Angular 2.0

# Architecture Diagram:



The architecture diagram identifies the eight main building blocks of an Angular application:

* [Modules](https://angular.io/docs/ts/latest/guide/architecture.html#modules)
* [Components](https://angular.io/docs/ts/latest/guide/architecture.html#components)
* [Templates](https://angular.io/docs/ts/latest/guide/architecture.html#templates)
* [Metadata](https://angular.io/docs/ts/latest/guide/architecture.html#metadata)
* [Data binding](https://angular.io/docs/ts/latest/guide/architecture.html#data-binding)
* [Directives](https://angular.io/docs/ts/latest/guide/architecture.html#directives)
* [Services](https://angular.io/docs/ts/latest/guide/architecture.html#services)
* [Dependency injection](https://angular.io/docs/ts/latest/guide/architecture.html#dependency-injection)

## Modules

Angular apps are modular and Angular has its own modularity system called Angular modules or NgModules.

Every Angular app has at least one Angular module class, [the root module](https://angular.io/docs/ts/latest/guide/appmodule.html),  
conventionally named AppModule.

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.

NgModule is a decorator function that takes a single metadata object whose properties describe the module

Don't fetch data in a component constructor. You shouldn't worry that a new component will try to contact a remote server when created under test or before you decide to display it. Constructors should do no more than set the initial local variables to simple values.

## OnDestroy()

This is the place to free resources that won't be garbage collected automatically. Unsubscribe from Observables and DOM events. Stop interval timers. Unregister all callbacks that this directive registered with global or application services. You risk memory leaks if you neglect to do so.

Main Topics

Securiy

Testing

Forms

## Forms:

framework support for two-way data binding, change tracking, validation, and error handling

**dependency injection is**.

It's a coding pattern in which a class receives its dependencies from external sources rather than creating them itself.

Wouldn't it be nice if you could simply list the things you want to build without having to define which dependency gets injected into what?

This is where the dependency injection framework comes into play. Imagine the framework had something called an *injector*. You register some classes with this injector, and it figures out how to create them.

The Car knows nothing about creating an Engine or Tires. The consumer knows nothing about creating a Car. You don't have a gigantic factory class to maintain

## Injector providers

A provider provides the concrete, runtime version of a dependency value. The injector relies on **providers** to create instances of the services that the injector injects into components and other services.

## Single-page application (SPA)

A **single-page application** (**SPA**) is a web application or web site that fits on a single web page with the goal of providing a user experience similar to that of a desktop application. In an SPA, either all necessary code – HTML, JavaScript, and CSS – is retrieved with a single page load,[[1]](https://en.wikipedia.org/wiki/Single-page_application#cite_note-Flanagan2006-1) or the appropriate resources are dynamically loaded and added to the page as necessary, usually in response to user actions. The page does not reload at any point in the process, nor does control transfer to another page, although the location hash or the HTML5 History API can be used to provide the perception and navigability of separate logical pages in the application.[[2]](https://en.wikipedia.org/wiki/Single-page_application#cite_note-2) Interaction with the single page application often involves dynamic communication with the web server behind the scenes.

In previous versions of ECMAScript, everything was still defined by a prototype. Now classes are defined and it makes it almost as readable as Java code. ECMAScript 6 also has full inheritance. With super (parameters) to the constructor of the parent is called. Furthermore, static variables can be declared.

TypeScript is an extension of ECMAScript, in fact:

***TypeScript = ES6 + Types + Annotations***

TypeScript is actually from Microsoft, which means the new Angular is also likely to be popular for .NET developers. TypeScript is a form of JavaScript which knows types and classes and  can be compiled to JavaScript. It is open source. TypeScript includes many aspects of object orientation such as inheritance and interfaces. It also has generics and lambdas.

Because Angular2 uses TypeScript, the functionality of TypeScript itself and its libraries can be used. Angular is just a framework which couples different features. Other libraries can easily be used. For example, you can use the MongoDB interface because it already has a connector in TypeScript.

<html ng-app="yourApp">

<body>

<p>Displing model data in view through controller</p>

<div ng-controller="realCustomer">

<h3>{{ customer.Name }} </h3>

<h3>{{ customer.Address }} </h3>

<h3>{{ customer.Email }} </h3>

</div>

</body>

</html>

<script type="text/javascript">

//Creating controller here

var app = angular.module('yourApp', []);

app.controller('realCustomer', function($scope) {

//Creating model here

$scope.customer = {

'Name' : 'Jimi',

'Address' : '12-13-283/A1',

'Email' : 'jimi@yahoo.com'

}

});

</script>

### **Injectable services**

The @Injectable() decorator tells TypeScript to emit metadata about the service. The metadata specifies that Angular may need to inject other dependencies into this service.

Although the HeroService doesn't have any dependencies at the moment, applying the @Injectable() decorator ​from the start ensures consistency and future-proofing.

## **Async services and Promises**

The HeroService returns a list of mock heroes immediately; its getHeroes() signature is synchronous.

src/app/app.component.ts

content\_copythis.heroes = this.heroService.getHeroes();

Eventually, the hero data will come from a remote server. When using a remote server, users don't have to wait for the server to respond; additionally, you aren't able to block the UI during the wait.

To coordinate the view with the response, you can use Promises, which is an asynchronous technique that changes the signature of the getHeroes() method.

------------------------

Inside the ngOnInit() lifecycle hook, use the paramMap Observable to extract the id parameter value from the [ActivatedRoute](https://angular.io/api/router/ActivatedRoute) service and use the HeroService to fetch the hero with that id.

src/app/hero-detail.component.ts

content\_copyngOnInit(): void {

this.route.paramMap

.switchMap((params: ParamMap) => this.heroService.getHero(+params.get('id')))

.subscribe(hero => this.hero = hero);

}

The switchMap operator maps the id in the Observable route parameters to a new Observable, the result of the HeroService.getHero() method.

If a user re-navigates to this component while a getHero request is still processing, switchMap cancels the old request and then calls HeroService.getHero() again.

The hero id is a number. Route parameters are always strings. So the route parameter value is converted to a number with the JavaScript (+) operator.

## **Observable data**

Observable is one of the key classes in the [RxJS library](http://reactivex.io/rxjs/).

Note: Component classes should be lean. They don't fetch data from the server, validate user input, or log directly to the console. They delegate such tasks to services.

Angular doesn't *enforce* these principles. It won't complain if you write a "kitchen sink" component with 3000 lines.

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*.

In general, add providers to the [root module](https://angular.io/guide/architecture#modules) so that the same instance of a service is available everywhere.

src/app/app.module.ts (module providers)

content\_copyproviders: [

BackendService,

HeroService,

Logger

],

Alternatively, register at a component level in the providers property of the @[Component](https://angular.io/api/core/Component) metadata:

src/app/hero-list.component.ts (component providers)

content\_copy@[Component](https://angular.io/api/core/Component)({

selector: 'app-hero-list',

templateUrl: './hero-list.component.html',

providers: [ HeroService ]

})

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

### If-Else

<div \*[ngIf](https://angular.io/api/common/NgIf)="condition; then thenBlock else elseBlock"></div>

<ng-template #thenBlock>...</ng-template>

<ng-template #elseBlock>...</ng-template>

Hackerrank

Ragunath Kannan

Polyglot engineer

1000$ / person

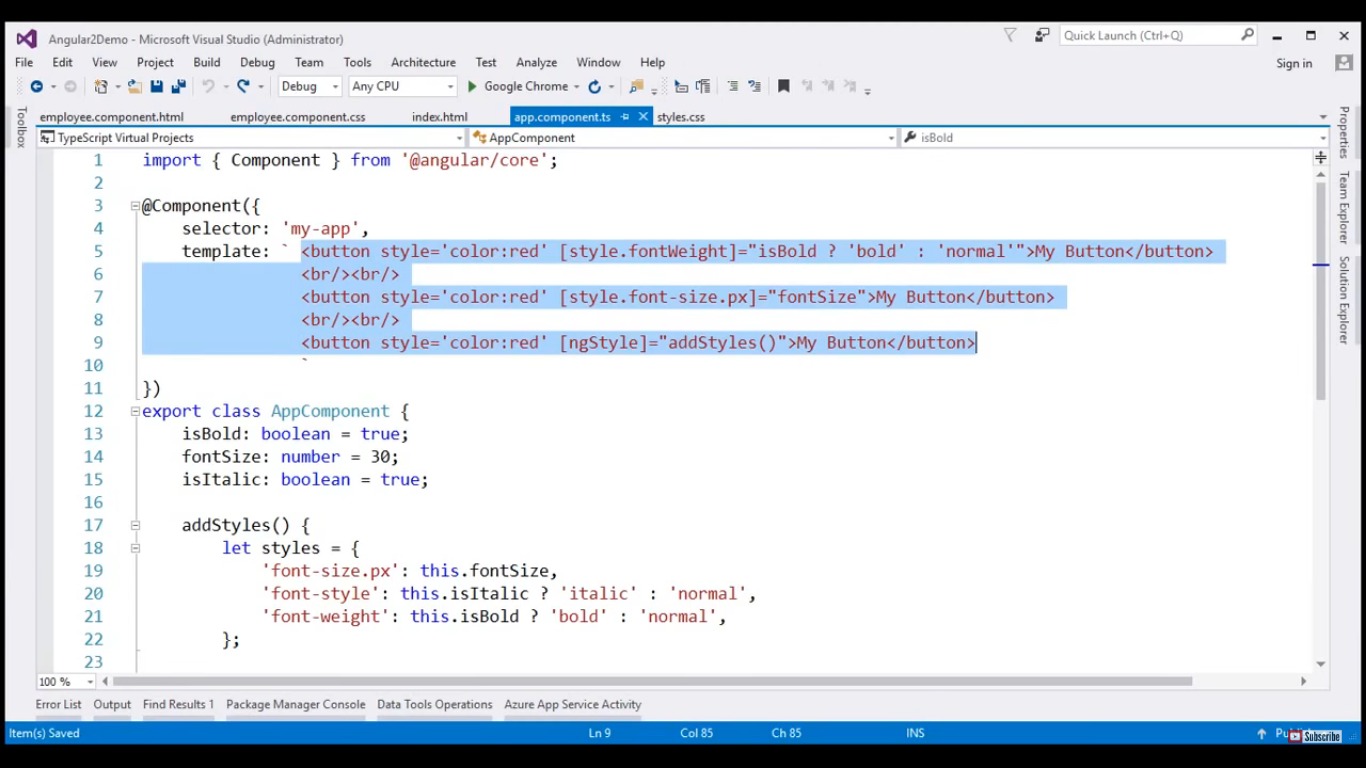
S1,2 – engi – finish

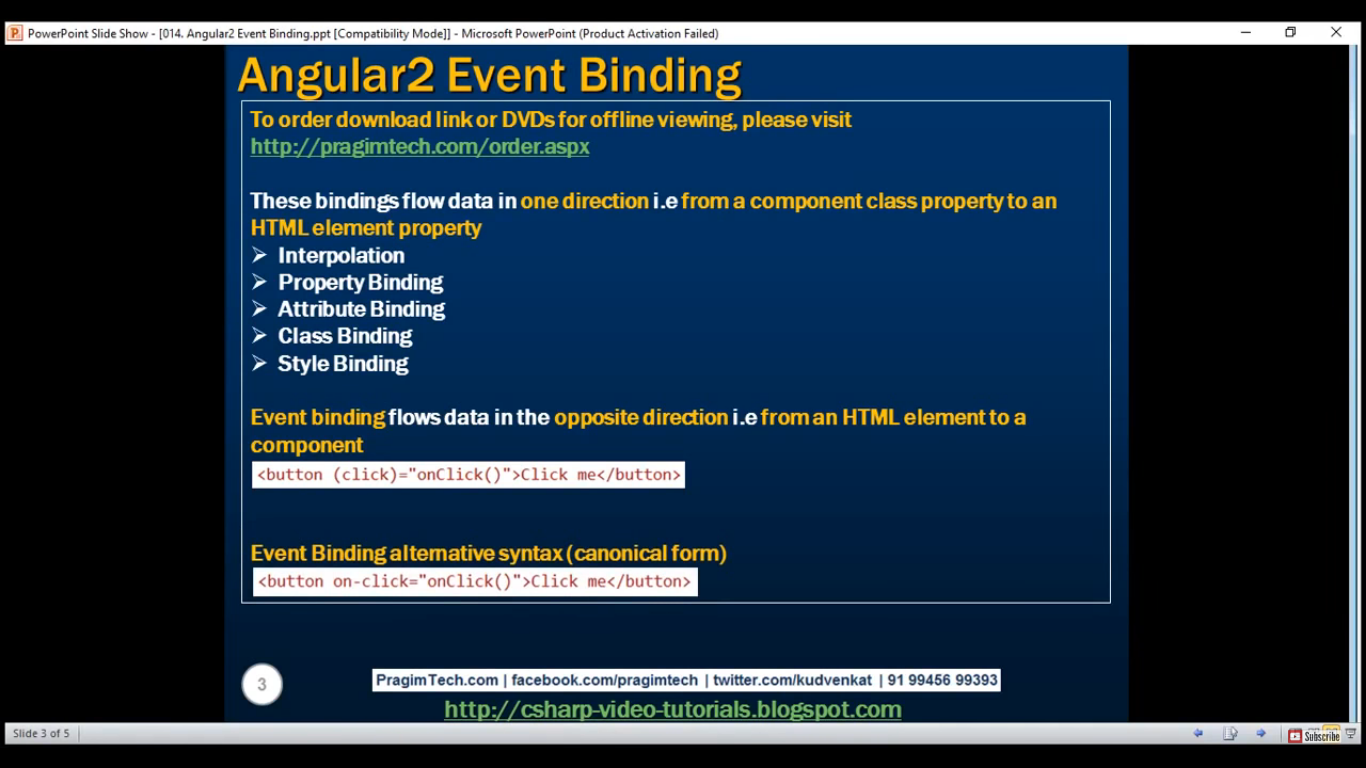
S3 arch

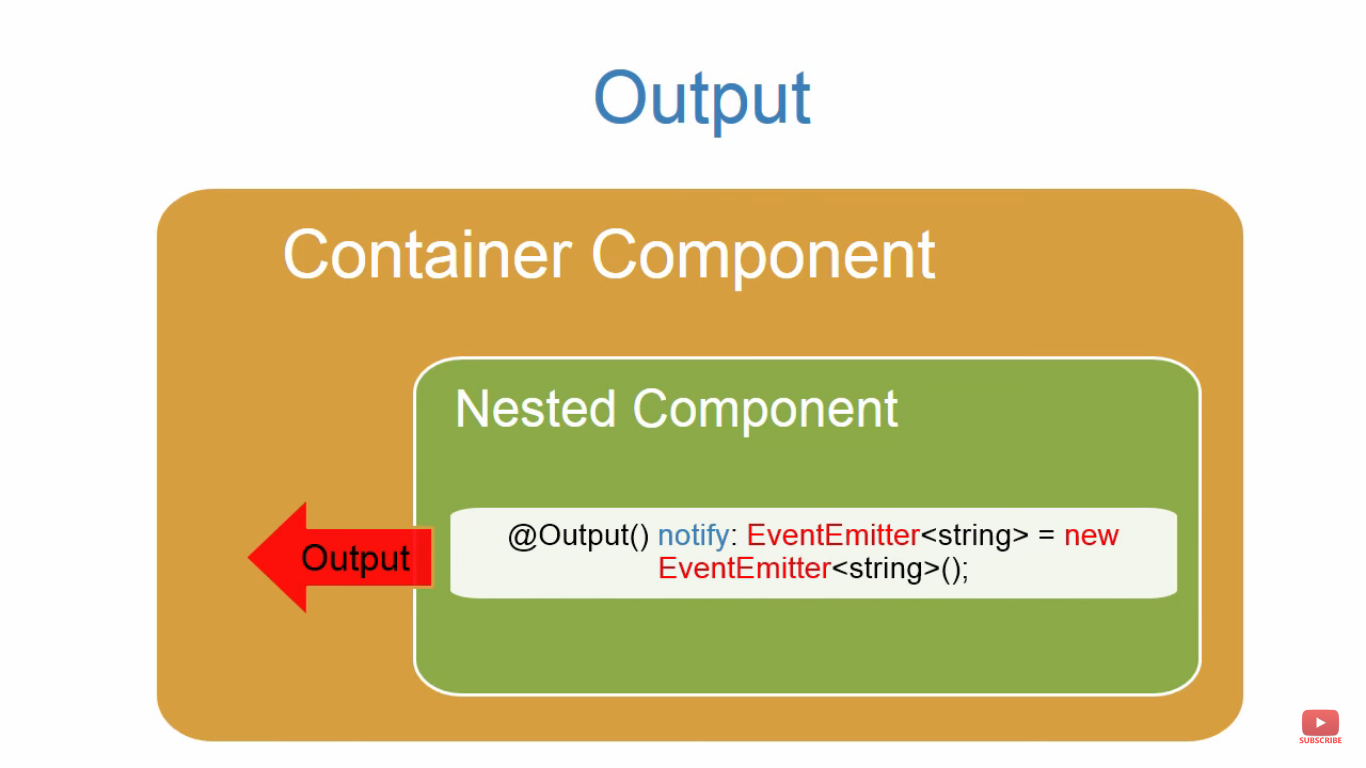
23 engi in cts

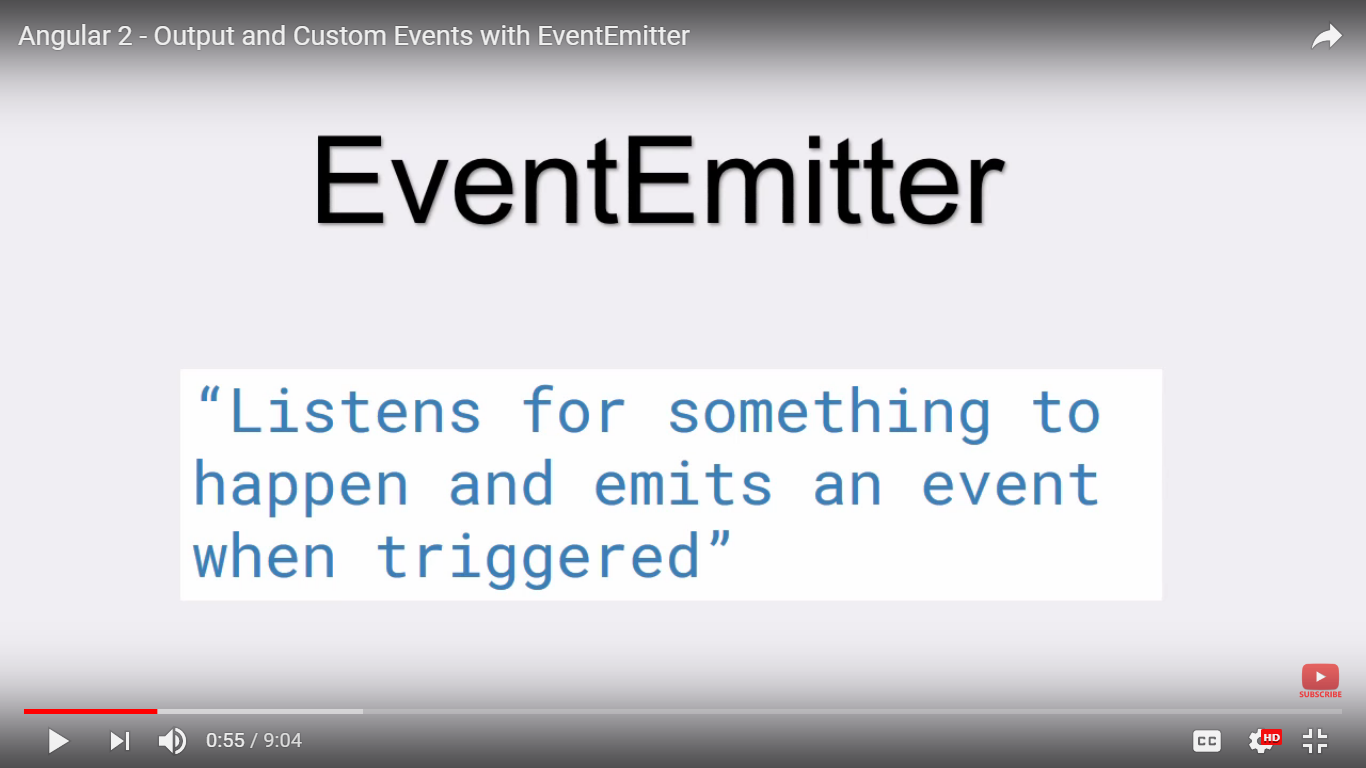
Industry wise

### NgStyle Example

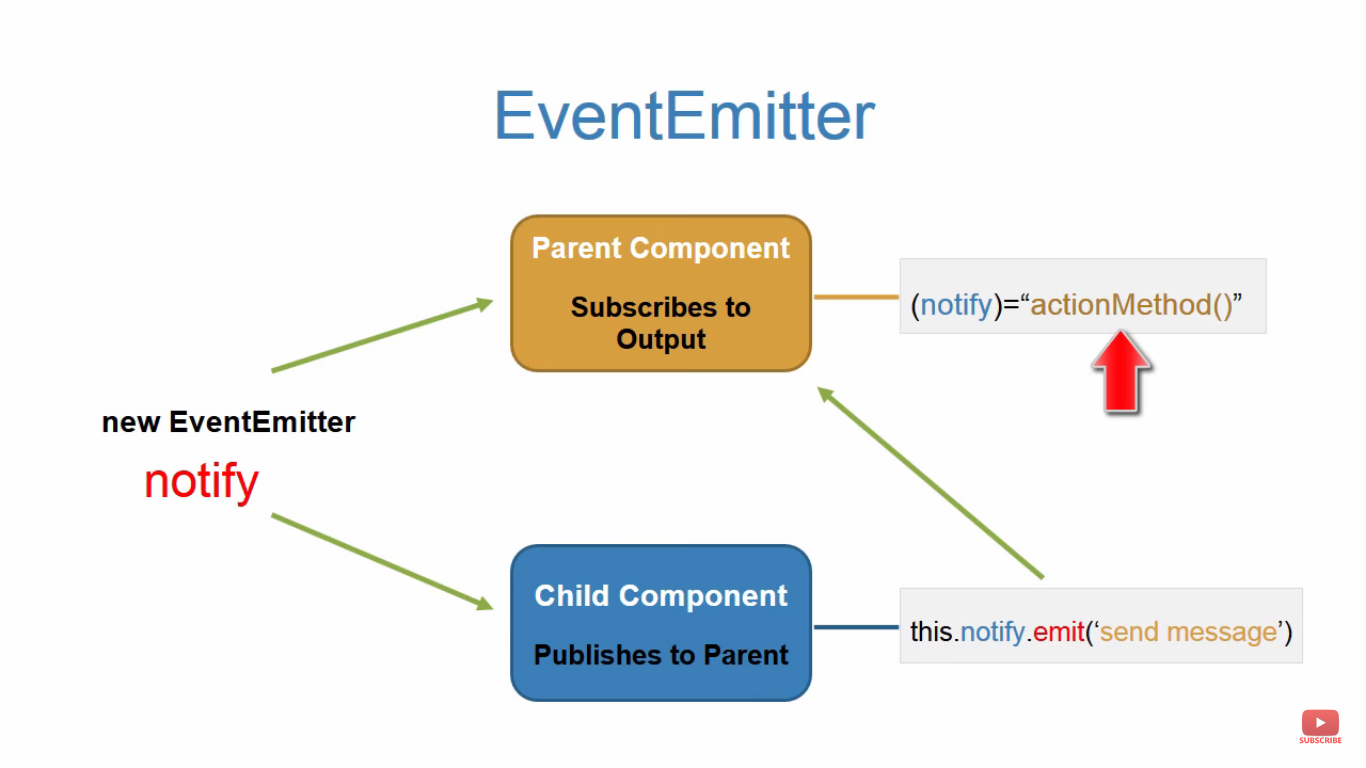


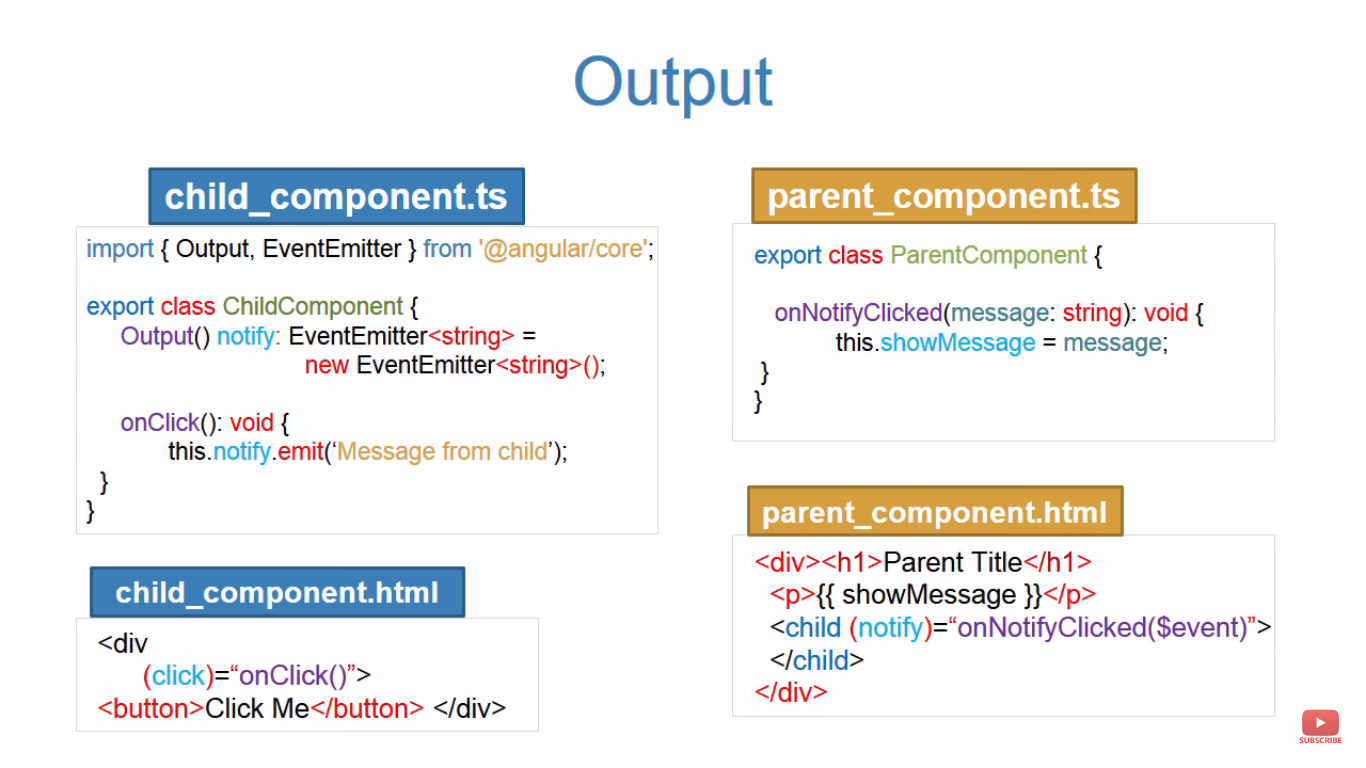






<https://www.youtube.com/watch?v=tZNWv-iW74I>





### 2 Way data binding with Event emitter

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

@Component({

selector: 'app-sizer',

template: `

<div>

<button (click)="dec()" title="smaller">-</button>

<button (click)="inc()" title="bigger">+</button>

<label [style.font-size.px]="size">FontSize: {{size}}px</label>

</div>`

})

export class SizerComponent {

@Input() size: number | string;

@Output() sizeChange = new EventEmitter<number>();

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

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

resize(delta: number) {

this.size = Math.min(40, Math.max(8, +this.size + delta));

this.sizeChange.emit(this.size);

}

}

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

<div [style.font-size.px]="fontSizePx">Resizable Text</div>

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