Angular

4.

**Creating New Angular App**

To create a new workspace and initial starter app:

Run the CLI command ng new and provide the name new- app, as shown here:

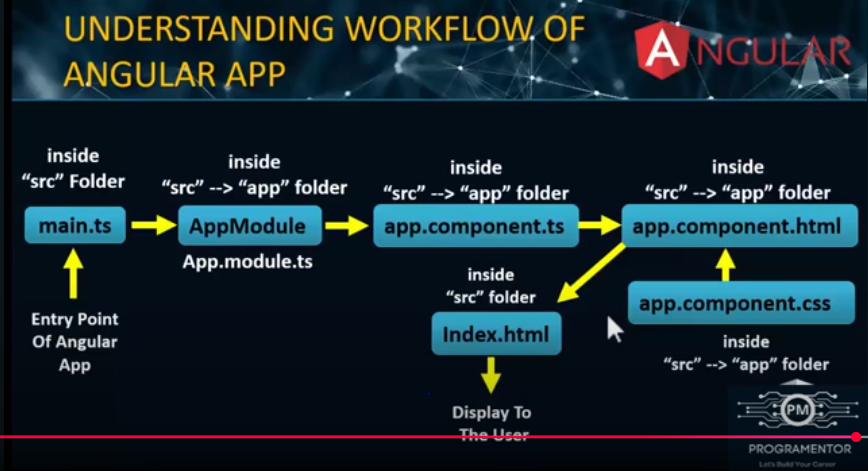
- ng new "new-app"

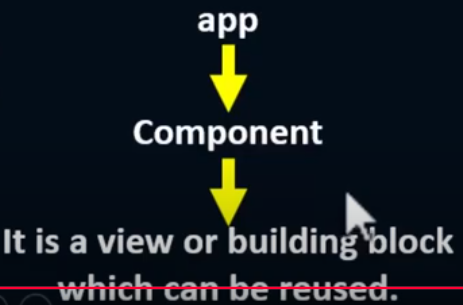
**Run the application**

* The Angular CLI includes a server, for you to build and serve your app locally.
* Navigate to the workspace folder, such as new-app.
* Run the following command
  + ng serve
* The ng serve command launches the server, watches your files, and rebuilds the app as you make changes to those files.
* The --open (or just -o) option automatically opens your browser to http://localhost:4200/

5.

**Flow Of Execution Of Angular App**





In Angular version 15 and above (including Angular 22), standalone components are the default, but you can still create and use traditional components within an NgModule if you explicitly use the --no-standalone option while creating your project. Here’s how to declare components in both scenarios:

**1. Standalone Project (Default for Angular 15+ and newer)**

**Standalone Components**

* In standalone mode, components do not need to be declared in an NgModule. Instead, they are self-contained and use the standalone: true property in the @Component decorator.
* You import them directly into other components or routes.

**Example of a Standalone Component**

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

@Component({

selector: 'app-my-component',

standalone: true, // Mark the component as standalone

template: `<h1>Standalone Component!</h1>`,

styleUrls: ['./my-component.component.css'],

})

export class MyComponent {}

**Traditional NgModule Project (Using --no-standalone)**

If you created your project with --no-standalone, components must be declared in an NgModule as before.

**Declaring a Component**

1. **Generate a Component**

ng generate component my-component --no-standalone

OR (shorter version):

ng g c my-component --no-standalone

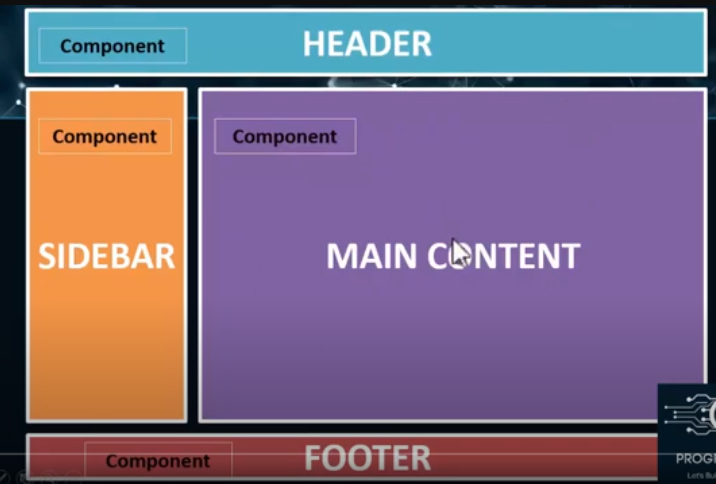
**This automatically:**

* + Creates the component files.
  + Adds the component to the declarations array of the closest NgModule (e.g., AppModule).

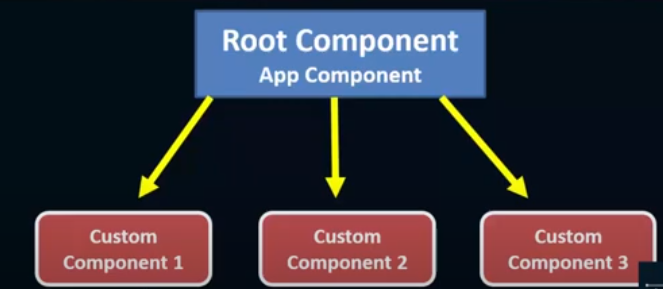
Important Points

* Every Angular App has 1 module file called AppModule / root module.
* Every Angular App has 1 pre-defined component called app / root component.
* Angular App can have one or more modules.
* Modules can have one or more components and services.
* Components have HTML and Class.

**6. Angular Components**

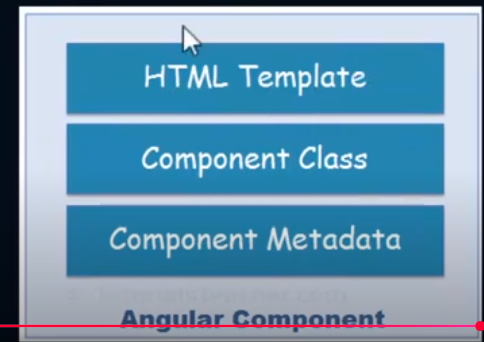


* Angular is a SPA framework, and a view is made of one or more component. An Angular component represents a portion of a view.
* The component is the basic building block of Angular.
* Generally, an interactive web page is made of HTML, CSS, and JavaScript. Angular component is no different.
* Angular Component = HTML Template + Component Class + Component Metadata.
* Components can be re-used.



**Generate Angular Component. using Angular CLI**

* You can create files for a component manually or using the Angular CLI command. Angular CLI reduces the development time. So, let's use Angular CLI to create a new component.
* Use the following CLI command to generate a component.
* ng generate component <component name>
* ng g c <component name>
* All Angular CLI command starts with ng, generate or g is a command, component is an argument and then the new the component.



HTML Template

HTML template is nothing but a regular HTML code with additonal Angular specific syntax to communicate with the component class.

Eg:

<p>{{title}}</p>

<p>{{name}}</p> //Variable

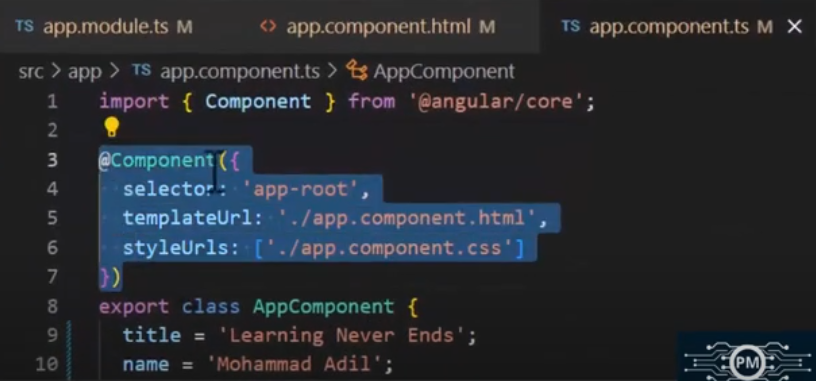
{{helloMessage()}} // Method

Class

Essentially, a component class is a TypeScript class that includes properties and methods. Properties store data and methods include the logic for the component. Eventually, this class will be compiled into JavaScript.

Metadata

Metadata is some extra data for a component used by Angular to execute the component, such as the location of HTML and CSS files of the component, selector, providers, etc.



Component Naming Convention

All the component files in Angular should follow the following format:

<component-name>.component.<file-type>

Example:

- app.component.html

- app.component.css

- app.component.ts

Ways of calling component's

There are 3 ways of calling our components inside app.component.html.

1. By using Custom Tag

Eg: <app-user> </app-user>

2. By using Class selector

Eg: <div class=”app-user”></div>

3. By using attribute

Eg: <div app-user> </div>

**INLINE TEMPLATE & INLINE STYLE**

In some cases, we don't need a separate file of HTML or CSS in our angular component, we want to implement HTML or CSS directly in our "example.component.ts" file.

There are 3 ways of implementing inline template or inline style.

1. We need Separate HTML file but no need of separate CSS file.

ng g c user1 --inline-style

2. We need Separate CSS file but no need of separate HTML file.

ng g c user2 --inline-template

3. We don't need both HTML and CSS files.

ng g c user3 --inline-template --inline-style

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

@Component({

selector: 'app-inline-component',

template: `<h1>Welcome to the Inline Component!</h1>`, // Inline HTML

styles: [`

h1 {

color: blue;

font-size: 24px;

}

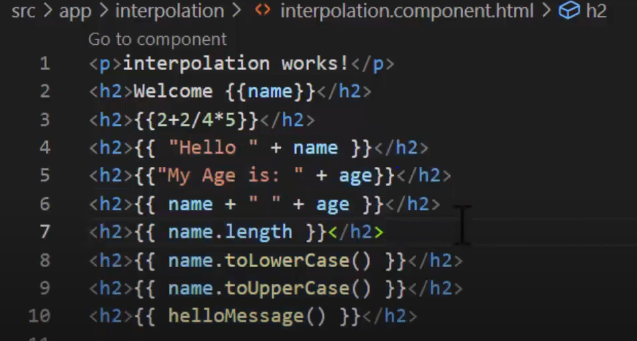
`], // Inline CSS

})

export class InlineComponent {}

Interpolation

* Interpolation allows us to include expressions as part of any string literal, which we use in our HTML.
* The angular evaluates the expressions into a string and replaces it in the original string and updates the view.
* You can use interpolation wherever you use a string literal in the view.
* Angular interpolation is also known by the name string interpolation. Because you incorporate expressions inside another string.
* Interpolation is one-way binding.
* Interpolation is one way as values go from the component to the template. When the component values change, the Angular updates the view. But if the values changes in the view components are not updated.
* The Angular uses the {{ }} (double curly braces) in the template to denote the interpolation. The syntax is as shown below
* {{ templateExpression }}



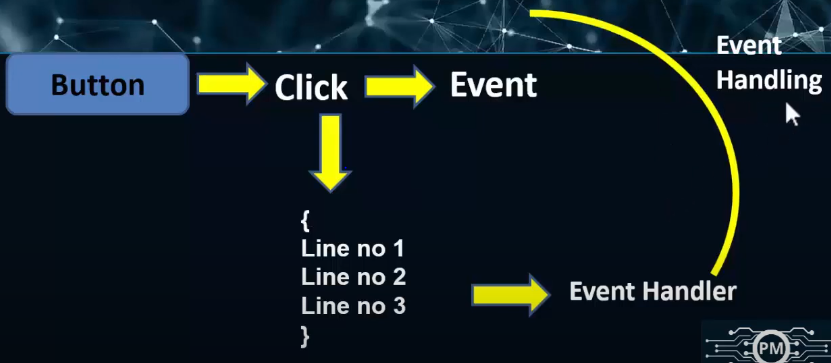
you.cannot make use of the following

* Assignments (=, +=, -=, ...)
* Keywords like new, typeof, instanceof, etc
* Chaining expressions with; or,
* The increment and decrement operators ++ and --
* bitwise operators such as | and &
* We cannot use JavaScript global variables.

**Event Handling or Event Binding In Angular**

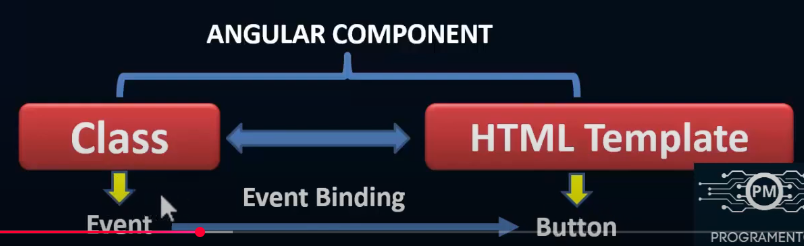
What Are Events ?

* In Web Development, an event is an action or event detected by a program.
* Events can be user actions, such as clicking a mouse button or pressing a key, or they can be events on the system, such as running out of memory.



Event Binding

Event binding lets you listen for and respond to user actions such as keystrokes, mouse movements, clicks, and touches.



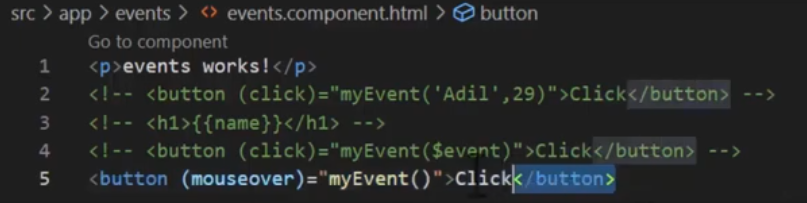
Binding to events

To bind to an event you use the Angular event binding syntax.

This syntax consists of a target event name within parentheses to the left of an equal sign, and a quoted template statement to the right.

Example:

<button (click)="onSave()">Save</button>



**Property Binding**

In Angular, **Property Binding** is a feature that allows you to dynamically set values for HTML element properties or directives using expressions in your component class. It's one of the core building blocks for dynamic and interactive UIs in Angular.

**Syntax**

[property]="expression"

Eg:  
<img [src]="imagePath" alt="Angular Logo">

<button [disabled]="isDisabled">Click Me</button>

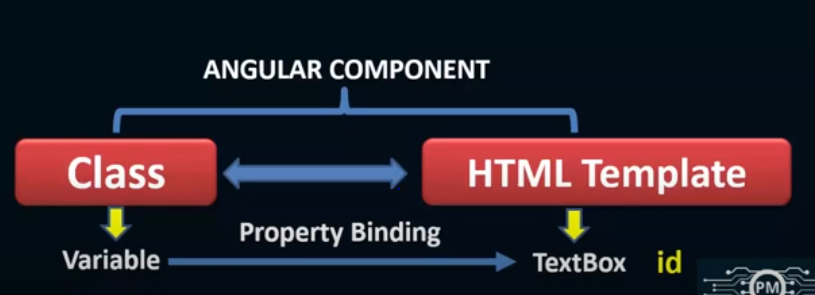
**app.component.ts**

export class AppComponent {

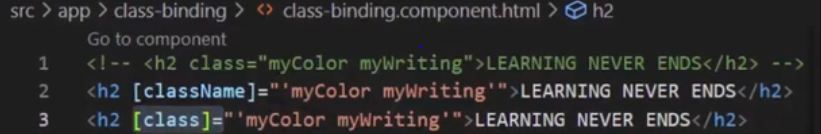
imagePath = 'https://angular.io/assets/images/logos/angular/angular.png'; i

sDisabled = true;

}



**Class Binding   
Class Binding** is a feature that allows you to dynamically add or remove CSS classes to HTML elements based on a condition or value in the component. This is useful for dynamically applying styles or indicating state changes in your application.



**ngClass directive**

The **ngClass directive** in Angular allows you to dynamically add or remove classes on an HTML element based on an expression. It is particularly useful for managing multiple classes conditionally or dynamically in a clean and expressive way.

**Syntax**

You can use ngClass with different types of expressions:

1. **Object Syntax**:

[ngClass]="{ 'class-name': condition, 'other-class-name': otherCondition }"

1. **Array Syntax**:

[ngClass]="['class1', 'class2', dynamicClass]"

1. **String Syntax**:

[ngClass]="'class1 class2 dynamic-class'"

**Style Binding**  
In Angular, **Style Binding** allows you to dynamically set inline styles on an HTML element based on an expression. This is particularly useful for dynamically applying styles to elements at runtime, depending on the state of the component.

**Syntax**

1. **Single Style Binding**:

[style.property]="expression"

* + property is the CSS property (e.g., color, font-size).
  + expression is a value or a component variable that determines the style.

1. **Multiple Styles Binding**:

[style]="styleObject"

* + styleObject is an object where the keys are CSS properties, and the values are the corresponding styles.

Eg:

<div [style.color]="textColor" [style.fontSize.px]="fontSize">

Single Style Binding Example

</div>

export class AppComponent

{ textColor = 'blue'; fontSize = 20; // in pixels }

**Template Reference Variable**

A **Template Reference Variable** in Angular is a variable that you can define in a template to reference a DOM element, component, or directive. It allows you to interact with those elements or components directly within your template, making it useful for scenarios like accessing element properties, calling component methods, or binding data.

**Syntax**

To define a template reference variable, use the # symbol followed by the variable name:

<tag #variableName></tag>

You can then use the variable in the template to reference the associated DOM element, directive, or component.

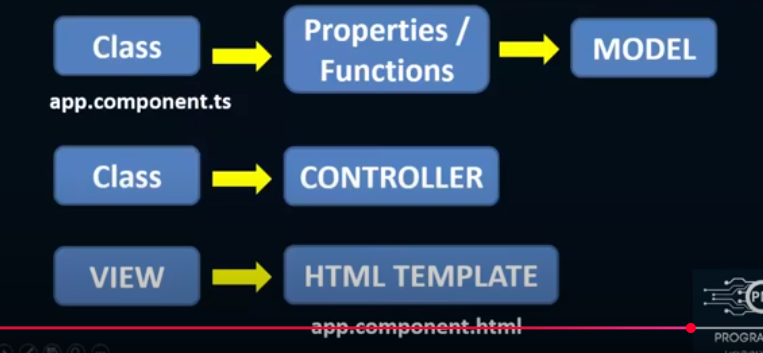


Eg:

<input #myInput type="text" placeholder="Type something">

<button (click)="onButtonClick(myInput.value)">Log Input Value</button>

**Model View Controller In Angular**



**Two Way Data Binding**

Two-way data binding is a technique in programming where changes in the UI (User Interface) automatically update the underlying data model, and changes in the data model automatically reflect in the UI. This synchronization ensures consistency between the view and the model without requiring additional boilerplate code to update them manually.

**Implementation Examples**

**1. In Angular (with ngModel)**

Angular uses two-way data binding with the [(ngModel)] directive:

<input [(ngModel)]="username" placeholder="Enter your name">

<p>Hello, {{ username }}</p>

* Typing in the input updates the username property in the model.
* Changing username in the component updates the input field automatically.



ngModel Directive

* The ngModel directive with (( )) syntax (also known as banana box syntax) syncs values from the UI to a property and vice-versa.
* So, whenever the user changes the value on UI, the corresponding property value will get automatically updated.
* [()] = [] + () where [] binds attribute, and () binds an event.
* The [(ngModel)] syntax is the recommended way of two-way data binding.

**Important Note**

* If you are using ngModel in your angular app then you have to add below line inside app.module.ts file.
* import { FormsModule } from '@angular/forms';

**Structural Directives in Angular**

Structural directives are responsible for the Structure and Layout of the DOM Element. It is used to hide or display the things on the DOM.

* nglf
* nglf-else
* ngSwitch
* ngFor

nglf Directive

* nglf directive is used to display or hide the DOM Element based on the expression value assigned to it. The expression value may be either true or false.
* The nglf Directive is used to add or remove HTML Elements according to the expression.
* The expression must return a Boolean value.
* If the expression is false then the element is removed, otherwise element is inserted.
* The nglf directive does not hide the DOM element. It removes the entire element along with its subtree from the DOM.

Eg:

<div \*ngIf="isVisible">Content is visible</div>

<div \*ngIf="isLoggedIn; else notLoggedIn">Welcome back!</div>

<ng-template #notLoggedIn>

<p>Please log in.</p>

</ng-template>

nglf-else

nglf-else works like a simple If-else statement, wherein if the condition is true then 'If' DOM element is rendered, else the other DOM Element is rendered.

Angular uses ng-template with element selector in order to display the else section on DOM.

ngSwitch Directive

* In Angular, ngSwitch is a structural directive which is used to Add/Remove DOM Element.
* It is similar to switch statement of any other programming language.
* The ngSwitch directive is applied to the container element with a switch expression.

Eg:

<div [ngSwitch]="status">

<p \*ngSwitchCase="'online'">User is online</p>

<p \*ngSwitchCase="'offline'">User is offline</p>

<p \*ngSwitchDefault>User status is unknown</p>

</div>

**ngFor Directive Angular**

The \*ngFor directive in Angular is a structural directive used to dynamically render a list of items based on an iterable, such as an array. It simplifies the process of creating repetitive elements in a template.

**Basic Syntax**

<div \*ngFor="let item of items">

{{ item }}

</div>

* **item**: A variable representing the current element in the iteration.
* **items**: The array or iterable that is being looped through.

**Examples**

**Rendering a Simple List**

<ul>

<li \*ngFor="let fruit of fruits">

{{ fruit }}

</li>

</ul>

In the component:

fruits = ['Apple', 'Banana', 'Cherry'];

**2. Accessing Index, First, Last, or Even/Odd**

You can use special variables provided by ngFor to access metadata about the current iteration.

<ul>

<li \*ngFor="let fruit of fruits; let i = index; let isFirst = first; let isLast = last;">

{{ i + 1 }}. {{ fruit }}

<span \*ngIf="isFirst">(First)</span>

<span \*ngIf="isLast">(Last)</span>

</li>

</ul>

**Nested Loops**

You can use ngFor inside another ngFor for nested structures.

<div \*ngFor="let category of categories">

<h3>{{ category.name }}</h3>

<ul>

<li \*ngFor="let item of category.items">{{ item }}</li>

</ul>

</div>

In the component:

categories = [

{ name: 'Fruits', items: ['Apple', 'Banana', 'Cherry'] },

{ name: 'Vegetables', items: ['Carrot', 'Broccoli', 'Spinach'] },

];

**Pass Data From Parent To Child Component In Angular**

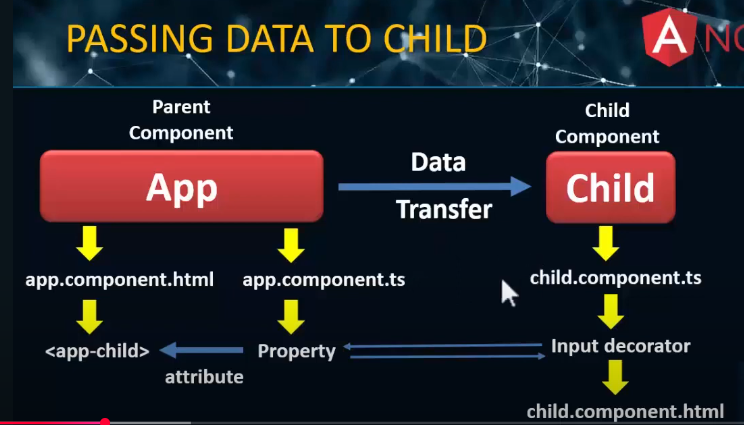
* In Angular, the Parent Component can communicate with the child component by setting its Property.
* To do that the Child component must expose its properties to the parent component.
* The Child Component does this by using the @Input decorator

In the Child Component

* Import the @Input module from @angular/Core Library
* Mark those property, which you need data from the parent as input property using @Input decorator

@Input Decorator

* The @Input Decorator is used to configure the input properties of the component.
* This decorator as also supports change tracking.
* When you mark a property as input property, then the Angular injects values into the component property using Property Binding.
* The Property Binding uses the [] brackets.
* The Binding source is enclosed in double quotes.



**Child Component**

Create a property in the child component and decorate it with @Input().

// child.component.ts

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

@Component({

selector: 'app-child',

template: `<p>Received from parent: {{ data }}</p>`,

})

export class ChildComponent {

@Input() data!: string; // Data from the parent

}

**Parent Component**

Use the child component's selector in the parent component's template and bind a value to the @Input() property using property binding.

// parent.component.ts

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

@Component({

selector: 'app-parent',

template: `

<h1>Parent Component</h1>

<app-child [data]="parentData"></app-child>

`,

})

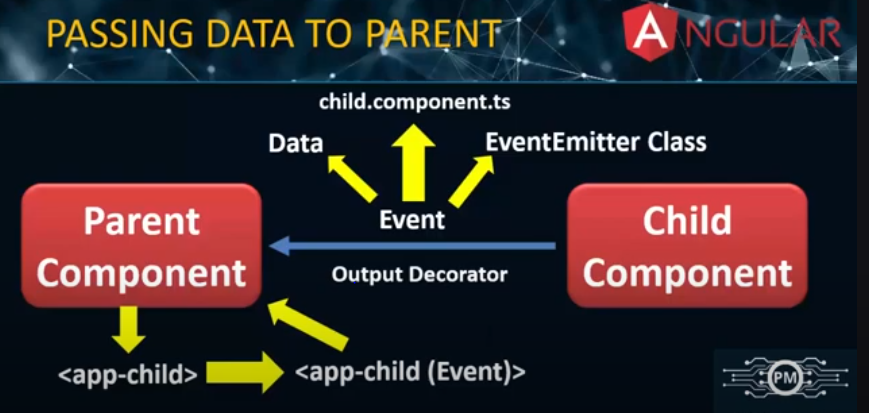
export class ParentComponent {

parentData: string = 'Hello from Parent!';

}

**Pass Data From Child To Parent Component In Angular**

Passing data from a **child to a parent component** in Angular is typically achieved using the @Output() decorator along with an EventEmitter



**1. Create the Parent and Child Components**

**Child Component**

In the child component, declare an EventEmitter with the @Output() decorator and emit an event when you want to pass data to the parent.

// child.component.ts

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

@Component({

selector: 'app-child',

template: `

<button (click)="sendData()">Send Data to Parent</button>

`,

})

export class ChildComponent {

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

sendData() {

const data = 'Hello from Child!';

this.dataEvent.emit(data); // Emit data to parent

}

}

**Parent Component**

In the parent component, listen to the child component's event using event binding ((eventName)="handlerFunction()") and handle the data passed by the child.

// parent.component.ts

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

@Component({

selector: 'app-parent',

template: `

<h1>Parent Component</h1>

<p>Data from Child: {{ receivedData }}</p>

<app-child (dataEvent)="handleData($event)"></app-child>

`,

})

export class ParentComponent {

receivedData: string = '';

handleData(data: string) {

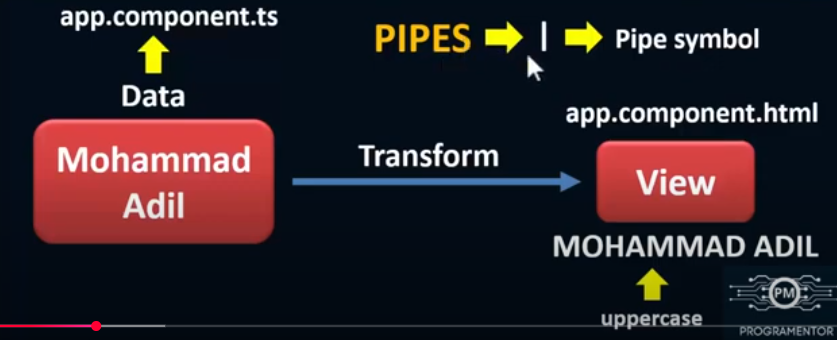
this.receivedData = data; // Handle data received from child

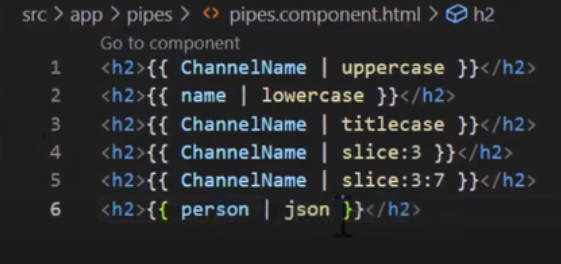
}

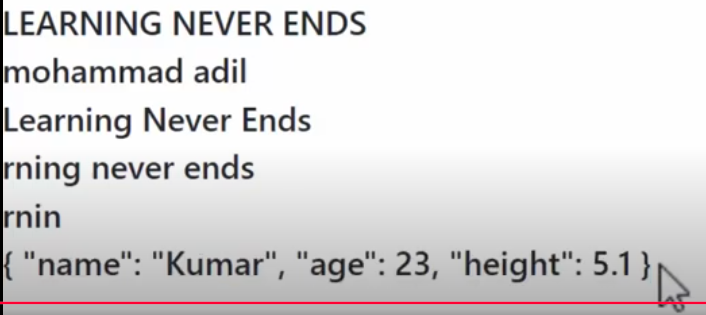
}

**Pipes In Angular**

* In Angular 1, filters are used which are later called Pipes onwards Angular2.
* In Angular 14, it is known as pipe and used to transform data.
* It is denoted by symbol |.
* Angular Pipes allow us to transform data before displaying them in the view.
* The character is used to transform data. Following is the syntax for the same
* {{ Welcome to Angular | lowercase}}
* It takes integers, strings, arrays, and date as input separated with | to be converted in the format as required and display the same in the browser.







**Custom Pipes In Angular**

**Creating Custom Pipes**

* Angular makes provision to create custom pipes that convert the data in the format that you desire.
* Angular Pipes are TypeScript classes with the @Pipe decorator.
* The decorator has a name property in its metadata that specifies the Pipe and how and where it is used.

Command to create custom pipe

ng g p pipe\_name

Best practice

ng g p folder/pipe\_name

**Angular Service**

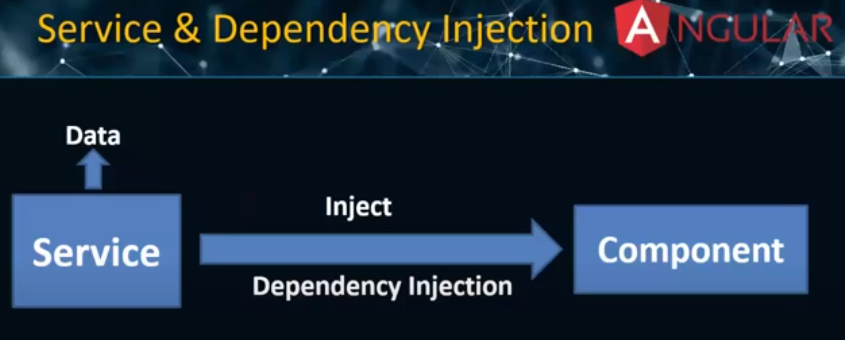
Service is nothing its just a class with specific purpose.

-Data Sharing Across multiple components.

-It is used to apply application logic

- We can use Services for external interactions such as connecting to a database.

* The best solution is to write services and inject in application where we need it.
* Services are usually implemented through dependency injection.



Naming Convention For Service

FileName.service.ts

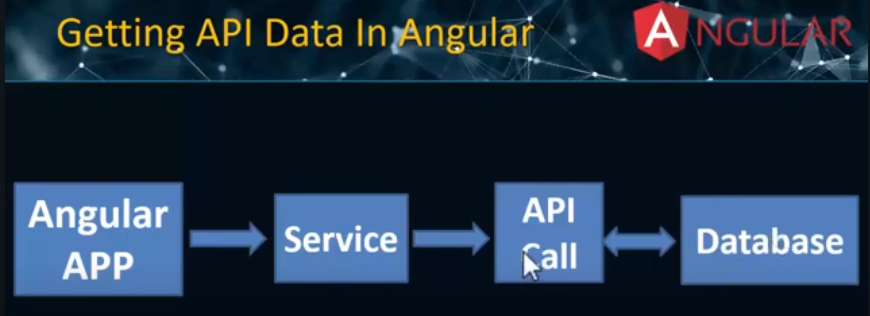
Example

Students.service.ts

Command to create Service

ng g s Service\_name

**Calling API In Angular**





**Angular Forms**

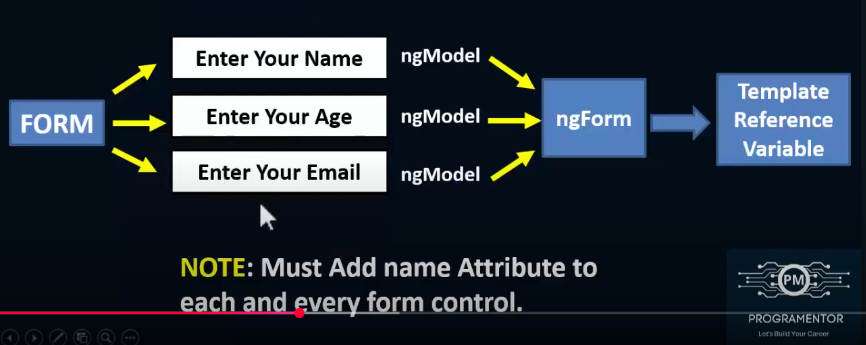
* Angular forms are used to handle user's input.
* We can use Angular form in our application to enable users to login, to update profile, to enter information, and to perform many other data-entry tasks.

In Angular, there are 2 approaches to handle user's input through forms:

1. Template-driven forms

2. Reactive forms (Model Driven Forms)

Template Driven Forms

* Template-driven forms are best if you want to add a simple form to your application. For example: login, contact, signup form.
* Template-driven forms are easy to use in the application but they are not as scalable (expandable / upgradable) as Reactive forms.
* Template-driven forms are mainly used if your application's requires a very basic form and logic. It can easily be managed in a HTML template.
* Template driven forms is created using directives in the template. Example: ngForm, ngModel
* Basic HTML validations can be used to validate the form fields. Example: required, maxlength, minlength, pattern.
* In the case of custom validations, directives can be used.
* Template-driven forms are less explicit.
* They are created in HTML template.
* It works like Asynchronously  
    
    
  

Useful Properties

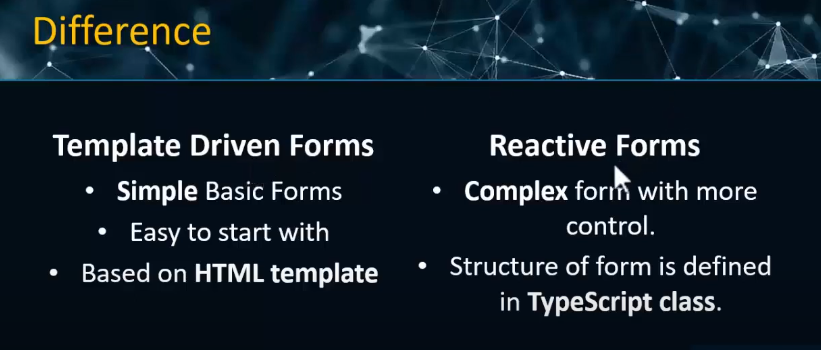
* touched
* untouched
* valid
* invalid
* dirty
* pristine

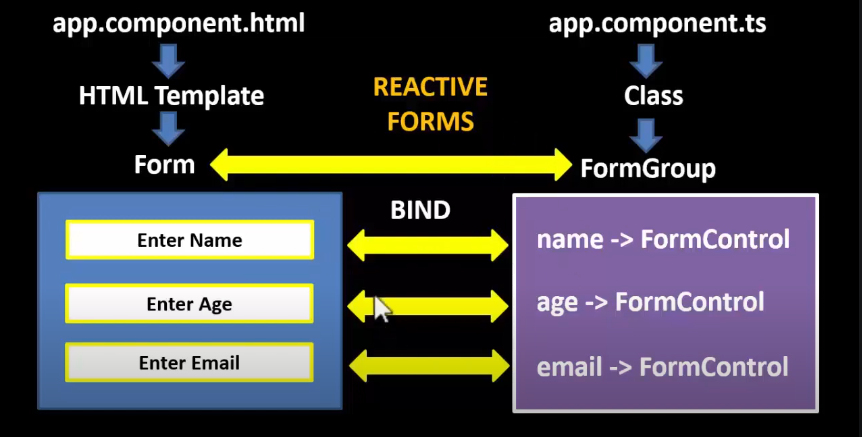
Reactive Forms

* Reactive forms also known as Model-driven forms
* Reactive forms were introduced with Angular 2
* Reactive forms are more robust.
* They are created in component class.
* Reactive forms are more scalable, reusable, and testable.
* They are most preferred to use if forms are a key part of your application.
* Reactive forms are more explicit (manual).
* It works like Synchronously
* Reactive forms are code-driven, unlike the template-driven approach.
* Reactive forms eliminate the anti-pattern of updating the data model via Two-Way data binding.

Here are some of the advantages of reactive forms:

* Using custom validators
* Changing validation dynamically
* Dynamically adding form fields





Built-in Validations

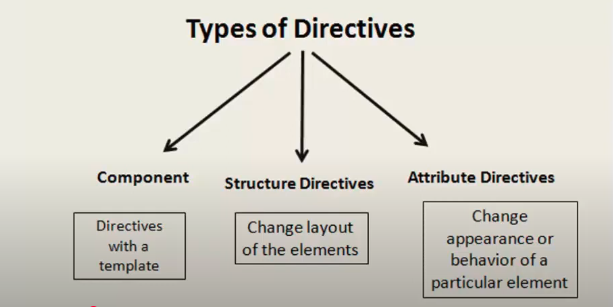
* Required
* MinLength and MaxLength
* Min
* Max
* Email
* Pattern
* Error Messages
* Red Border Color Of Input Box
* Getter Function
* Disabled Button

Directives In Angular

* In Angular, Directives are defined as classes that can add new behavior to the elements in the template or modify existing behavior.
* The purpose of Directives in Angular is to maneuver the DOM, be it by adding new elements to DOM or removing elements and even changing the appearance of the DOM elements.

What Is Meant By Directives in Angular?

* Directives are meant to be a function that executes when found in the DOM by the Angular compiler to extend the power of the HTML with new syntax.
* Directives have a name and can be predefined or custom-defined so that they can be called anything.



| **Type** | **Description** | **Example** |
| --- | --- | --- |
| **1️.Component Directive** | A directive with a template (basically an Angular component). | @Component({ selector: 'app-my-component' }) |
| **2️.Structural Directive** | Modifies the DOM structure by adding or removing elements. | \*ngIf, \*ngFor, \*ngSwitch |
| **3️.Attribute Directive** | Modifies the behavior or appearance of an element. | ngClass, ngStyle, Custom Directives |

Command to create Directives

ng g directives shared/CustomDirectives

Creating Our Own Attribute Directive

Custom Directive is very similar to creating the Angular component. The custom directive is created using the @Directive decorator to replace the @component decorator.

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

@Directive({

selector: '[appCustomDirective]'

})

export class CustomDirectiveDirective {

constructor(private el:ElementRef) {

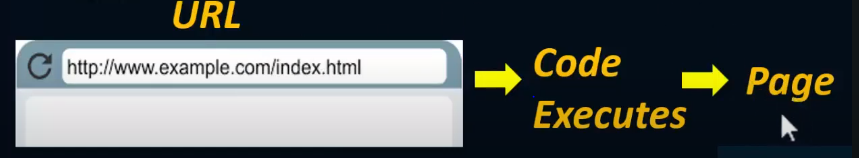
this.el.nativeElement.style.color="red";

}

}

**Routing**

Routing is the mechanism by which requests (as specified by a URL and HTTP method) are routed to the code that handles them.



Angular Router

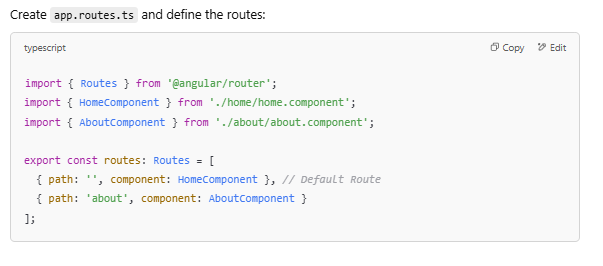
* The Angular router is an essential element of the Angular platform.
* It allows developers to build Single Page Applications with multiple states and views using routes and components and allows client side navigation and routing between the various components.
* It's built and maintained by the core team behind Angular development and it's contained in the @angular/router package
* If you want to navigate to different pages in your application, but you also want the application to be a SPA (Single Page Application), with no page reloading then you can use Angular Router.

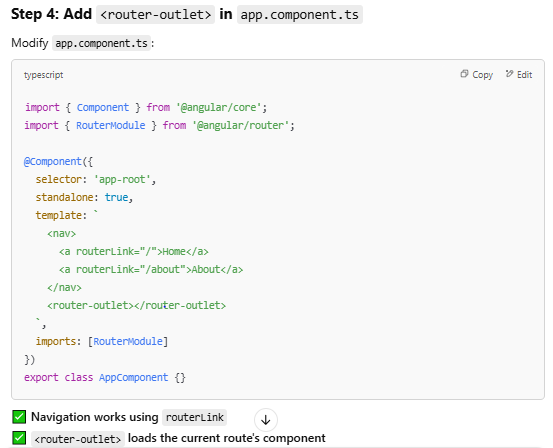
Features In Angular Router

* The Components, routes and paths,
* The router outlet,
* The route matching strategies,
* Route parameters,
* Query parameters,
* Route guards,
* Route resolvers,
* The routerLink directive (replaces the href attribute),
* Auxiliary routes,
* Primary and secondary router outlets.

The Router-Outlet is a directive exported by RouterModule and acts as a placeholder that indicates to the router where it needs to insert the matched component(s).

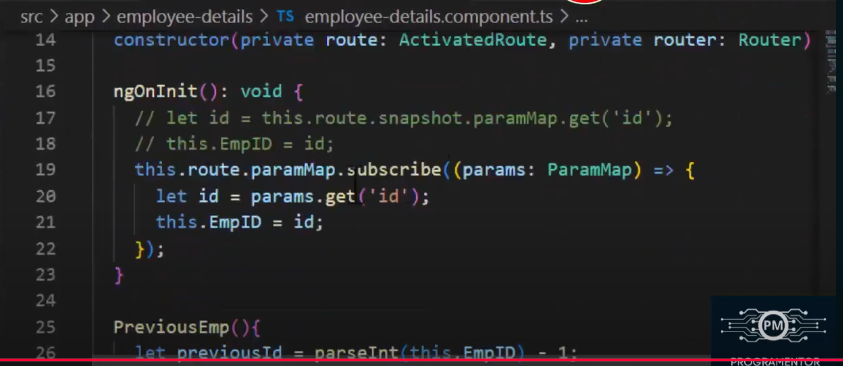
A collection of routes defines the router configuration which is an instance of Routes.





paramMap:

* paramMap is the method of Activated Route and it will return an observable and it will give data only if we subscribe to it
* The argument in the subscribe method is an arrow function.
* The observable provides the parameter which we strongly typed to paramMap.
* paramMap is coming from Router package.
* We should have imported the Router package.



**Optional Route Parameters in Angular 🚀**

* Optional route parameters allow passing additional **optional** data in the URL.
* Unlike required parameters (/user/:id), optional parameters are **not mandatory**.
* They are typically passed using **query parameters** (?key=value) or **matrix parameters** (;key=value).

**Absolute vs. Relative Paths in Angular Routing 🚀**

Angular provides two ways to define navigation paths:

* **Absolute Paths** (Start from the root /)
* **Relative Paths** (Start from the current route)

**1️.Absolute Path in Angular**

**🔹 Definition**

* An **absolute path** starts from the **root of the application** (/).
* It **ignores** the current route and navigates directly from the app’s base URL.

**✅ Example: Absolute Navigation**

this.router.navigate(['/dashboard']);

✅ **Always navigates to /dashboard, regardless of the current route.**

**2️.Relative Path in Angular**

**🔹 Definition**

* A **relative path** is based on the **current route**.
* Useful for navigating **within the same feature/module**.

**✅ Example: Relative Navigation**

Assume the **current route** is /products:

this.router.navigate(['details'], { relativeTo: this.route });

✅ **Navigates to /products/details instead of /details.**

**Child Routes in Angular 🚀**

**Child Routes** allow you to create **nested routes** inside a parent route. This is useful for structuring modules, dashboards, and feature-based navigation.

**Why Use Child Routes?**

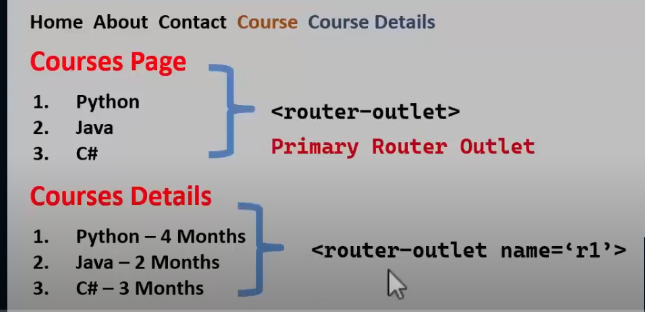
* Organizes related views under a parent route.
* Helps in creating **nested views** (e.g., /products/list, /products/details/1).
* Improves code modularity by grouping related routes.

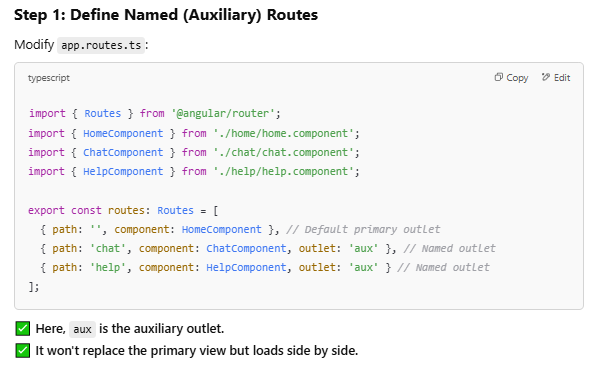


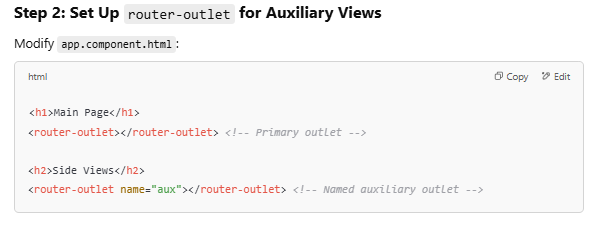
**Auxiliary Routes (Named Outlets) in Angular 🚀**

**What are Auxiliary Routes?**

* **Auxiliary routes** allow multiple views to be displayed **at the same time** using **named outlets**.
* Useful for **sidebars, chat windows, notifications, or independent sections**.
* Helps in creating **complex layouts** with multiple router-outlets.

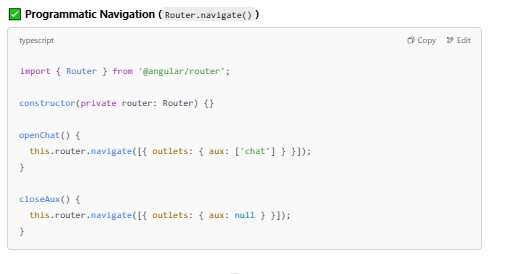












ng-content

* The ng-content is used when we want to insert the content dynamically inside the component that helps to increase component reusability.
* Using ng-content we can pass content inside the component selector and when angular parses that content that appears at the place of ng-content.

Content Projection

* Content projection is a pattern in which you insert, or project, the content you want to use inside another component.
* For example, you could have a Card component that accepts content provided by another componer

Types Of Content Project

1. Single-slot content projection

-With this type of content projection, a component accepts content from a single source.

2. Multi-slot content projection

-In this scenario, a component accepts content from multiple sources.

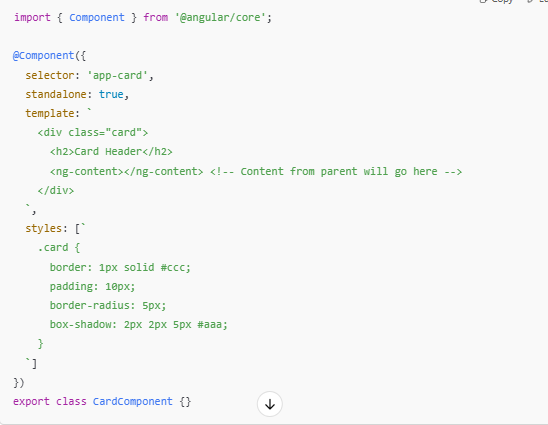
3. Conditional content projection

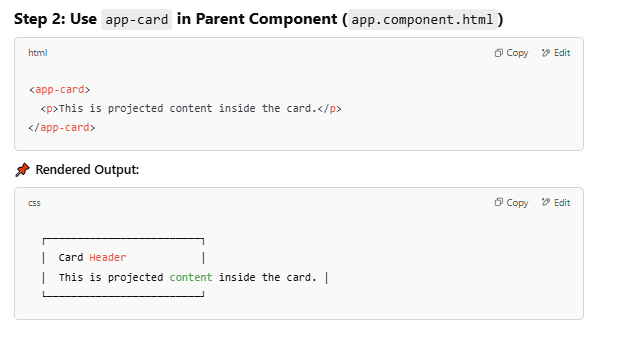
- Components that use conditional content projection render content only when specific conditions are me

**Single-Slot Content Projection (ng-content)**

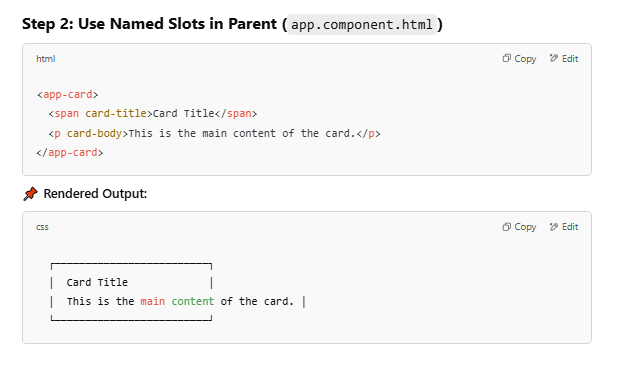
**✅ Example: Parent Passes Content to Child**

**Step 1: Create a Child Component (card.component.ts)**









Ng-Template

* ng-template is a virtual element and its contents are displayed only when needed (based on conditions).
* ng-template should be used along with structural directives like [nglf], [ngFor], [NgSwitch] or custom structural directives.
* ng-template never meant to be used like other HTML elements. It's an internal implementation of Angular's structural directives.
* When you use a structural directive in Angular we will add a prefix asterisk(\*) before the directive name. This asterisk is short hand notation for ng-template.
* Whenever Angular encounter with the asterisk (\*) symbol, we are informing Angular saying that it is a structural directive and Angular will convert directive attribute to ng-template element
* ng-template is not exactly a true web element. When we compile our code, we will not see a ng-template tag in HTML DOM.
* Angular will evaluate the ng-template element to convert it into a comment section in HTMLDOM.



