname}}  </p>
```

app.component.html | user-item.component.html | app.component.ts

```
<h1>
    Welcome to {{ title }}!!!
    <app-hello-world></app-hello-world>
    <app-user-item></app-user-item>
</h1>
```

When we have a property on a component, we can show that value in our template by using **{{ }}** to display the value of the variable

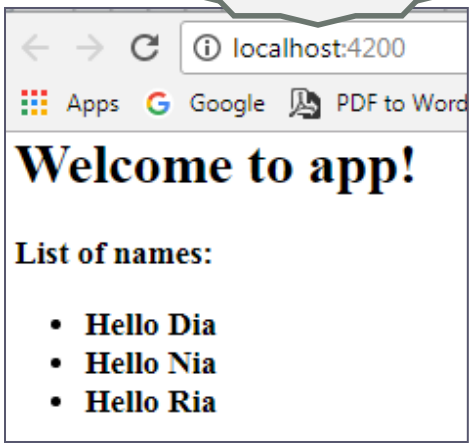
Example-3 : Working With Arrays

- ng g c user-list

user-list.component.ts

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

@Component({ ... })
export class UserListComponent implements OnInit {
   names:string[];

  constructor() {
    this.names=['Dia','Nia','Ria'];
  }

  ngOnInit() {  }
}
```

output



user-list.component.html

```
List of names:
<ul>
   <li *ngFor="let name of names">
       Hello {{name}}
   </li>
</ul>
```

app.component.html | user-item.component.html | app.component.ts

```
<h1>
    Welcome to {{ title }}!!!
    <app-hello-world></app-hello-world>
    <app-user-item></app-user-item>
    <app-user-list></app-user-list>
</h1>
```

ngFor : lets us iterate over a list of objects in our template. Ie we want to repeat the same markup for a collection of objects

Introducing @Input

- In the previous HTML, while iterating thru list of names, I am using a template – Hello {{uname}}. But this is a template that already exists in our previous component.
  - Instead of rendering each name within the UserListComponent, we ought to use UserItemComponent as a child component – ie we should let our UserItemComponent specify the template (and functionality) of each item in the list.

```
<parent-component>
  <child-component></child-component>
</parent-component>
```

- To do this, we need to do three things:
  - 1. Configure the UserListComponent to render to UserItemComponent (in the template)
  - 2. Configure the UserItemComponent to accept the name variable as an input and
  - 3. Configure the UserListComponent template to pass the name to the UserItemComponent.

@Input()

- @Input() : provides a way to *pass data into the child component.*

```
import { Component, OnInit, Input } from '@angular/core';
@Component({
  selector: 'app-user-item',
  templateUrl: './user-item.component.html',
  styleUrls: ['./user-item.component.css']
})
export class UserItemComponent implements OnInit {
  //uname:string;
  @Input() uname:string;

  constructor() {
    //this.uname = 'Shrilata';
  }
  ngOnInit() { }
}
```

List of Names:

- **Hello Dia**
- **Hello Nia**
- **Hello Ria**
- **Hello Mia**

```
List of Names:
<ul>
    <li *ngFor="let name of names">
      <app-user-item [uname]="name">
      </app-user-item>
    </li>
</ul>
```

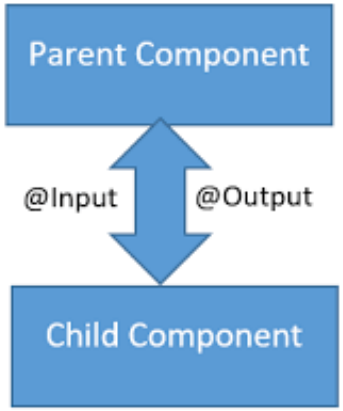
The value for item will come from the parent component

bind the property in the parent component's template

@Input() and @Output() properties

- @Input() and @Output() act as the API of the child component in that they allow the child to communicate with the parent.
  - Think of @Input() and @Output() like ports or doorways—@Input() is the doorway into the component allowing data to flow in while @Output() is the doorway out of the component, allowing the child component to send data out.

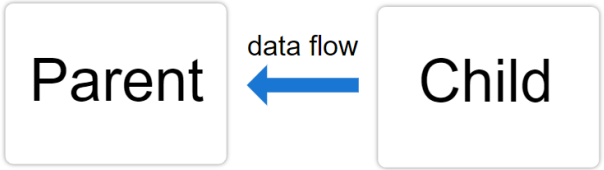
- Though @Input() and @Output() often appear together in apps, you can use them separately.
  - If the nested component is such that it only needs to send data to its parent, you wouldn't need an @Input(), only an @Output(). The reverse is also true in that if the child only needs to receive data from the parent, you'd only need @Input().



@Output

- @Output allows to pass data back from child to parent.
  - The child component then has to raise an event so the parent knows something has changed.
  - To raise an event, @Output() works hand in hand with EventEmitter, which is a class in @angular/core that you use to emit custom events.
  - Just like with @Input(), you can use @Output() on a property of the child component but its type should be EventEmitter.

```
@Output()
  public myevent = new EventEmitter();
```



- When you use @Output(), edit these parts of your app:
  - The child component class and template
  - The parent component class and template

Demo : @Output

- ng g c parent

parent.component.ts × | parent.component.html | TS child.cor

y-app > src > app > parent > TS parent.component.ts > ...

```
1   import { Component, OnInit } from '@angular/core';
2
3   @Component({
4     selector: 'app-parent',
5     templateUrl: './parent.component.html',
6     styleUrls: ['./parent.component.css']
7   })
8   export class ParentComponent {
9     pData = "data-from-parent";
10
11    public message;
12  }
```

parent.component.html × | TS parent.component.ts | TS child.component.ts | child.co

ny-app > src > app > parent > parent.component.html > ...

```
1
2   In Parent :-> Data From Child Component : {{message.cname}}
3
4   <app-child [parentData]="pData" (myevent)="message=$event"></app-child>
```

binding the parent's variable to the child's event

tells Angular to connect the event in the child, myevent, to the var "message" in the parent, and that the event that the child is notifying the parent about is to be the data that goes into message. In other words, this is where the actual hand off of data takes place. The $event contains the data that was emitted by the child.

Demo : @Output

- ng g c child

child.component.ts × | parent.component.html | TS parent.component.ts | child.component.h

y-app > src > app > child > TS child.component.ts > ...

```
1   import { Component, EventEmitter, Input, OnInit, Output } from '@angular/core';
2
3   @Component({
4     selector: 'app-child',
5     templateUrl: './child.component.html',
6     styleUrls: ['./child.component.css']
7   })
8   export class ChildComponent {
9     obj = { cname: "Shrilata-child-data",
10            caddress: "Aundh-child-data" }
11
12    @Input('parentData')
13    public strmsg;
14
15    @Output()
16    public myevent = new EventEmitter();
17
18    onclick() {
19      this.myevent.emit(this.obj);
20    }
21  }
```

myevent is a event that I will be emitting

In Parent :-> Data From Child Component :
In Child :-> Data from Parent :
click me

In Parent :-> Data From Child Component : Shrilata-child-data
In Child :-> Data from Parent : data-from-parent
click me

child.component.html × | TS child.component.ts | parent.component.

ny-app > src > app > child > child.component.html > ...

```
1   <div>In Child :->  Data from Parent :  {{strmsg}}</div>
2   <button (click)=onclick()>click me</button>
```

Example-4 : Working with class

- ng g c user-model

model.ts | user-model.component.html | user-model.component.ts | app.component.html | user-item.comp

```
export class Model {
   user; items;
   constructor() {
      this.user = "Shrilata";
      this.items = [new TodoItem("Buy Flowers", false),
                    new TodoItem("Get Shoes", false),
                    new TodoItem("Collect Tickets", false),
                    new TodoItem("Call Joe", false)]
   }
}
export class TodoItem {
   action; done;
   constructor(action, done) {
      this.action = action;
      this.done = done;
   }
}
```

user-model.component.html | user-model.component.ts

```
<h2 style="color:blue">
   {{ getName() }}'s To Do List
</h2>
```

output

Shrilata's To Do List

user-model.component.ts | model.ts | app.component.html | user-item.component.html

```
import { Component, OnInit } from '@angular/core';
import { Model } from "./model";

@Component({
export class UserModelComponent implements OnInit {
  model = new Model();
  getName() {
     return this.model.user;
  }
  ngOnInit() {  }
}
```

Example-4 : Working with class (extended)

user-model.component.html | user-model.component.ts | model.ts | app.component.html | user-item.comp

```
<h2 style="color:blue">
  {{ getName() }}'s To Do List
</h2>
<table border="1" style="border-collapse:collapsed">
   <tr><th>Sr no</th><th>Description</th></tr>
   <tr *ngFor="let item of getTodoItems(); let i = index">
      <td>{{ i + 1 }}</td>
      <td>{{ item.action }}</td>
   </tr>
</table>
```

user-model.component.ts | model.ts | app.component.ht

```
import { Component, OnInit } fro
import { Model } from "./model";

@Component({
  selector: 'app-user-model',
  templateUrl: './user-model.component.html',
  styleUrls: ['./user-model.component.css']
})
export class UserModelComponent implements OnInit {
    model = new Model();
    getName() {
     return this.model.user;
    }

    getTodoItems() {
     return this.model.items;
    }
 ngOnInit() {  }
}
```

output

Shrilata's To Do List

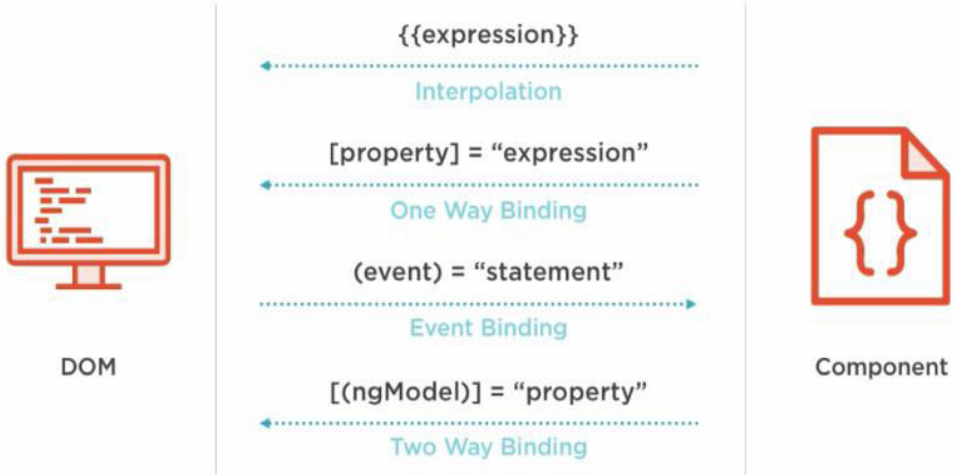
Sr no	Description
1	Buy Flowers
2	Get Shoes
3	Collect Tickets
4	Call Joe

Template in component class itself

```
import { Component  } from "@angular/core";
@Component({
    selector: 'my-App',
    template: `
                <div>
                <strong>{{firstname}}</strong>
                 <strong>{{lastname}}</strong>
                </div>  `
})
export class AppComponent  {
    firstname: string = "Sachin";
    lastname:string  = "Tendulkar"
}
```

Data binding

- Data binding : a mechanism for coordinating parts of a template with parts of a component.
  - Allows data from objects to be bound to HTML elements and vice-versa
  - Add binding markup to template HTML to tell Angular how to connect both sides.; Angular takes care of data binding.
- There are four forms of data binding syntax. Each form has a direction — to the DOM, from the DOM, or in both directions.
  - Interpolation : binds value of an expression to UI element in HTML
  - Property Binding : Enclosing property (attribute of HTML element) copies value to property.
  - Event Binding : handler to event.
  - Two-way binding : The ng-model is used to for two way data binding.



Interpolation

- {{ }} – is the interpolation operator.
  - Used whenever you need to communicate properties (variables/ objects/ arrays) from the component class to the template.
  - Interpolation binds the data one-way; format **{{ propertyName }}**
- Eg

```
//in component class - .ts file
export class HomeComponent {
      itemCount: number = 4;
}
```

```
//in .html file
<p>Your bucket list ({{ itemCount }})</p>
```

  - we've used interpolation {{ }} to show the **itemCount** property in the browser.
  - Another example :

```
template : `
   <h2>{{2+2}}</h2>
   <h2>{{"2+2"}}</h2>
   <h2>{{"Welcome : " + name}}</h2>
   <h2>{{name.length}}</h2>
   <h2>{{name.toUpperCase()}}</h2>
   <h2>{{greet()}}</h2> `
export class TestComponent{
   public name="Shrilata";
   greet(){
      return "Hello " + this.name;
   }
}
```

4

2+2

Welcome : Shrilata

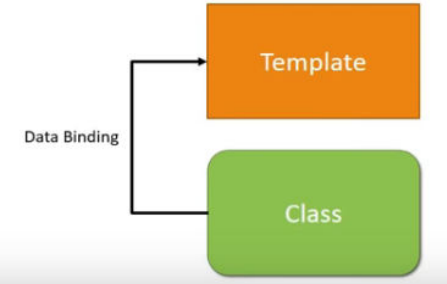
8

SHRILATA

Hello Shrilata

Property Binding

- Property binding is used to bind values to the **DOM properties** of the HTML elements.
  - It is one-way - communication is from the component class **to** the template.
- Ways to define a property binding in Angular:
  - <img [src]="angularLogo">
  - <img bind-src="angularLogo">




```
export class AppComponent {
    angularLogo = 'angular.png';
}
```

  - Depending on the values, it will change the existing behavior of the HTML element.
    Syntax - *[property] ='expression'*

```
Eg : status:boolean=false;
<button [disabled]="status">Click</button>
```

Simple demo : Property Binding

- Eg – 1 : Ng g c databind

```
template : `
<img src="someserver.com/someimage.gif">
`
```

- Now, in databind.component.ts, create a property:

```
class …{
   myImage;
    constructor(){
        this.myImage="someserver.com/someimage.gif";
   }
```

- And in html : <img [src]="myImage"> or <img bind-src="myImage">

```
//Eg – 2 : Another example:
/home.component.ts
btnText: string = 'Add an Item';

//home.component.html
<!– Change From: --> <input type="submit" value="Add Item">
<!-- To: --> <input type="submit" [value]="btnText">

You will see the new button value as defined in our component class
```

Example : Property Binding

```
export class Model {
   user;
   items;
   contact;
   constructor() {
      this.user = "Shrilata";
      this.contact = {email:"shrilata@gmail.com",
                      phone:9977886600
                     };
      this.items = [
          new TodoItem("Buy Flowers", false),
          new TodoItem("Get Shoes", false),
          new TodoItem("Collect Tickets", false),
          new TodoItem("Call Joe", false)]
   }
}
```

output

Name : Shrilata
Email : shrilata@gmail.com
Phone : 9977886600
Buy Flowers

```
export class UserModelComponent {
    model = new Model();
    contact;
    constructor(){
        this.contact = this.model.contact;
    }
    getName() {
     return this.model.user;
    }
```

user-model.component.html | model.ts | app.component.html | user-item.component.html | app.com

```
Name  : <input type="text" [value]="getName()"><br>
Email : <input type="text" [value]="contact.email"><br>
Phone : <input type="text" [value]="contact.phone"><br>

<input type="button" [value]="getTodoItems()[0].action">
```

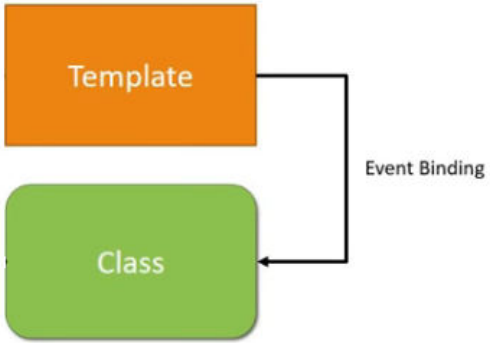
One can send values or even execute methods

user-model component

Event Binding

- Event binding is **one-way** data binding which sends the value from the view to the component
- When a specific DOM event happens (eg click, change, keyup), calls the specified method in the component.

```
event-eg.component.ts | first.js | RunServer.js | user-model.component.ts | model.ts

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

@Component({
  selector: 'event-eg',
  template: `
    <button (click)="handleClick()">Click here</bu
  `
})
export class EventEgComponent {
  handleClick(){
   console.log("button clicked!");
  }
}
```



- Alternate syntax : <button on-click=" handleClick()">Click me</button>

Event Binding

- Setting properties on event:

```
@Component({
  selector: 'event-eg',
  template: `
    <button (click)="handleClick()">Click here
    {{greeting}}
  `
})
export class EventEgComponent {
  greeting="";
  handleClick(){
   this.greeting = "Hello World";
  }
}
```

Click here Hello World

Event Binding

- Finding info about event itself:
  - Set a parameter in event - $event – it's a special var for Angular – gives all info about the DOM event that was raised

```
@Component({
  selector: 'event-eg',
  template: `
    <button (click)="handleClick($event)">
       Click here
    </button>
    {{greeting}}
  `
})
export class EventEgComponent {
  greeting="";
  handleClick(event){
   this.greeting = "Hello! Event type : " + event.type;
  }
}
```

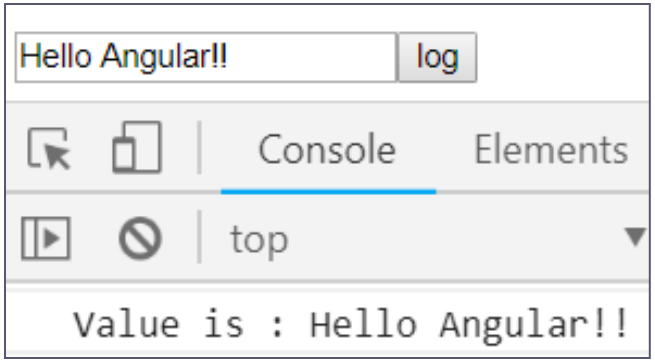
Click here Hello! Event type : click

Event Binding

- Template reference variables
  - When there is user interaction, we may want some data to flow from view to class to perform an operation. Eg : Use the value in a text field to perform some validations.
  - A template reference variable is often a reference to a DOM element within a template. It can also be a reference to an Angular component or directive or a web component.
  - Use the hash symbol (#) to declare a reference variable.
  - You can refer to a template reference variable anywhere in the template. The myInput variable declared on this <input> is consumed in a <button> elsewhere in the template

event-eg.component.ts | first.js | RunServer.js | user-model.component.ts | model.ts

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

@Component({
  selector: 'event-eg',
  template: `
    <input #myInput type="text">
    <button (click)="log(myInput.value)">log</buttc
  `
})
export class EventEgComponent {
  log(myvalue){
     console.log("Value is : " + myvalue);
  }
}
```



Event Binding

>ng cg c user-manage

user-manage.component.ts | user-manage.component.html | app.component.html | useradd.c

```
export class UserManageComponent implements OnInit{
  titlelabel:string="Cricket Players";
  names = ["Virat","Yuvraj","Dhoni"];
  addName(name){
      this.names.push(name);
  }
  delName(name){
      for(var i in this.names){
        if(this.names[i]==name){
            this.names.splice(parseInt(i),1);
            console.log("found"); break;
        }
      }}
  delAllNames(){
      this.names.splice(0,this.names.length);
  }
```

Cricket Players

- Virat
- Yuvraj
- Dhoni

Player Name:

add player | del player | del players

user-manage.component.html | app.component.html | useradd.component.ts

```
<h2>{{titlelabel}}</h2>
<h4>
  <ul><li *ngFor="let n of names">{{n}}</li></ul>
</h4>
Player Name:<input name="t1" #player><br>
<input type="submit" value="add player" (click)="addName(player.value)">
<input type="submit" value="del player" (click)="delName(player.value)">
<input type="submit" value="del players" (click)="delAllNames()">
```

Event Binding

user-manage.component.ts | user-manage.component.html | app.component.html | useradd.c

```
export class UserManageComponent implements OnInit{

  showEvent(event){
      console.log(event.target.value);
  }
  hide(event){
     event.target.style.visibility = 'hidden';
  }
  show(event){
     event.target.style.visibility = 'visible';
  }
```

user-manage.component.html | app.component.html | useradd.component.ts | user-manage.component.ts

```
<input type="text" value="Shrilata T" (click)="showEvent($event)">
<input type="text" (keyup)="showEvent($event)">
<input type="button" (click)="showEvent($event)">
<h1><div (mouseover)="hide($event)"
         (mouseout)="show($event)">Div element</div></h1>
```

Displays on console every key entered

Displays “a Button”

Displays Shrilata T

Alternatively hides and shows Div element

Two-way data binding

- Angular 2 provides the ngModel directive which combines the square brackets of property binding with the parentheses of event binding in a single notation.
  - **Syntax :** <input [(ngModel)] =“data1”>

```
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule } from "@angular/forms"
  ...
  @NgModule({
    imports: [BrowserModule, FormsModule],
      ...
})
```

Note *: When you are using NgModal directive, make sure you have imported the Form Module in app.module.ts file*

twoway-bind.component.ts | twoway-bind.component.html | app.component.html | useradd.comp

```
export class TwowayBindComponent implements OnInit {
  username:string="";
```

twoway-bind.component.html | app.component.html | useradd.component.ts

```
Enter name : <input [(ngModel)]="username">
Welcome user {{username}}
```

Enter name : Shrilata
Welcome user Shrilata

Directives

- A directive transforms DOM according to instructions given.
  - You can also create your own custom directive
  - There are three kinds of directives in Angular:

Component	Attribute Directive	Structural Directives
Used to specify the template/ html for the Dom Layout	Used to alter the appearance/behaviour of the html element in the Dom Layout	Alter layout by adding, removing, and replacing elements in the DOM.
Built in **@component**	**Built in** **ngStyle, ngClass, ngModel**	**Built in** ***ngIf,*ngFor,*ngSwitch**

  - In Angular, components are just one type of directives.
  - Attribute directive: in templates they look like regular HTML attributes, hence the name.

Structural directive - *ngFor

- NgFor is a directive that iterates over collection of data.
  - NgFor uses following local variables.
    - **index**: Provides the index for current loop iteration. Index starts from 0.
      first: Provides Boolean value. It returns true if the element is first in the iteration otherwise false.
    - **last**: Provides Boolean value. It returns true if the element is last in the iteration otherwise false.
    - **even**: Provides Boolean value. For every index of elements in the iteration, if even then returns true otherwise false.
    - **odd**: Provides Boolean value. For every index of elements in the iteration, if odd then returns true otherwise false.

```
<div *ngFor="let user of users; let i = index">
    Row {{i}} : {{user.name}} - {{user.age}}
</div>
```

```
export class User {
  constructor(public name: string,
              public age: number) {
  }
}
```

```
export class StructDirComponent{
    users = [
      new User('Mahesh', 20),
      new User('Krishna', 22),
      new User('Narendra', 30)
    ];
```

```
<ul>
 <li *ngFor="let user of users">
  {{user.name}} - {{user.age}}
 </li>
</ul>

<b>index variable demo </b>
 <p *ngFor="let user of users; let i = index">
  Row {{i}} : Name: {{user.name}}
 </p>

 <b>first and last variable demo </b>
 <div *ngFor="let user of users; let i = index; let f=first; let l=last;">
  Row {{i}} : Name: {{user.name}}, is first row: {{f}}, is last row: {{l}}
 </div>

 <b>even and odd variable demo </b>
 <div *ngFor="let user of users; let i = index; let e=even; let o=odd;">
  Row {{i}} : Name: {{user.name}}, is even row: {{e}}, is odd row: {{o}}
 </div>
```

- Mahesh - 20
- Krishna - 22
- Narendra - 30

Example-1

index variable demo

Row 0 : Name: Mahesh

Row 1 : Name: Krishna

Row 2 : Name: Narendra

first and last variable demo
Row 0 : Name: Mahesh, is first row: true, is last row: false
Row 1 : Name: Krishna, is first row: false, is last row: false
Row 2 : Name: Narendra, is first row: false, is last row: true
even and odd variable demo
Row 0 : Name: Mahesh, is even row: true, is odd row: false
Row 1 : Name: Krishna, is even row: false, is odd row: true
Row 2 : Name: Narendra, is even row: true, is odd row: false

ngfor-demo.component.ts | ngfor-demo.component.html | model.ts | app.component.h

```
import { Component, OnInit } from '@angular/core';
import { Model } from "./model";
@Component({...})
export class NgforDemoComponent implements OnInit {
   model = new Model();
   person;
  constructor() {
     this.person = this.model.person;
  }
  ngOnInit() { }
}
```

ngfor-demo.component.html | model.ts | ngfor-demo.component.ts | app.component

```
<h4>List of people</h4>
<table border="1">
   <tr><th>Name</th><th>Age</th><th>City</th></tr>
   <tr *ngFor="let p of person">
    <td>{{ p.name }}</td>
    <td>{{ p.age }}</td>
    <td>{{ p.city }}</td>
   </tr>
</table>
```

model.ts | ngfor-demo.component.ts | app.component.html | useradd.component.ts | user-mo

```
export class Model {
   person;
  constructor() {
     this.person = [
       { name: 'Anderson', age: 35, city: 'Sao Paulo' },
       { name: 'John', age: 12, city: 'Miami' },
       { name: 'Peter', age: 22, city: 'New York' }
     ];
   }
}
```

Example-2

List of people

Name	Age	City
Anderson	35	Sao Paulo
John	12	Miami
Peter	22	New York

Structural directive - NgIf

- Use ngIf directive when you want to display or hide an element based on a condition
  - If the result of the expression returns a false value, the element will be removed from the DOM. Examples:
    - <div *ngIf="false"> <!-- never displayed -->
    - <div *ngIf="a > b"> <!-- displayed if a is more than b -->
    - <div *ngIf="str == 'yes'"> <!-- displayed if str is the string "yes" →
    - <div *ngIf="myFunc()"> <!-- displayed if myFunc returns truthy -->

```
<b>NgIf with HTML Elements </b><br/>
<p *ngIf="isValid"> Data is valid. </p>
<p *ngIf="!isValid"> Data is not valid. </p>

<div *ngFor="let id of ids">
  Id is {{id}}
  <div *ngIf="id%2 == 0">
    <div [ngClass]="'one'">Even Number</div>
  </div>
  <div *ngIf="id%2 == 1">
    <div [ngClass]="'two'">Odd Number</div>
  </div>
</div>
<div *ngIf="user1"> Id:{{user1.id}} Name: {{user1.name}}
</div>

<div *ngIf="user2"> Id:{{user2.id}} Name: {{user2.name}}
</div>
```

```
export class StructDirComponent{
    isValid = true;
    ids = [1,2,3,4];
    user1 = new User("Shrilata",20);
    user2:User;

}
```

Data is valid.

Id is 1
Odd Number
Id is 2
Even Number
Id is 3
Odd Number
Id is 4
Even Number

Id: Name: Shrilata

Structural directive - *ngSwitch

```
export class StructuralDirectiveComponent  {
    data2=0;
}
```

```
Input using ngModel : <input [(ngModel)] ='data2'>
<div [ngSwitch]="data2">
<p *ngSwitchCase="1">You selected One</p>
<p *ngSwitchCase="2">You selected Two</p>
<p *ngSwitchCase="3">You selected Three</p>
<p *ngSwitchDefault>Sorry Invalid selection!!</p>
</div>
```

Input using ngModel : 1

You selected One

- Another example:

```
  template: `

    <div [ngSwitch]="color">
      <div *ngSwitchCase="'red'">You picked red color</div>
      <div *ngSwitchCase="'blue'">You picked blue color</div>
      <div *ngSwitchCase="'green'">You picked green color</div>
    </div>

  `,
  styles: []
})
export class TestComponent implements OnInit {

  public color = "red";
```

Attribute directive - ngStyle

- NgStyle is used to set multiple inline styles for html element.
  - You can set a given DOM element CSS properties from Angular expressions.
  - The simplest way to use this directive is by doing [style.<cssproperty>]="value“

```
<div [style.background-color]="'yellow'">
  Uses fixed yellow background
</div>
```

This uses the NgStyle directive to set the background-color CSS property to the literal string 'yellow'.

  - Another way to set fixed values is by using the NgStyle attribute and using key value pairs for each property you want to set, like this:

```
<div [ngStyle]="{color: 'white', 'background-color': 'blue'}">
  Uses fixed white text on blue background
</div>
```

Attribute directive - ngStyle

- But the real power of the NgStyle directive comes with using dynamic values

attr-directive.component.html

```
<button style='color:blue' [ngStyle]="ApplyStyles()">Style Applied</button>
```

```
export class AttrDirectiveComponent  {
    isBold: boolean = true;
    fontSize: number = 30;
    isItalic: boolean = true;

    ApplyStyles() {
        let styles = {
            'font-weight': this.isBold ? 'bold' : 'normal',
            'font-style': this.isItalic ? 'italic' : 'normal',
            'font-size.px': this.fontSize
        };

        return styles;
    }
```

Style Applied

Attribute directive - ngClass

- NgClass allows you to dynamically set and change the CSS classes for a given DOM element.
  - The first way to use this directive is by passing in an object literal.
  - The object must have the keys as the class names and the values should be a truthy/falsy value to indicate whether the class should be applied or not.

```
.bordered {
  border: 1px dashed black;
  background-color: #eee; }
```

Styles.css

abc.component.html

```
<div [ngClass]="{bordered: false}">This is never bordered</div>
<div [ngClass]="{bordered: true}">This is always bordered</div>
```

This is never bordered

This is always bordered

output

  - But, it's a lot more useful to use the NgClass directive to make class assignments dynamic.

Attribute directive - ngClass

```
@Component({
  selector: 'app-attr-directive',
  template: ` <br>
    <button class='colorClass' [ngClass]='applyClasses()'>
        Style Applied Using Class</button>
    `,
  styles: [`
    .boldClass{
       font-weight:bold;
       font-size : 30px;
    }
    .italicsClass{font-style:italic}
    .colorClass{color:grey}
  `]
})
export class AttrDirectiveComponent  {
    applyBoldClass: boolean = true;
    applyItalicsClass: boolean = true;
    applyClasses() {
        let classes = {
            boldClass: this.applyBoldClass,
            italicsClass: this.applyItalicsClass
        };
        return classes;
    }
```

```
<div *ngFor="let id of ids">
  Id is {{id}}
  <div *ngIf="id%2 == 0">
    <div [ngClass]="'one'">Even Number</div>
  </div>
  <div *ngIf="id%2 == 1">
    <div [ngClass]="'two'">Odd Number</div>
  </div>
</div>
```

irective.component.html | structural-directive.component.css

```
.one{color:blue}
.two{color:red}
```

Style Applied Using Class

Id is 1
Odd Number
Id is 2
Even Number
Id is 3
Odd Number
Id is 4
Even Number

Pipe

- Pipes are used to format or transform how data is displayed in our templates.
  - They don't actually change the underlying data, but they do change how it is displayed in the template in which the pipe is applied
  - Pipes are classes that are used to prepare data for display to the user.
  - Syntax : myValue | myPipe:param1:param2 | mySecondPipe:param1
  - Angular 4 provides some built-in pipes:
    - Lowercasepipe
    - Uppercasepipe
    - Titlecasepipe
    - Datepipe
    - Currencypipe
    - Jsonpipe
    - Percentpipe
    - Decimalpipe
    - Slicepipe

```
//component
export class AppComponent {
  title = 'Angular 6 Project';
}
```

```
<!– template →
title : {{title | lowercase}}
title : {{title | uppercase}}
```

```
title : angular 6 project
title : ANGULAR 6 PROJECT
```

Pipe

- A pipe can accept any number of optional parameters to better its output.
  - To add parameters to a pipe, suffix with a colon followed by parameter value
  - If the pipe accepts multiple parameters, separate the values with colons
- Numberpipe : formats number values
  - The number pipe accepts a single argument that specifies the number of digits that are included in the formatted result.
  - Argument format: "<minIntegerDigits>.<minFactionDigits>-<maxFractionDigits>"
    - Eg "3.2-2", which specifies that at least three digits should be used to display the integer portion of the number and that two fractional digits should always be used
    - minIntegerDigits : default value is 1.
    - minFractionDigits : default value is 0; maxFractionDigits : default value is 3.

```
{{200 | number:'2.5-5'}}
{{3.14 | number:'.5-5'}}
{{3.1415926 | number}}
{{200 | number:'5.5-5'}}
{{200 | number}}
{{8.7844 | number:'1.2-2'}}
```

```
200.00000
3.14000
3.142
00,200.00000
200
8.78
```

Pipe

- Currencypipe formats number values that represent monetary amounts
  - Syntax - value | currency
  - By default, the currency is USD but we can change it by specifying the country name.
  - Eg - value | currency:'GBP‘

```
{{0.23|currency}}
{{0.23|currency:'USD’}}
{{0.23|currency:'GBP'}}
{{0.23|currency:'INR'}}
{{100.2345|currency:"USD":"symbol":"2.2-2"}}
```

$0.23
$0.23
£0.23
₹0.23
$100.23

- Percentpipe formats number values as percentages.

```
{{0.1 | percent}}
{{0.2 | percent}}
{{0.5 | percent}}
{{1.5 | percent}}
```

10%
20%
50%
150%

Pipe

- Datepipe performs location-sensitive formatting of dates.
- See angular's official doc for more : https://angular.io/api/common/DatePipe

```
@Component({
  selector: 'app-pipe-demo',
  template: `
 {{today}}            <br>
 {{today  | date}}   <br>
 {{today  | date:'short'}}
 {{today  | date:'dd/MM/yyyy'}}
 {{today  | date:'d/M/y'}}
 {{now}} ` })
export class PipeDemoComponent
   today=1491009511139;
   now = new Date();
```

1491009511139
Apr 1, 2017
4/1/17, 6:48 AM
01/04/2017
1/4/2017
Wed Oct 31 2018 23:29:26 GMT+0530

- Jsonpipe creates a JSON representation of a data value. No arguments are accepted by this pipe

```
@Component({
  selector: 'app-pipe-demo',
  template: `
   Person : {{person}}
   Person : {{person | json}}
   names : {{names}}
   names : {{names  | json}}
 `
 })
export class PipeDemoComponent
  names: string[] = ['paul', 'david','joe'];
  person={
      name:"Joy",
      age:23,
      address:{city:"Pune", st:"Mah"}
  };
```

```
Person : [object Object]
Person : { "name": "Joy", "age": 23, "address": { "city": "Pune", "st": "Mah" } }
names : paul,david,joe
names : [ "paul", "david", "joe" ]
```

Pipe

- Slicepipe operates on an array or string and returns a subset of the elements or characters it contains
  - The objects /characters selected by the pipe are specified using two arguments:
    - start : mandatory; start index for items to be included in the result
    - end : optional; specifies how many items from the start index should be included in the result

```
@Component({
  selector: 'app-pipe-demo',
  template: `
  {{name}}
  sliced to {{name | slice:0:3}}
  <ul>
    <li *ngFor="let n of names | slice:1:4">{{n}}</li>
  </ul> `  })
export class PipeDemoComponent
  names = ['paul', 'david', 'harry', 'nate', 'joe'];
  name = "Anuradha Singh";
```

Anuradha Singh
sliced to Anu

- david
- harry
- nate

- Another example

```
export class PipeDemoComponent implements OnInit {

  today = new Date();
  name = "Anu Singh";
  pi = Math.PI;
  discount:number = 1.09876;
  names: string[] = ['paul', 'david', 'harry', 'nate', 'joe']
  person={
      name:"Anjali",
      age:23,
      address:{city:"Pune", state:"Mah"},
      languages:["Eng","Hin","Marathi"]
  };
  value;
  change(value) { this.value = value; }
  constructor() { }
  ngOnInit() { }
}
```

```
<h2>Pipe Demo</h2>
Name : {{name | lowercase}}
Name : {{name | uppercase}}
Today is : {{today}}
Today is : {{today | date}}
Today is : {{today | date:"dd/MM/yyyy"}}
PI : {{pi}} transformed : {{pi | number:'.5-5'}}
PI : {{pi |  number:'2.10-10'}}
Currency : {{discount}}
Currency : {{discount | currency:'USD':true}}
Currency : {{discount | currency:'INR':false:'4.2-2'}}
Person : {{person}}
Person : {{person | json}}
Discount percentage : {{discount | percent:'3.2-2'}}
{{name}} : sliced to {{name | slice:0:3}}
<h2>Names</h2>
<ul>
  <li *ngFor="let n of names | slice:1:4">{{n}}</li>
</ul>
<div>
 Name: <input #user (keyup)="change(user.value)">
<span style="color:blue">Lowercase: {{value | lowercase}}
<span style="color:red">Uppercase: {{value | uppercase}}
</div>
```

Pipe Demo

Name : anu singh
Name : ANU SINGH
Today is : Wed Oct 31 2018 13:44:10 GMT+0530 (India Standard Time)
Today is : Oct 31, 2018
Today is : 31/10/2018
PI : 3.141592653589793 transformed : 3.14159
PI : 03.1415926536
Currency : 1.09876
Currency : $1.10
Currency : INR0,001.10
Person : [object Object]
Person : { "name": "Anjali", "age": 23, "address": { "city": "Pune", "state": "Mah" }, "languages": ["Eng", "Hin", "Marathi"] }
Discount percentage : 109.88%
Anu Singh : sliced to Anu

Names

- david
- harry
- nate

Name: shrilata
Lowercase: shrilata
Uppercase: SHRILATA

Service

- Services are objects that define functionality required by other building blocks such as components or directives.
  - Angular comes with a number of built in services; however you can also easily create your own.
  - Shared code across your application is almost always best placed inside of a service.
  - Components are for displaying the UI, and services are meant to help manage data or other reusable snippets of logic
  - Services can provide cross-cutting functionality for your application, such as logging, authentication, and messaging.
  - They can contain code to request and store data from external servers
  - For example, applications that require a user to log in will need to have a service to help manage the user's state.
- **Services can depend on other services**.
  - For example, we could have a CustomerService that depends on the Logger service, and also uses BackendService to get customers. That service in turn might depend on the HttpClient service to fetch customers asynchronously from a server.
- Services can be injected not just into components, but also directives and pipes.

Example-1 : Services

- ng g s crypto

onent.ts | app.module.ts | templateform-demo.component.html | templateform-demo.component

```ts
import { CryptoService } from './crypto.service';

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

onent.ts | crypto.service.ts | app.module.ts | templateform-demo.component.html

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

@Injectable({
  providedIn: 'root'
})
export class CryptoService {

   coins= [
    {id: 1, name: 'BTC'},
    {id: 2, name: 'XRP'}
  ];
  getMyItems(){
    return this.coins;
  }
}
```

onent.ts | testcrypto.component.ts | crypto.service.ts | app.module.ts | templateform-demo.compo

```ts
import { Component } from '@angular/core';
import { CryptoService } from '../crypto.service';

@Component({  ...  })
export class TestcryptoComponent {
  coins = [];
  constructor(private cryptoservice: CryptoService) {
      this.coins = cryptoservice.getMyItems();
  }
}
```

onent.ts | testcrypto.component.html | testcrypto.component.ts

```html
<table border="1">
  <tr>
    <th>ID</th>
    <th>Name</th>
  </tr>
  <tr *ngFor="let coin of coins">
    <td>{{coin.id}}</td>
    <td>{{coin.name}}</td>
  </tr>
</table>
```

ID	Name
1	BTC
2	XRP

Example-2 : Services

```
export class Customer {
    id: number;
    firstname: string;
    lastname: string;
    age: number
}
export const customers: Customer[] = [
            {id: 1, firstname: 'Mary', lastname: 'Taylor', age: 24},
            {id: 2, firstname: 'Peter', lastname: 'Smith', age: 18},
            {id: 3, firstname: 'Lauren', lastname: 'Taylor', age: 31}];
```

```
import { Injectable } from '@angular/core';
import { customers } from './testcustomer/customer.model';
import { Customer } from './testcustomer/customer.model';

@Injectable({ providedIn: 'root'})
export class CustomerService {
     getCustomers(): Customer[] {
          return customers;
}}
```

Example-2 : Services

onent.ts | testcustomer.component.html | testcustomer.component.ts | customer.service.ts | customer.model.t

```
import { Component, OnInit } from '@angular/core';
import { CustomerService } from '../customer.service';
import { Customer } from './customer.model';

@Component({
  selector: 'testcustomer',
  templateUrl: './testcustomer.component.html'
})
export class TestcustomerComponent implements OnInit{
  customers: Customer[];
  constructor(private customerService: CustomerService) { }
  ngOnInit() {
    this.customers = this.customerService.getCustomers();
  }
}
```

onent.ts | testcustomer.component.html | testcustomer.component.ts | custo

```
<h1>Customer List</h1>
<ul>
  <li *ngFor="let customer of customers">
    {{ customer | json }}
  </li>
</ul>
```

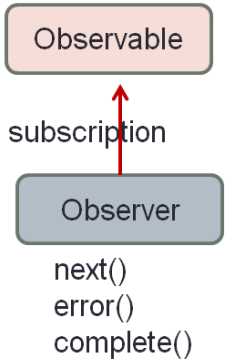
Customer List

- { "id": 1, "firstname": "Mary", "lastname": "Taylor", "age": 24 }
- { "id": 2, "firstname": "Peter", "lastname": "Smith", "age": 18 }
- { "id": 3, "firstname": "Lauren", "lastname": "Taylor", "age": 31 }

Observable Data

- The CustomerService.getCustomers() method is *synchronous*
  - The TestCustomerComponent consumes the getCustomers() result as if customers could be fetched synchronously.
  - this.customers = this.customerService.getCustomers();
  - The service currently returns *mock customers*. But in real life, the app will fetch customers from a remote server, which is an inherently *asynchronous* operation.
  - The CustomerService must wait for the server to respond, getCustomers() cannot return immediately with customer data, & the browser might block while the service waits.
  - Thus, CustomerService.getCustomers() must be asy*nchronous*
  - It can take a callback. It could return a Promise. It could return an Observable.
    - Lets see how the method will return an Observable in part because it will eventually use the Angular HttpClient.getmethod to fetch the customers and HttpClient.get() returns an Observable.

- **Observable** is basically a wrapper around a data source.
  - It is a sequence of items that arrive asynchronously over time.
- RxJs : is a library that enables us to work with observables.



Observable CustomerService

onent.ts | customer.service.ts | testcustomer.component.html | testcustomer.c

```
import { Observable, of } from 'rxjs';

@Injectable({
  providedIn: 'root'
})
export class CustomerService {
  /*getCustomers(): Customer[] {
    return customers;
  }*/
  getCustomers(): Observable<Customer[]> {
    return of(customers);
  }
}
```

nent.ts | testcustomer.component.ts | testcustomer.component.html | customer.model.ts | app.module.ts

```
export class TestcustomerComponent implements OnInit{
  customers: Customer[];
  constructor(private customerService: CustomerService) { }
  /*ngOnInit() {
    this.customers = this.customerService.getCustomers();
  }*/
   ngOnInit() {
    this.customerService.getCustomers()
    .subscribe(customers => this.customers = customers);
  }
```

Observable.subscribe() is the critical difference.

Http mechanism

Http mechanism

- Http GET request from EmpService
- Receive the observable and cast it into an Employee array
- Subscribe to the observable from Emplist and EmpDetail component
- Assign the emp array to a local variable
- Most front-end applications communicate with backend services over the HTTP
  - The Angular HttpClient offers a simplified client HTTP API for applications that rests on the XMLHttpRequest interface exposed by browsers.
  - Before you can use HttpClient, you need to import the Angular HttpClientModule.
  - Most apps do so in the root AppModule.

```
import { NgModule }        from '@angular/core';
import { BrowserModule }   from '@angular/platform-browser';
import { HttpClientModule } from '@angular/common/http';
@NgModule({
  imports: [ BrowserModule,
             // import HttpClientModule after BrowserModule.
             HttpClientModule ],
  providers: [EmpHttpService],
  declarations: [ AppComponent ],
  bootstrap: [ AppComponent ]
})
export class AppModule {}
```

Http mechanism

- Having imported HttpClientModule into the AppModule, you can inject the HttpClient into an application class.
- Example:

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

@Injectable()
export class ConfigService {
  constructor(private http: HttpClient) { }
}
```

onent.ts | emp-http.service.ts | employee.ts | app.module.t

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable} from 'rxjs/Observable';
import { IEmployee } from './employee';

@Injectable({
  providedIn: 'root'
})
export class EmpHttpService {
  //baseUrl: string = 'http://localhost/phpdemo/Angular-JSON.php';
  //if you dont have a running web server, you can also locally
  //create a json file and point to it via URL
  baseUrl: string = '/assets/data/emp.json';

  constructor(private http: HttpClient) { }

  getEmployees() : Observable<IEmployee[]> {
    return this.http.get<IEmployee[]>(this.baseUrl);
  }
}
```

onent.ts | employee.ts | app.module.ts | emp-htt

```
export interface IEmployee{
    id:number,
    name:string,
    sal:number
}
```

ng g s EmpHttp // creates emp-http.Service.ts

Example 1 : Working with HttpClient Module

- The data file:

onent.ts | Angular-JSON.php | emp.json | emp-http.service.ts | employe

```php
<?php
$json = file_get_contents('./emp.json');
echo($json);
?>
```

onent.ts | emp.json | emp-http.service.ts | employee.ts | app.module.ts | customer.s

```json
[
    { "id": 11, "name": "Anita", "sal":23000},
    { "id": 12, "name": "Puneet", "sal":40000},
    { "id": 13, "name": "Vineet", "sal":45000},
    { "id": 14, "name": "Sarita", "sal":20000},
    { "id": 15, "name": "Mayuri", "sal":80000}
]
```

onent.ts | emp-list.component.html | Angular-JSON.php | emp.json

```html
<h1> List of employees</h1>
<table border="1">
  <tr><th>Emp Id</th><th>Emp Name</th>
      <th>Emp Salary</th></tr>
  <tr *ngFor="let c of employees">
    <td>{{c.id}}</td>
    <td>{{c.name}}</td>
    <td>{{c.sal}}</td>
  </tr>
</table>
```

Emp Id	Emp Name	Emp Salary
11	Anita	23000
12	Puneet	40000
13	Vineet	45000
14	Sarita	20000
15	Dhara	80000

- Create component : ng g c EmpList

onent.ts | emp-list.component.ts | Angular-JSON.php | emp.json | emp-http.service.ts | employee.ts

```typescript
import { Component, OnInit } from '@angular/core';
import { EmpHttpService } from '../emp-http.service';
import { IEmployee } from '../employee';

@Component({ ... })
export class EmpListComponent {// implements OnInit {
  public employees:IEmployee[] = [];
  constructor(private empService : EmpHttpService) { }

  ngOnInit() {
      //this.employees = this.empService.getEmployees();
      this.empService.getEmployees()
      .subscribe(data => this.employees = data);
  }
}
```

Routers

- Routing basically means navigating between pages.
  - Here the pages that we are referring to will be in the form of components.
  - A route is responsible for indicating which component in your application needs to be activated.
  - The active route is determined by whatever path is present in the URL.

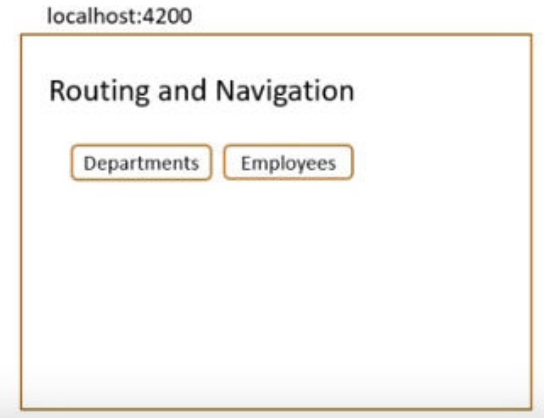
- For instance, if we visit the / path of a website, we may be visiting the **home route of that website.** Or if we visit /about we want to render the “about page”, and so on.
  - For example, imagine we are writing an inventory application. When we first visit the application, we might see a search form where we can enter a search term and get a list of products that match that term.
  - After that, we might click a given product to visit that product’s details page.

Angular Router

- Angular Router is an official Angular routing library, written and maintained by the Angular Core Team.
  - It's a JavaScript router implementation that's designed to work with Angular and is packaged as @angular/router.
- Angular Router does the following:
  - it activates all required Angular components to compose a page when a user navigates to a certain URL
  - it lets users navigate from one page to another without page reload
  - it updates the browser's history so the user can use the *back* and *forward* buttons when navigating back and forth between pages.
- In addition, Angular Router allows us to:
  - redirect a URL to another URL
  - resolve data before a page is displayed
  - run scripts when a page is activated or deactivated
  - lazy load parts of our application.

Steps to create a routing app

1. Generate a CLI project with routing option
2. Generate the components. Eg DeptList and EmpList
3. Configure the routes
4. Add buttons and use directives to navigate



Step -1 : Generate a CLI project with routing option

- ng new RouterPrj --routing
- Go to index.html and add base tag in <head> in it if its not already there.
  - This helps the app construct URL's while navigating.

```
g.module.ts   index.html   app.module.ts
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>RouterPrj</title>
  <base href="/">
```

- Ensure that routing module is imported in app module (app.module.ts):

```
g.module.ts   index.html   app.module.ts   app.module.ts
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
@NgModule({
  declarations: [AppComponent],
  imports: [ BrowserModule,
    AppRoutingModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Step – 2 : Generate the components

ng g c dept-list
ng g c emp-list

Step – 3 : Configure the routes

- Open <prj>\src\app\app-routing.module.ts file.
  - This file contains the routing module for our app ; here we configure the different routes
  - To tell the paths and destinations to your browser, import the RouterModule and Routes from @angular/router into your app.module.ts file.

app-routing.module.ts | app.module.ts | app.module.ts | dept-list.component.ts

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

const routes: Routes = [];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

app.module.ts | app-routing.module.ts | dept-list.component.ts | app.compone

```
import { AppRoutingModule, routingComponents
      from './app-routing.module';

@NgModule({
  declarations: [ AppComponent,
    routingComponents
  ],
```

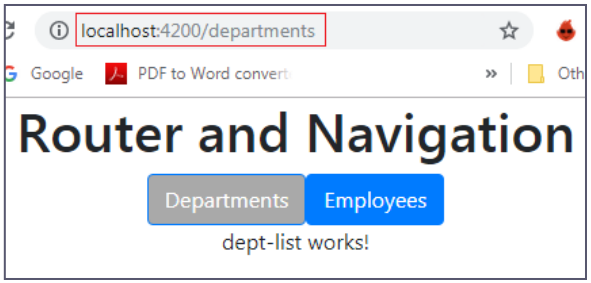
app-routing.module.ts | app.module.ts | dept-list.component.ts | app.component.html

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { DeptListComponent } from './dept-list/dept-list.component';
import { EmpListComponent } from './emp-list/emp-list.component';

const routes: Routes = [
  {path : 'departments', component : DeptListComponent},
  {path : 'employees', component : EmpListComponent}
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }

export const routingComponents = [DeptListComponent,EmpListComponent ];
```

Step – 4 : Add buttons and use directives to navigate

app.component.html | app.module.ts | app-routing.module.ts | dept-list.component.ts | styles.css

```
<div style="text-align:center">
  <h1>
    Router and Navigation
  </h1>
  <nav>
    <a class="btn btn-primary" routerLink="/departments"
       routerLinkActive="myactive">Departments</a>
    <a class="btn btn-primary" routerLink="/employees"
       routerLinkActive="myactive">Employees</a>
  </nav>
<!-- Routed views go here -->
<router-outlet></router-outlet>
```



- Directives used here:
  - **RouterLink** : lets you link to specific routes in your app.
  - **RouterLinkActive** : lets you add a CSS class to an element when the link's route becomes active
  - **RouterOutlet** : acts as a placeholder that Angular dynamically fills based on the current router state.