Angular Basics: A Comprehensive Introduction

Understanding the Core Concepts and Architecture of Angular

Angular is a powerful, open-source web application framework maintained by Google. It is widely used for building dynamic, feature-rich, and scalable single-page applications (SPAs) using TypeScript, a superset of JavaScript. Since its introduction, Angular has become one of the most popular frameworks in the web development community, thanks to its robust toolset, strong typing, and modular architecture.

# What Is Angular?

Angular is a platform and framework for building client-side applications using HTML, CSS, and TypeScript. Unlike its predecessor AngularJS, which was based on JavaScript, modern Angular leverages TypeScript to bring advanced features like static typing, classes, and interfaces, making code more maintainable and less prone to errors.

Angular facilitates the development of single-page applications, where the content dynamically updates without requiring a full page reload. This results in faster, more seamless user experiences.

# Key Features of Angular

* **Component-Based Architecture:** Angular applications are built using components—self-contained units that encapsulate logic, data, and presentation. Components make it easier to organize and reuse code.
* **Two-Way Data Binding**: Angular provides seamless synchronization between the model (data) and the view (UI), so changes in the data are immediately reflected on the UI and vice versa.
* **Dependency Injection:** Angular has a built-in dependency injection system, making it easier to develop, test, and maintain code by decoupling components from their dependencies.
* **Directives**: Angular offers powerful directives that let you extend HTML with custom behavior and control DOM rendering.
* **Routing:** Built-in router enables navigation between different views or pages within an application without reloading the page.
* **Services:** Services are used to encapsulate business logic, data access, and other non-UI functionalities, promoting code reusability and separation of concerns.
* **Testing**: Angular is designed with testing in mind, with support for unit and end-to-end testing through tools like Jasmine and Protractor.

# Setting Up an Angular Project

To get started with Angular, you’ll need to have Node.js and npm (Node Package Manager) installed on your system. Once they are installed, you can use the Angular CLI (Command Line Interface) to scaffold and manage Angular applications.

Install Angular CLI globally:

* npm install -g @angular/cli

Create a new Angular project:

ng new my-angular-app

Run the development server:

cd my-angular-app

ng serve

Open your browser and navigate to [URL] to view your application.

# Angular Project Structure

When you create a new Angular project, you’ll see a directory structure similar to the following:

* src/app/: Contains the main application code, including components, services, and modules.
* src/assets/: Static assets like images, stylesheets, etc.
* src/environments/: Environment-specific configuration files.
* angular.json: Configuration file for the Angular CLI.
* package.json: Lists project dependencies and scripts.

# Core Building Blocks of Angular

## 1. Modules

Angular applications are modular, and every Angular app has at least one module, the root module (usually called AppModule). Modules group components, directives, pipes, and services together.

Example:

@NgModule({

declarations: [AppComponent],

imports: [BrowserModule],

bootstrap: [AppComponent]

})

export class AppModule { }

## 2. Components

Components are the heart of Angular applications. Each component consists of:

* Template (HTML): Defines the view.
* Class (TypeScript): Contains logic and data.
* Styles (CSS): Defines how the component looks.

A simple component:

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

@Component({

selector: 'app-root',

template: '

# Hello, Angular!

'

})

export class AppComponent { }

## 3. Templates

Templates are HTML files or inline HTML code that define the structure of your view. Angular templates can use special syntax for data binding, event handling, and directives.

Example of interpolation:

{{ title }}

## 4. Data Binding

Angular supports several types of data binding:

* Interpolation: {{ data }}
* Property Binding: [property]="value"
* Event Binding: (event)="handler()"
* Two-Way Binding: [(ngModel)]="property"

## 5. Directives

Directives are special markers in templates that add behavior to elements. There are three types:

* Component Directives: Custom components.
* Structural Directives: Change the DOM layout, e.g., \*ngIf, \*ngFor.
* Attribute Directives: Change the appearance or behavior of an element, e.g., ngClass, ngStyle.

## 6. Services and Dependency Injection

Services are classes that provide specific functionality, such as fetching data from APIs or managing state. By injecting services into components, you promote reusability and separation of concerns.

Example service:

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

@Injectable({

providedIn: 'root',

})

export class DataService {

getData() {

return ['Angular', 'React', 'Vue'];

}

}

## 7. Routing

The Angular Router enables navigation between different views or components in your application. Define routes in the app-routing.module.ts file:

const routes: Routes = [

{ path: 'home', component: HomeComponent },

{ path: 'about', component: AboutComponent }

];

Use the directive in your template to display routed views.

# Best Practices

* Organize code using modules and folders for maintainability.
* Use services for business logic rather than placing logic in components.
* Keep components focused on presentation and user interaction.
* Utilize Angular CLI for consistent project scaffolding and management.
* Write unit tests and end-to-end tests to ensure code reliability.
* Embrace TypeScript features like interfaces and types for safer code.

# Conclusion

Angular is a comprehensive framework that empowers developers to build robust, maintainable single-page applications. Its modular architecture, powerful data binding, dependency injection, and out-of-the-box tooling make it a top choice for enterprise and large-scale web development. By understanding the basics—components, modules, templates, services, and routing—you lay a strong foundation for building and scaling complex applications with Angular. As you advance, you can explore advanced topics such as state management, lazy loading, advanced forms, and performance optimization to harness the full power of Angular.

# Creating Modules in Angular with Examples

Angular’s modular approach enables developers to organize application functionality into distinct, reusable units called modules. Each Angular module is defined by a class decorated with @NgModule, which specifies its components, directives, pipes, and services.

## What Is an Angular Module?

An Angular module is simply a class annotated with @NgModule. Every Angular application has at least one root module, typically named AppModule. Additional feature modules can be created to encapsulate related functionality and help structure your codebase.

## Basic Structure of a Module

Below is a simple example of a root module (app.module.ts):

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

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

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

@NgModule({

declarations: [AppComponent],

imports: [BrowserModule],

providers: [],

bootstrap: [AppComponent]

})

export class AppModule { }

## Creating a Feature Module

Suppose you want to create a module dedicated to user-related functionality (e.g., UserModule). Here’s how you can do it:

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

import { CommonModule } from '@angular/common';

import { UserListComponent } from './user-list/user-list.component';

import { UserDetailComponent } from './user-detail/user-detail.component';

@NgModule({

declarations: [

UserListComponent,

UserDetailComponent

],

imports: [

CommonModule

],

exports: [

UserListComponent,

UserDetailComponent

]

})

export class UserModule { }

Explanation:

* declarations: Components, directives, and pipes that belong to this module.
* imports: Other modules whose exported classes are needed by component templates declared in this module (here, CommonModule for common directives such as \*ngIf and \*ngFor).
* exports: Subset of declarations that should be visible and usable in the components of other modules.

## Registering the Feature Module

To use the UserModule in your application, import it into your root module (app.module.ts):

import { UserModule } from './user/user.module';

@NgModule({

imports: [

BrowserModule,

UserModule

],

declarations: [AppComponent],

bootstrap: [AppComponent]

})

export class AppModule { }

## Benefits of Using Modules

* Separation of concerns: Different functionalities are neatly separated.
* Code reusability: Feature modules can be reused across projects.
* Lazy Loading: Modules can be loaded on demand, improving application performance.
* Organization: Large applications become more manageable and scalable.

## Example: Lazy Loading a Module

Angular allows you to load modules only when needed through lazy loading, often applied to feature modules:

// app-routing.module.ts

const routes: Routes = [

{

path: 'users',

loadChildren: () => import('./user/user.module').then(m => m.UserModule)

}

];

This approach keeps the initial bundle size small and improves load times.

## Conclusion

Mastering modules in Angular is fundamental to building scalable and maintainable applications. By structuring your code into modules, you facilitate better team collaboration, code organization, and feature encapsulation, while unlocking advanced capabilities like lazy loading. Begin by modularizing your application, and you’ll soon enjoy a cleaner codebase and a smoother development process.

**Modules in Angular :**

Creating and managing Angular modules involves using the Angular CLI to generate modules and then configuring them to encapsulate related functionalities.

1. Creating a New Angular Module:

* **Open your terminal or command prompt**: and navigate to the root directory of your Angular project.
* **Generate a new module**: using the Angular CLI command:

Code

ng generate module <module-name>

For example, to create a module named auth, you would run:

Code

**ng generate module auth**

This command creates a new folder (e.g., src/app/auth) containing the auth.module.ts file. If you want a routing file for the module, add the --routing flag:

Code

**ng generate module auth --routing**

2. Configuring the Angular Module:

* **Open the generated module file**: (e.g., src/app/auth/auth.module.ts) in your code editor.
* Locate the @NgModule decorator. This is where you configure the module's properties.

TypeScript

import { NgModule } from '@angular/core';  
 import { CommonModule } from '@angular/common';  
  
 @NgModule({  
 declarations: [], *// Declare components, directives, pipes belonging to this module*  
 imports: [CommonModule], *// Import other modules*  
 providers: [], *// Register services available within this module*  
 exports: [] *// Export components, directives, pipes to be used by other modules*  
 })  
 export class AuthModule { }

* declarations:

Add components, directives, and pipes that belong specifically to this module.

* + Example: If you create an AuthComponent within the AuthModule, you would declare it here.
* imports:

Import other Angular modules (e.g., CommonModule, FormsModule, RouterModule) or custom feature modules that this module depends on.

* providers:

Register services that should be available within this module.

* exports:

List components, directives, or pipes from this module that you want to make available for use in other modules that import this module.

3. Managing Components within a Module:

Generate components within the module's directory.

Code

ng generate component <component-name> --module=<module-name>

For example, to create a LoginComponent within the AuthModule:

Code

ng generate component login --module=auth

This automatically declares the LoginComponent in the auth.module.ts file.

4. Using the Module in Other Parts of Your Application:

* **Import the module**: into another module where you want to use its exported components or services. For instance, to use AuthModule in AppModule, you would import it in app.module.ts:

TypeScript

import { AuthModule } from './auth/auth.module'; *// Adjust path as needed*  
  
 @NgModule({  
 declarations: [AppComponent],  
 imports: [  
 BrowserModule,  
 AuthModule *// Import the AuthModule*  
 ],  
 providers: [],  
 bootstrap: [AppComponent]  
 })  
 export class AppModule { }

By following these steps, you can effectively create, configure, and manage Angular modules to organize your application's features and improve maintainability.

**Angular — New Control flow with a working example**

[#angular](https://dev.to/t/angular)[#javascript](https://dev.to/t/javascript)[#signals](https://dev.to/t/signals)[#webdev](https://dev.to/t/webdev)

If you are watching recent Angular works, you’d probably know the addition of new control flows with new syntax. The new control flow is planned for the Angular v17 release that would probably face the public in November 2023. More details on why @ is chosen can be found [here](https://blog.angular.io/meet-angulars-new-control-flow-a02c6eee7843).

In the Angular v17.0.0-next.6 release, the team has shipped the control flow changes which are not production-ready but we can play around with it.

Let's see what are the available control flows.

**If else statement**

In previous versions, NgIf is used to show DOM elements conditionally which is declarative but is not suitable for the [new change detection strategy](https://maddydeep28.medium.com/angular-signals-new-change-detection-strategy-44f7ee9c359a). The new @if block will resolve this problem and will work well with signal-based change detection.

@Component({

selector: 'app-root',

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

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

})

export class AppComponent {

show = true;

showAnotherIf = false;

}

@if (show) {

<span>Inside if</span>

} @else if (showAnotherIf) {

<span>Inside else if</span>

} @else {

<span>Inside else</span>

}

**For loop**

The for loop allows you to render the given content based on the iterable object and it provides some useful context properties to work with. In addition, it provides a @empty block which will be rendered when no item is present in the given array.

@Component({

selector: 'app-root',

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

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

})

export class AppComponent {

skills = ['javascript', 'html', 'css'];

}

<ul>

@for (item of skills; track item; let i = $index, f = $first; let l = $last, ev = $even, o = $odd; let co = $count) {

<li>{{item}}

<ul>

<li>Index - {{i}}</li>

<li>Is First - {{f}}</li>

<li>Is Last - {{l}}</li>

<li>Is even - {{ev}}</li>

<li>Is odd - {{o}}</li>

<li>Count - {{co}}</li>

</ul>

</li>

} @empty {

<li>No item</li>

}

</ul>

**Switch case**

The new switch case control flow can be used as follows. You can use @default block to mention the default content to be rendered when no case is matched.

@Component({

selector: 'app-root',

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

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

})

export class AppComponent {

caseNo = 4;

}

@switch (caseNo) {

@case (1) {

<span>Rendering case 1</span>

}

@case (2) {

<span>Rendering case 2</span>

}

@case (3) {

<span>Rendering case 3</span>

}

@default {

<span>Rendering default</span>

}

}

**How to enable the new control flow in the Angular v17.0.0-next.6?**

Even though the changes are available on the package, you cannot directly use this now and will end up with the below error.

error NG5002: Invalid ICU message. Missing ‘}’

or

error NG5002: Unexpected character “EOF”

(Do you have an unescaped “{“ in your template?

Use “{{ ‘{‘ }}”) to escape it.)

To try the new control flows now you have to add the below config in the angularCompileroptions.

{

"angularCompilerOptions": {

....

"\_enabledBlockTypes": [

"if", "switch", "for"

]

}

}

I got your mind's voice, yeah \_enabledBlockTypes is an internal one that will be changed eventually. But for now, it will allow you to play around with the control flows.

**Angular Templates**

When experimenting with control flows, understanding Angular templates is essential, as they allow you to define dynamic and interactive views using HTML syntax enhanced by Angular's template features. Here’s an overview and some examples to help you get started:

# What Are Angular Templates?

Angular templates are HTML views with special syntax for displaying data, handling user input, and controlling rendering logic using structural directives (like \*ngIf, \*ngFor, etc.). Templates are at the core of Angular's component-driven architecture.

# Common Template Syntax and Examples

* Interpolation
* Display component data inside the template.
* Hello, {{ userName }}!
* If your component has userName = 'Alex', this renders as: Hello, Alex!
* Property Binding
* Bind a component property to an HTML element property.
* Event Binding
* Respond to user actions like clicks.
* Logout
* Two-way Binding
* Keep component and input values in sync.

## Structural Directives

These directives shape or reshape the DOM's structure based on data.

* \*ngIf
* Conditionally add or remove elements.
* Welcome back!
* \*ngFor
* Repeat a block of HTML for each item in a list.
* {{ item }}
* \*ngSwitch
* Switch among multiple templates.
* Active
* Inactive
* Unknown

## Template Reference Variables

Give a name to a DOM element to reference it elsewhere in the template.

Search

## Template Expressions

Templates can contain simple expressions—no assignments, chaining, or side effects.

{{ 2 \* rating + 5 }}

# Summary

Angular templates blend HTML with Angular-specific syntax to let you create dynamic, data-driven views. By combining interpolation, property/event bindings, and structural directives, you can build robust and interactive user interfaces. Experiment with these techniques—especially now that you have control flows enabled—to see Angular’s power in action!

**Binding in Angular**

Binding in Angular is the mechanism through which you link your component's data and logic to the template, enabling seamless communication between the user interface and your application's internal state.

Angular provides several types of binding:

* Interpolation: Embeds dynamic values into your HTML, like {{ username }}.
* Property binding: Sets element properties using square brackets, such as .
* Event binding: Listens for user actions with parentheses, for instance .
* Two-way binding: Synchronizes data between the template and component using [(ngModel)], allowing real-time updates in both directions.

By mastering these binding techniques, you unlock the ability to create highly responsive, interactive applications qwhere data flows naturally between your code and the visual elements. Angular’s binding syntax makes it easy to manage complexity and ensure that your interface always reflects the underlying state of your application.

**AngularJS Data Binding**

In this article, we will see the Data Binding in AngularJS, along with understanding the various types of Data Binding available with their implementations.

Angular provides a function [Data Binding](https://docs.angularjs.org/guide/databinding) which helps us to have an almost real-time reflection of the input given by the user i.e. it creates a connection between Model and View. **Data Binding** is a way to synchronize the data between the model and view components automatically. AngularJS implements data-binding that treats the model as the single-source-of-truth in your application & for all the time, the view is a projection of the model. Unlike React, angular supports two-way binding. In this way, we can make the code more loosely coupled. Data binding can be categorized into 2 types, ie., One-way Binding & Two-way Binding.

**One-way Binding:**This type of binding is unidirectional, i.e. this binds the data flow from either component to view(DOM) or from the view(DOM) to the component. There are various techniques through which the data flow can be bind from component to view or vice-versa. If the *data flow from component to view(DOM)*, then this task can be accomplished with the help of **String Interpolation**& **Property Binding.**

[**Interpolation**](https://www.geeksforgeeks.org/angular-js/string-interpolation-in-angular-8/)**:** Angular interpolation is used to display a component property in the respective view template with double curly braces syntax. Interpolation is used to transfer properties mentioned in the component class to be reflected in its template.

**Syntax**:

class="{{variable\_name}}"

**Example:**This example describes the **Interpolation**in AngularJS.

<**h3**>Binding Types</**h3**>

<**p**>Interpolation</**p**>

<**br**>

<**h5**>

Addition of 3 and 5 with

Interpolation is {{3+5}}

</**h5**>

<**h5**>

Addition of 3 and 5 without

Interpolation is 3+5

</**h5**>

<**h2**>{{val}}</**h2**>

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

@Component({

selector: 'my-app',

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

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

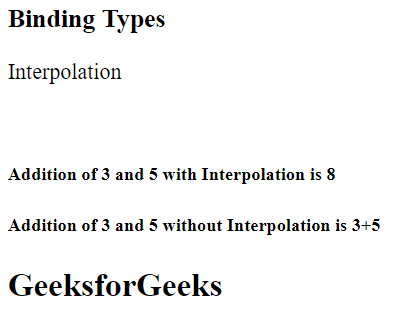
})

**export** **class** AppComponent {

val: string = 'GeeksforGeeks';

}

**Output:**

[](https://media.geeksforgeeks.org/wp-content/uploads/20220714071253/1-300x232.png)Interpolation

[**Property Binding**](https://www.geeksforgeeks.org/angular-js/property-binding-in-angular-8/)**:**Similar to Java, variables defined in the parent class can be inherited by the child class which is templates in this case. The only difference between Interpolation and Property binding is that we should not store non-string values in variables while using interpolation. So if we have to store Boolean or other data types then use Property Binding. In simple words, we bind a property of a DOM element to a field which is a defined property in our component TypeScript code.

**Syntax**:

[class]="variable\_name"

**Example:**This example describes the **Property Binding** in Angular JS.

<**h3**>Binding Types</**h3**>

<**p**>Property Binding</**p**>

<**input** type="text"

ng-bind="{{ Geeks }}"><**br**>

<**h5**>

Learning Property binding on {{ Geeks }}

</**h5**>

<**img** [src]="image" height="50px" weight="40px">

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

@Component({

selector: "app-root",

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

styleUrls: ["./app.component.css"],

})

**export** **class** AppComponent {

title = "Geeks";

classtype = "text-danger";

Geeks = "GeeksforGeeks";

image =

"https://media.geeksforgeeks.org/wp-content/uploads/geeksforgeeks-6.png";

}

**Output**:

[](https://media.geeksforgeeks.org/wp-content/uploads/20220714073643/property-300x234.png)Property Binding

If the flow of data is from view to component, then this can be achieved by using the Event Binding.

[**Event Binding**](https://www.geeksforgeeks.org/angular-js/event-binding-in-angular-8/)**:**An event is created whenever either a key is pressed or on a mouse clicked. It is used to handle the events raised by the user actions like button click, mouse movement, keystrokes, etc. When the DOM event happens at an element(e.g. click, keydown, keyup), it calls the specified method in the particular component. Using Event Binding, we can bind data from DOM to the component and hence can use that data for further purposes.

**Syntax**:

<button class="btn btn-block"

(click)=showevent($event)>

Event

</button>

showevent(event) {

alert("Welcome to GeeksforGeeks");

}

**Example:**This example describes the **Event Binding** in Angular JS.

<**h3**>Binding Types</**h3**>

<**p**>Event Binding</**p**>

<**button** class="btn btn-block"

(click)="Clickme($event)">

Click Here

</**button**>

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

@Component({

selector: "app-root",

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

styleUrls: ["./app.component.css"],

})

**export** **class** AppComponent {

title = "Geeks";

Clickme(event) {

alert("Welcome to GeeksforGeeks");

}

}

**Output**:

[A computer screen shot of a computer

AI-generated content may be incorrect.](https://media.geeksforgeeks.org/wp-content/uploads/20220714080022/EventBinding.gif)Event Binding

**Two-way Binding:**In this type of binding, the immediate changes to the view & component, will be reflected automatically, i.e. when the changes made to the component or model then the *view*will render the changes simultaneously. Similarly, when the data is altered or modified in the view then the model or component will be updated accordingly.

In *app.module.ts,* we have to include *FormsModule* in imports like the way given down also we have to import FormsModule. We have to include FormsModule, since *ngModel*is not a property included in the project we develop using *ng new project-name,* so we have to include it by importing this Module.

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

imports: [

BrowserModule,

FormsModule,

AppRoutingModule,

],

**Example:**This example describes the implementation of **Two-way Data Binding**.

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

<**h1** style="color: green">

GeeksforGeeks

</**h1**>

<**h3**>Two-way Data Binding</**h3**>

<**input** type="text"

placeholder="Enter text"

[(ngModel)]="val" />

<**br** />

{{ val }}

</**div**>

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

@Component({

selector: "my-app",

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

styleUrls: ["./app.component.css"],

})

**export** **class** AppComponent {

val: string;

}

**Output:**

[A computer screen shot of a computer

AI-generated content may be incorrect.](https://media.geeksforgeeks.org/wp-content/uploads/20220714092620/TwoWay.gif)Two-way Data Binding

Directives in Angular: Concepts and Examples

Understanding Angular Directives and Their Practical Applications

Angular, a popular framework for building dynamic web applications, is built around several powerful concepts, among which directives stand out as a way to extend HTML with custom behavior. Directives in Angular can be used to manipulate the DOM, create reusable components, and implement custom logic right within your HTML templates. In this document, we will explore the types of directives in Angular, provide detailed examples, and explain how they can be leveraged to create robust and interactive applications.

# What are Directives in Angular?

Angular directives are classes that add behavior to elements in your Angular applications. Directives can change the appearance, behavior, or layout of a DOM element, component, or even another directive. They are one of the core features of Angular, allowing developers to write cleaner, more maintainable code by separating concerns and promoting reuse.

There are three main types of directives in Angular:

* Component Directives
* Structural Directives
* Attribute Directives

# 1. Component Directives

Component directives are the most commonly used directives in Angular. In fact, every Angular component is a directive with a template. The @Component decorator is used to define them. Component directives encapsulate templates, styles, and logic, making them reusable across the application.

## Example: Basic Component Directive

Suppose you have a simple component as shown below:

@Component({

selector: "app-greeting",

template: " Hello, {{name}}! ",

styles: ["h1 { color: green; }"]

})

export class GreetingComponent {

name: string = "Angular";

}

This GreetingComponent is a component directive. You can use it in other templates as .

# 2. Structural Directives

Structural directives are responsible for changing the DOM layout by adding or removing elements. They are easy to identify because their names usually start with an asterisk (\*). The most commonly used structural directives are \*ngIf, \*ngFor, and \*ngSwitch.

## Example: \*ngIf Directive

The \*ngIf directive conditionally includes a template based on the value of an expression.

Welcome back, user!

If isLoggedIn is true, the div is rendered; otherwise, it is not present in the DOM.

## Example: \*ngFor Directive

The \*ngFor directive is used for iterating over a list of items.

{{fruit}}

Assuming fruits = ['Apple', 'Banana', 'Orange'];, Angular will render a list item for each fruit.

## Example: \*ngSwitch Directive

The \*ngSwitch directive allows you to conditionally switch between multiple views.

Red!

Blue!

Other color

Depending on the value of color, the corresponding view will be displayed.

# 3. Attribute Directives

Attribute directives are used to change the appearance or behavior of DOM elements. Unlike structural directives, they do not add or remove elements but can change their properties and styles.

## Example: ngClass Directive

The ngClass directive allows you to dynamically add or remove CSS classes.

Status

The active and disabled classes will be applied based on the component's properties.

## Example: ngStyle Directive

The ngStyle directive lets you set inline styles dynamically.

Dynamic Styling!

The selectedColor and fontSize are component properties that define the style.

## Example: Custom Attribute Directive

You can create your own attribute directive for custom behavior. Here’s an example that changes the background color of an element when hovered:

import { Directive, ElementRef, HostListener } from '@angular/core';

@Directive({

selector: '[appHoverHighlight]'

})

export class HoverHighlightDirective {

constructor(private el: ElementRef) {}

@HostListener('mouseenter') onMouseEnter() {

this.el.nativeElement.style.backgroundColor = 'yellow';

}

@HostListener('mouseleave') onMouseLeave() {

this.el.nativeElement.style.backgroundColor = null;

}

}

Apply it to any element like this:

Hover over me!

When you hover over the paragraph, its background color will change.

# Directive Lifecycle and Best Practices

Directives in Angular have their own lifecycles, similar to components. They can implement lifecycle hooks such as ngOnInit, ngOnChanges, and ngOnDestroy to handle initialization, changes, and cleanup.

When creating custom directives, keep these best practices in mind:

* Keep directives focused: Each directive should do one thing and do it well.
* Reuse logic: Use directives for shared logic across multiple components.
* Use ElementRef and Renderer2 wisely: Direct DOM manipulation should be minimized and done using Angular's abstractions.
* Encapsulate behavior: Hide implementation details from consuming components.

# Commonly Used Built-in Directives

Angular provides a range of built-in directives that simplify application development:

|  |  |  |  |
| --- | --- | --- | --- |
| Directive | Type | Description | Example Usage |
| \*ngIf | Structural | Conditionally adds/removes elements | <div \*ngIf="isVisible">Show me</div> |
| \*ngFor | Structural | Loops over collections | <li \*ngFor="let item of items">{{item}}</li> |
| [ngClass] | Attribute | Conditional CSS classes | <div [ngClass]="{'active': isActive}"></div> |
| [ngStyle] | Attribute | Dynamic inline styles | <span [ngStyle]="{color: color}"></span> |
| [ngModel] | Attribute | Two-way data binding | <input [(ngModel)]="val"> |
| [hidden] | Attribute | Hides element conditionally | <div [hidden]="!show">Hidden content</div> |
| ng-container | Structural | Logical grouping without rendering extra DOM | <ng-container \*ngIf="condition">...</ng-container> |

# How to Create a Custom Directive

To create your own directive, use the Angular CLI command:

ng generate directive highlight

This will scaffold a new directive in your project. Update its logic to suit your needs.

## Example: Text Color Directive

Here’s a custom directive that sets the text color:

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

@Directive({

selector: '[appTextColor]'

})

export class TextColorDirective implements OnInit {

@Input() appTextColor: string;

constructor(private el: ElementRef) {}

ngOnInit() {

this.el.nativeElement.style.color = this.appTextColor || 'blue';

}

}

Usage in a template:

This text is red.

# When to Use Directives

Directives are vital when you need to:

* Encapsulate UI logic and behaviors for reuse across components.
* Manipulate the DOM beyond what Angular templates allow natively.
* Apply conditional rendering or styling without altering the component logic.
* Separate presentation concerns from business logic.

# Conclusion

Angular directives empower developers to extend HTML with powerful, reusable functionality. Through built-in directives like \*ngIf, \*ngFor, and ngClass, and the ability to create custom directives, you can efficiently manage your application's UI and logic. Whether you're building simple dynamic forms or complex interactive dashboards, mastering directives is key to becoming an effective Angular developer.

==================================================================

**Built-in Pipes in Angular 20**

Angular provides a rich set of built-in pipes for transforming data in templates:

| **Pipe Name** | **Purpose** |
| --- | --- |
| DatePipe | Formats date values based on locale |
| CurrencyPipe | Formats numbers as currency |
| DecimalPipe | Formats numbers with decimal places |
| PercentPipe | Converts numbers to percentage strings |
| UpperCasePipe | Converts text to uppercase |
| LowerCasePipe | Converts text to lowercase |
| TitleCasePipe | Capitalizes the first letter of each word |
| SlicePipe | Extracts a portion of a string or array |
| JsonPipe | Converts objects to JSON string (for debugging) |
| KeyValuePipe | Converts objects/maps to key-value pairs |
| AsyncPipe | Automatically subscribes to Observables or Promises |
| I18nPluralPipe | Handles pluralization based on locale |
| I18nSelectPipe | Selects values based on locale-specific keys |

<!-- app.component.html -->

<h2>{{ companyName | titlecase }}</h2>

<p>Date: {{ purchaseDate | date:'fullDate' | uppercase }}</p>

<p>Amount: {{ totalAmount | currency:'INR':'symbol':'1.2-2' }}</p>

<p>Discount: {{ discountRate | percent }}</p>

<p>Short Note: {{ note | slice:0:20 | lowercase }}</p>

<p>Debug Info: {{ productDetails | json }}</p>

// app.component.ts

export class AppComponent {

companyName = 'acme technologies';

purchaseDate = new Date(2025, 7, 9);

totalAmount = 12345.678;

discountRate = 0.15;

note = 'This is a limited-time offer for premium customers only.';

productDetails = { id: 101, name: 'Laptop', specs: { ram: '16GB', cpu: 'i7' } };

}

**Inline**, **external**, and **scoped styles**

**inline**, **external**, and **scoped styles** in web development, especially relevant when working with Angular or any modern frontend framework:

**1. Inline Styles**

**Definition**: CSS rules applied directly to an HTML element using the style attribute.

**Example**:

<h2 style="color: green; font-size: 18px;">Welcome to the Dashboard</h2>

**Pros**:

* Quick and easy for small tweaks
* Overrides other styles due to high specificity

**Cons**:

* Not reusable
* Hard to maintain for large projects

**2. External Styles**

* **Definition**: CSS rules stored in a separate .css file and linked to the HTML or Angular component.

**Example** :

<!-- index.html or component.html -->

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

/\* styles.css \*/

h2 {

color: blue;

font-size: 24px;

}

**Pros**:

* Clean separation of concerns
* Reusable across multiple components/pages
* Easier to maintain and optimize

**Cons**:

* Requires additional HTTP request (though often cached)

**3. Scoped Styles (Component-Level**

Definition: Styles defined within an Angular component’s .css or .scss file that apply only to that component.

// app.component.ts

@Component({

selector: 'app-root',

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

styleUrls: ['./app.component.css'] // Scoped to this component

})

**/\* app.component.css \*/**

h2 {

color: purple;

}

Angular uses ViewEncapsulation to scope styles by default, preventing leakage into other components.

Pros:

* Encapsulated styling
* Avoids global conflicts
* Ideal for modular design

Cons:

* May require ::ng-deep or :host selectors for deep styling

**Best practices in Angular :**

When styling Angular components, adhering to best practices ensures maintainable and scalable applications. Here are several key guidelines:

* **Prefer Component Stylesheets**: Place CSS rules in the component’s styles or styleUrls property rather than global stylesheets. This leverages Angular’s encapsulation and keeps styles modular.
* **Use :host and :host-context selectors**: Employ :host for targeting the component wrapper and :host-context for applying styles based on the parent context. This enables nuanced, scoped styling.
* **Avoid ::ng-deep when possible**: While ::ng-deep can pierce encapsulation, it’s deprecated and may be removed in the future. Use it sparingly and prefer other encapsulation strategies.
* **Leverage CSS Variables:** For themes and dynamic styling, CSS variables can provide flexibility across components.
* **Utilize Angular’s SCSS support**: SASS/SCSS allows nesting, variables, and mixins, improving style organization and reuse.
* **Minimize global styles**: Reserve global styles for resets or typography. Encapsulated component styles should handle most layout and appearance concerns.
* **Consistent Naming Conventions**: Use clear, consistent class names and selectors to avoid confusion and ensure maintainability.
* **Responsive Design:** Incorporate media queries and flexible units to maintain usability across devices.

By following these practices, you can ensure your Angular components remain visually coherent, modular, and easy to manage as your application grows.

**Component Lifecycle in Angular**

In Angular, Components are the fundamental building blocks of an application. Understanding the lifecycle of these components is crucial for effective Angular Development. Angular provides several lifecycle hooks that allow developers to tap into key moments in a Component’s lifecycle and execute custom logic during those times.

**Component Lifecycle Stages**

The component lifecycle in Angular consists of several stages:

* **Creation:**The component is instantiated and its dependencies are injected.
* **Change Detection:** Angular checks for changes in the data-bound properties.
* **Rendering:**The component's template is rendered or updated.
* **Destruction:**The component is destroyed and cleaned up.

**Lifecycle Hooks**

Angular provides a set of lifecycle hooks that allow developers to execute code at specific stages of a component’s lifecycle.

**1. ngOnChanges**

It is called before ngOnInit (if the component has bound inputs) and whenever one or more data-bound input properties change. It is used to respond to changes in input properties.

ngOnChanges(changes: SimpleChanges) {  
 console.log('Changes detected:', changes);  
}

**2. ngOnInit**

It is Called once, after the first ngOnChanges. It is used to initialize the component after Angular first displays the data-bound properties.

ngOnInit() {  
 console.log('Component initialized');  
}

**3. ngDoCheck**

It is called during every change detection run, immediately after ngOnChanges and ngOnInit. It is used to detect and act upon changes that Angular can't or won't detect on its own.

ngDoCheck() {  
 console.log('Custom change detection');  
}

**4. ngAfterContentInit**

It is called once after the first ngDoCheck. It is used to perform any additional initialization required for the content.

ngAfterContentInit() {  
 console.log('Content initialized');  
}

**5. ngAfterContentChecked**

It is called after ngAfterContentInit and every subsequent ngDoCheck. It is used to act upon any changes after the content has been checked.

ngAfterContentChecked() {  
 console.log('Content checked');  
}

**6. ngAfterViewInit**

It is called once after the first ngAfterContentChecked. It is used to perform additional initialization required for the view.

ngAfterViewInit() {  
 console.log('View initialized');  
}

**7. ngAfterViewChecked**

It is called after ngAfterViewInit and every subsequent ngAfterContentChecked. It is used to act upon any changes after the view has been checked.

ngAfterViewChecked() {  
 console.log('View checked');  
}

**8. ngOnDestroy**

It is called immediately before Angular destroys the component. It is used to clean up any resources, such as subscriptions and event handlers, to avoid memory leaks.

ngOnDestroy() {  
 console.log('Component destroyed');  
}

**Step-by-Step Guide to Create a Standalone Angular Component**

**Step 1: Install Angular CLI**

Ensure you have Node.js installed. Then, install Angular CLI globally:

npm install -g @angular/cli

**Step 2: Create a New Angular Project**

Create a new Angular project:

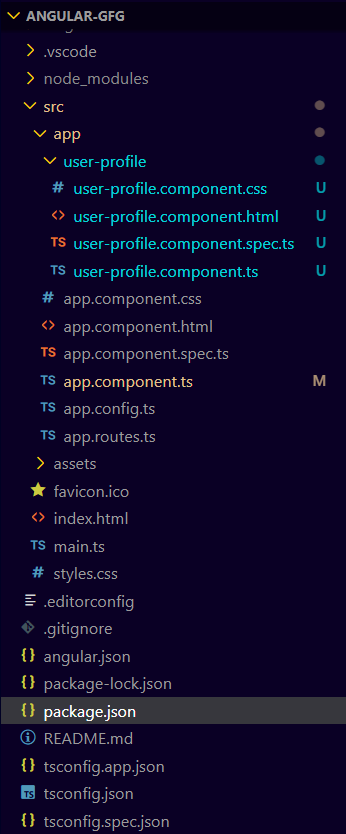
ng new angular-gfg  
cd angular-gfg

**Step 3: Generate a Standalone Component**

Generate a new standalone component called user-profile:

ng generate component user-profile --standalone

**Folder Structure**



Folder Structure

**Dependencies**

"dependencies": {  
 "@angular/animations": "^17.3.0",  
 "@angular/common": "^17.3.0",  
 "@angular/compiler": "^17.3.0",  
 "@angular/core": "^17.3.0",  
 "@angular/forms": "^17.3.0",  
 "@angular/platform-browser": "^17.3.0",  
 "@angular/platform-browser-dynamic": "^17.3.0",  
 "@angular/router": "^17.3.0",  
 "rxjs": "~7.8.0",  
 "tslib": "^2.3.0",  
 "zone.js": "~0.14.3"  
 }

**Step 4: Implement Lifecycle Hooks in UserProfileComponent**

*<!--src/app/user-profile/user-profile.component.html-->*

<**div** \*ngIf="user">

<**h2**>{{ user.name }}</**h2**>

<**p**>{{ user.email }}</**p**>

</**div**>

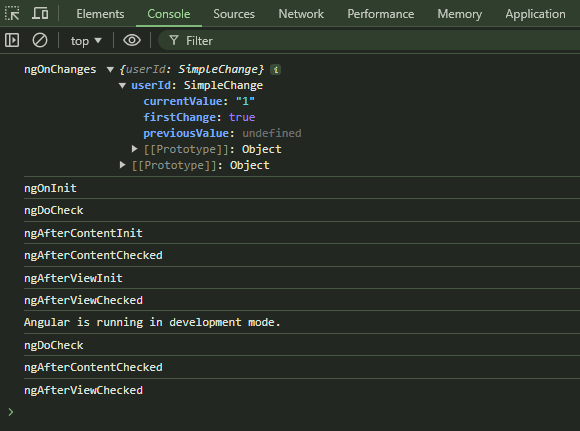
**Step 5: Run the Application**

Run the application to see the lifecycle hooks in action:

ng serve

Open your browser and navigate to http://localhost:4200. You should see the component displayed along with the console logs showing the lifecycle hook calls.

**Output**



**Passing data between parent and child components**

Sharing Data from Parent to Child Components in Angular using @Input()

In this article, we will explore how to share data from a parent component to a child component using the @Input() decorator in Angular.

Understanding @Input() Decorator

The @Input() decorator is an Angular feature that allows you to pass data from a parent component to a child component. It essentially creates an input property on the child component, which can be bound to a value in the parent component's template. Whenever the value of the input property changes in the parent, the child component is automatically updated with the new value.

Setting Up the Parent Component

Let's start by creating a simple example. Imagine we have a parent component that displays a user's name, and we want to pass this name to a child component for display.

1. Create a new parent component using the Angular CLI:

ng generate component parent

HTTP

Open the parent.component.ts file and define a property with the @Input() decorator:

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

@Component({

selector: 'app-parent',

template: `

<h1>Hello, {{ userName }}!</h1>

<app-child [inputName]="userName"></app-child>

`,

})

export class ParentComponent {

userName = 'Tahir Ansari';

}

JavaScript

Creating the Child Component

Now, let's create the child component that will receive and display the user's name.

1. Generate a child component using the Angular CLI:

ng generate component child

HTTP

In the child.component.ts file, use the @Input() decorator to define an input property:

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

@Component({

selector: 'app-child',

template: `

<p>Received name from parent: {{ receivedName }}</p>

`,

})

export class ChildComponent {

@Input() inputName: string;

get receivedName() {

return this.inputName;

}

}

JavaScript

Wiring Up the Module

ensure that you declare both the parent and child components in your module:

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

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

import { ParentComponent } from './parent.component';

import { ChildComponent } from './child.component';

@NgModule({

imports: [BrowserModule],

declarations: [ParentComponent, ChildComponent],

bootstrap: [ParentComponent],

})

export class AppModule {}

JavaScript

Conclusion

In this article, we've seen how to pass data from a parent component to a child component in Angular using the @Input() decorator. This feature allows for seamless communication between components and enables dynamic updates whenever the data in the parent component changes.

Angular's component-based architecture, combined with features like @Input(), empowers developers to build modular and maintainable applications by promoting the separation of concerns and reusability of components.

**Using Input and Output decorators**

@Input() And @Output() Decorator In Angular

In this article, I am exploring two very important points, related to the Angular 2 + version, which the part of the Parameter Decorator, and these points are called @Input and @Output decorators. Both are used to transform the data from one component to another component. Or, you can say pass the different types of data form parent to child component and child to parent component.

Or

In a simple way, transform/exchange data between two components.

Introduction

In this article, I am exploring two very important points, related to the Angular 2 + version, which the part of the Parameter Decorator, and these points are called @Input and @Output decorators. Both are used to transform the data from one component to another component. Or, you can say pass the different types of data from parent to child component and child to parent component. Or, in a simple way transform/exchange data between two-component.

Let's explore each, one by one.

**@Input Decorator**

@Input is a decorator to mark a property as an input. @Input is used to define an input property, to achieve component property binding. @Input decorator is used to pass data (property binding) from parent to child component. The component property should be annotated with @Input decorator to act as input property.

Let's explore it practically.

I have created an angular application which is AngApp. I have created two components. They are app components and student components. I will transfer the data from the parent to the child component, using @Input decorator. I am assuming my, app-component is the parent component and student-component is the child component.

To make the parent-child relation, keep the instance (selector of student component) of the student component inside the template URL (app.component.html) of the app component.

Open app.component.html: Inside this file, we keep an instance of the student component.

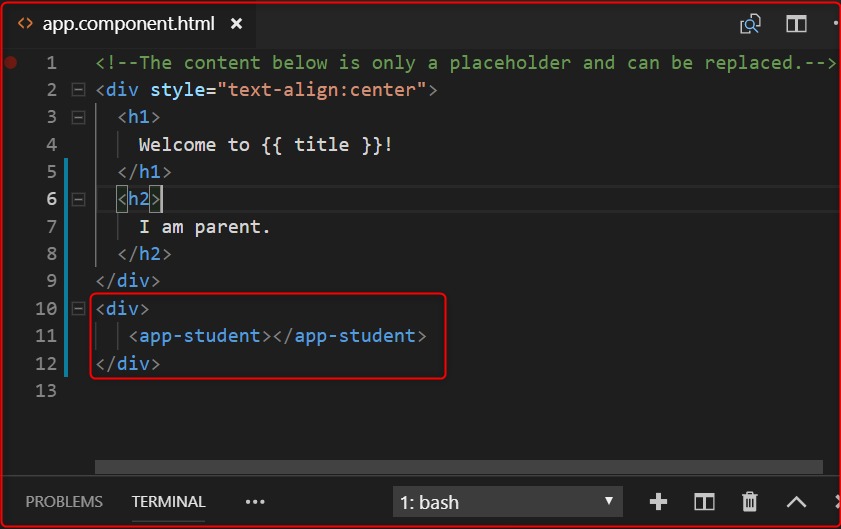
Example

<div>

<app-student></app-student>

</div>

Markup



**Listing 1.0**

In the above image, the selected area is the child component.

Now, we want to send the message from parent to child component.

Let's open the parent component's .ts file (app.component.ts) and declare a variable inside the AppComponent class, to store the message. This message is received by the child component.

myInputMessage: string = "I am the parent component";

TypeScript



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

@Component({

selector: 'app-student',

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

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

})

export class StudentComponent implements OnInit {

@Input() myinputMsg: string;

constructor() { }

ngOnInit() {

console.log(this.myinputMsg);

}

}

TypeScript

In the above image, we have declared a variable( myInputMessage) shown in the selected area.

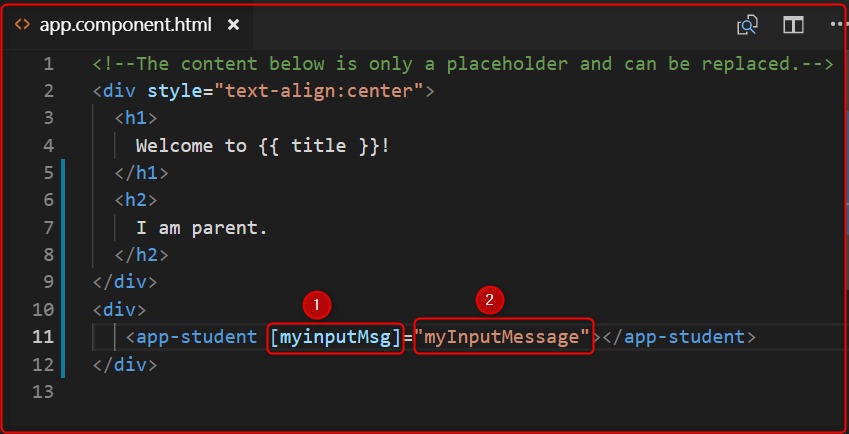
Now, let's open parent component views (app.component.html) and pass this variable inside the child component instance, which is passed inside the parent component.

<div>

<app-student [myinputMsg]="myInputMessage"></app-student>

</div>

Markup



The above image represents 2 points. Let's explain each of the points.

1. Denotes those variables that will be used by the child component (student component) with @Input decorator to fetch the message from the parent component and
2. denotes those variables that are passed the parent component message to the child component.

Now, open the child component's .ts file (student.component.ts) and import the Input decorator, using the myinputMsg variable with @Input decorator and print it inside the constructor or ngOnInit().

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

@Component({

selector: 'app-student',

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

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

})

export class StudentComponent implements OnInit {

@Input() myinputMsg: string;

constructor() { }

ngOnInit() {

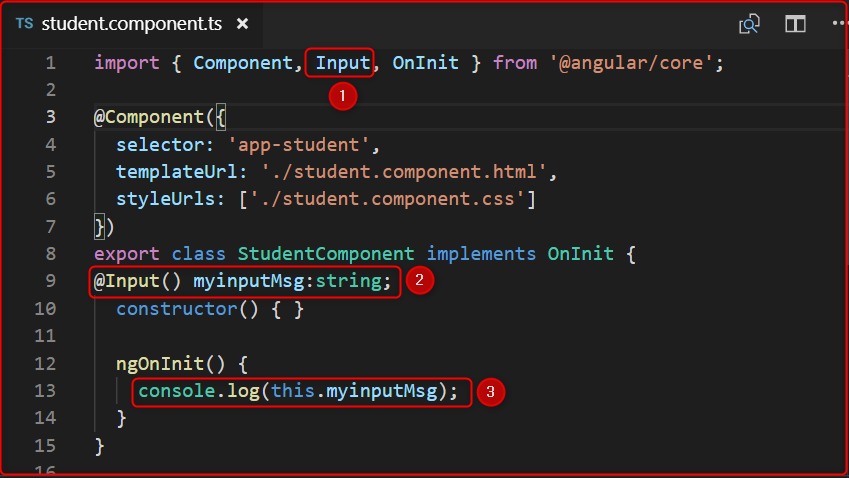
console.log(this.myinputMsg);

}

}

TypeScript

**Output**



**Output**

In image 4, represent 3 points. Let's explain each of the points.

1. First import the Input decorator, which is provided by angular, and the full path is @anuglar/core.
2. Use @Input decorator and declare those variables which are passed by the parent component HTML (app.component.html) file's point 1. When we declare that variable (myinputMsg) with @Input decorator it automatically fetches the value of the parent component variable with the help of @Input decorator.
3. Print the values of this variable inside the constructor or ngOnInit(). We have used inside ngOnInit().

Let's output,



@Output Decorator

@Output decorator is used to pass the data from the child to the parent component. @Output decorator binds a property of a component, to send data from one component to the calling component. @Output binds a property of the type of angular EventEmitter class.

To transfer the data from the child to the parent component, we use the @Output decorator.

Let's Open the child component' .ts file (student.component.ts).

For use the @Output decorator we have to import, two important decorators, they are Output and EventEmitter.

EventEmitter

Use in components with the @Output directive to emit custom events synchronously or asynchronously, and register handlers for those events by subscribing to an instance.

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

TypeScript

Now, create any variable with the @Output decorator.

@Output() myOutput: EventEmitter<string> = new EventEmitter();

TypeScript

Here in the place of string, we can pass different types of DataTypes.

After that create a variable to store and pass the message to the parent component.

outputMessage: string = "I am child component.";

TypeScript

Code

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

@Component({

selector: 'app-student',

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

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

})

export class StudentComponent implements OnInit {

@Input() myinputMsg: string;

@Output() myOutput: EventEmitter<string> = new EventEmitter();

outputMessage: string = "I am child component.";

constructor() { }

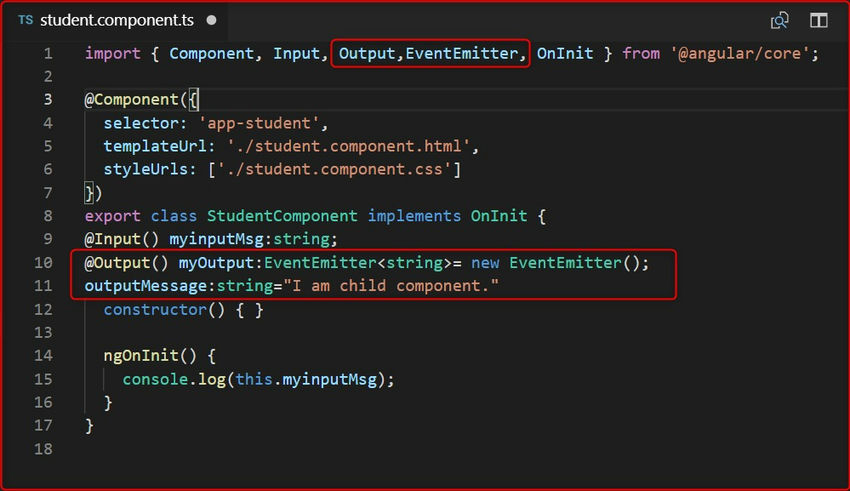
ngOnInit() {

console.log(this.myinputMsg);

}

}

TypeScript



Send the value of the output message, to the parent component. Then we create a button and click on this button. We will send the values to the parent component.

Let's open the child component HTML page and create a button and click the event of this button. We then send the values.

student.component.html.

<button (click)="sendValues"> Send Data </button>

Markup

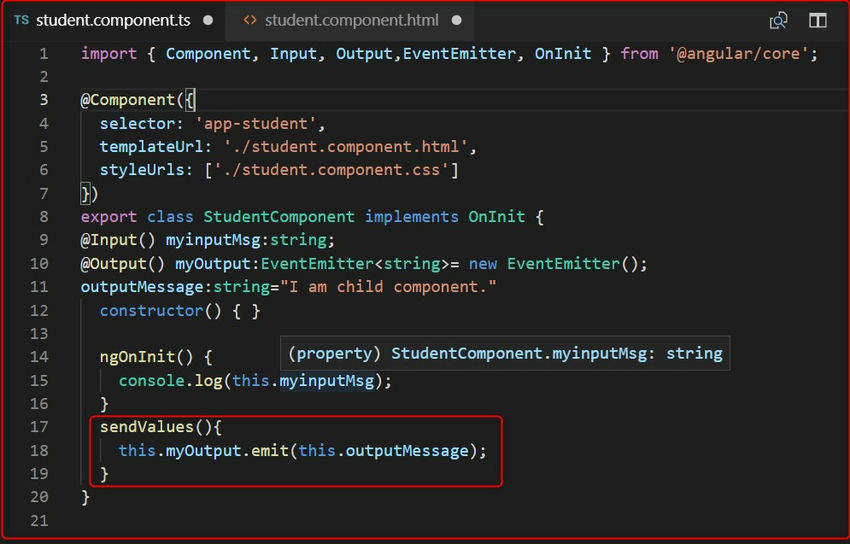
Now fire the click on student.component.ts.

sendValues() {

this.myOutput.emit(this.outputMessage);

}

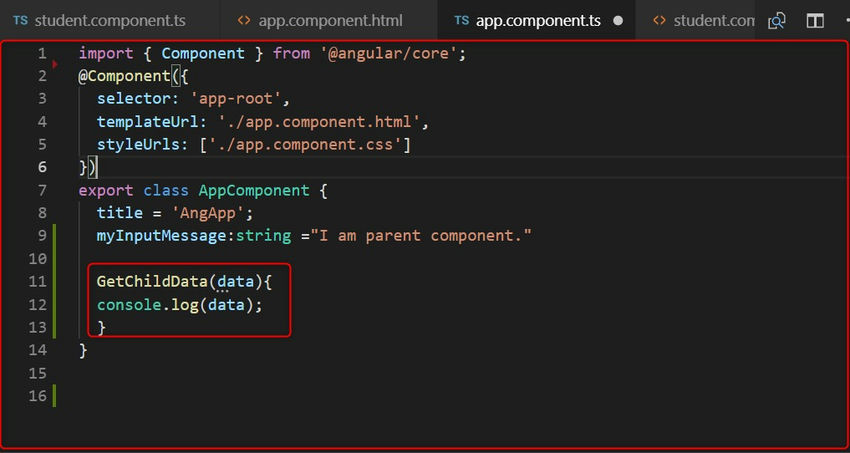
JavaScript



Now, to fetch the value we have to go app.component.html file and use the below code.

<app-student [myinputMsg]="myInputMessage" (myOutput)="GetChildData($event)"></app-student>

Markup



function which is GetChildData() on parent component' .ts file, for fetch the data from child component.

Open the app.component.ts:

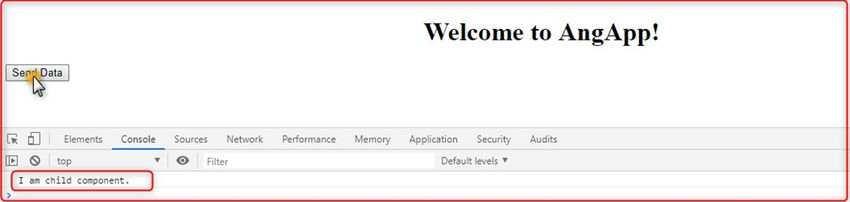
Code:

GetChildData(data){

console.log(data);

}

TypeScript



Conclusion

In this article, we have learned how to pass data from parent to child component and vice versa, and which decorators are more responsible for doing this task, called @Input() and @Output() decorators. I hope it will help you.

Communication Between Components In Angular

Transferring information among various components is a crucial requirement in any application, and Angular is no exception. While this practice is widespread, Angular offers several built-in methods to facilitate it. However, there might be situations where these default approaches fall short, necessitating the implementation of more innovative solutions.

In this session, we will explore a few commonly used methods to communicate between components.

The tools which I have leveraged for the tutorial are given below.

1. Visual Studio Code
2. Node 18.14.0
3. Angular 15

The source code can be downloaded from [GitHub](https://github.com/Prasadnair/CommunicationBetweenComponents)

Input and Output Properties (Parent-Child Communication)

* You can use input properties to pass data from parent to child components.
* Define an input property in the child component using the @Input() decorator, and then bind the property in the parent component.
* Use output properties with events to communicate from the child component back to the parent component.
* Define an output property in the child component using the @Output() decorator and emit events using EventEmitter.

Let us look at the sample code below, which will give you an overview of parent-to-child and child-to-parent communications.

**Parent Component**

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

@Component({

selector: 'app-parent',

template: `

<h1>Parent Component</h1>

<input [(ngModel)]="parentMessage"/>

<p>{{ childMessage }}</p>

<app-child [message]="parentMessage" (notify)="onNotify($event)"></app-child>`

})

export class IOParentComponent {

parentMessage = 'Message from parent';

childMessage ="";

onNotify(message: string) {

this.childMessage =message;

}

}

C#

**Child Component**

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

@Component({

selector: 'app-child',

template: `

<h2>Child Component</h2>

<p>{{ message }}</p>

<input [(ngModel)]="childMessage"/>

<button (click)="notifyParent()">Notify Parent</button>

`

})

export class IOChildComponent {

@Input() message: string ='';

@Output() notify = new EventEmitter<string>();

childMessage: string ='Message from child';

notifyParent() {

this.notify.emit(this.childMessage);

}

}

C#

Services (Cross-Component Communication) using RxJS Observables and Subjects

* Create a service that can be injected into the components needing to communicate.
* The service acts as a shared source of data and functionality.
* Components can subscribe to data streams or invoke methods provided by the service to communicate with each other.

Let us take a look at the example

**Shared Service**

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

import { Subject } from 'rxjs';

@Injectable()

export class DataService {

private dataSubject = new Subject<string>();

setData(data: string) {

this.dataSubject.next(data);

}

getData() {

return this.dataSubject.asObservable();

}

}

C#

**Component 1**

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

import { DataService } from './dataservice.service';

@Component({

selector: 'app-scomponent1',

template: ` <h1>Component 1</h1>

<input type="text" [(ngModel)]="message">

<button (click)="sendMessage()">Send Message</button>

<app-component2>/</app-component2>`

})

export class SComponentOne {

message: string="";

constructor(private dataService: DataService) { }

sendMessage() {

this.dataService.setData(this.message);

}

}

C#

**Component 2**

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

import { DataService } from './dataservice.service';

@Component({

selector: 'app-component2',

template: `

<h1>Component 2</h1>

<p>{{ receivedMessage }}</p>

`

})

export class SComponentTwo {

receivedMessage: string ='';

constructor(private dataService: DataService) { }

ngOnInit() {

this.dataService.getData().subscribe(message => {

this.receivedMessage = message;

});

}

}

C#

The below reference link explains the Mediator and Service Bus approaches for component communication in Angular.

[Component communication in Angular for Senior devs by 'iamprovidence'](https://medium.com/@iamprovidence/component-communication-in-angular-for-senior-devs-4e3c9f0f2c59)

These are some of the commonly used methods for component communication in Angular. The choice of method depends on the specific requirements of your application and the relationship between the components involved.