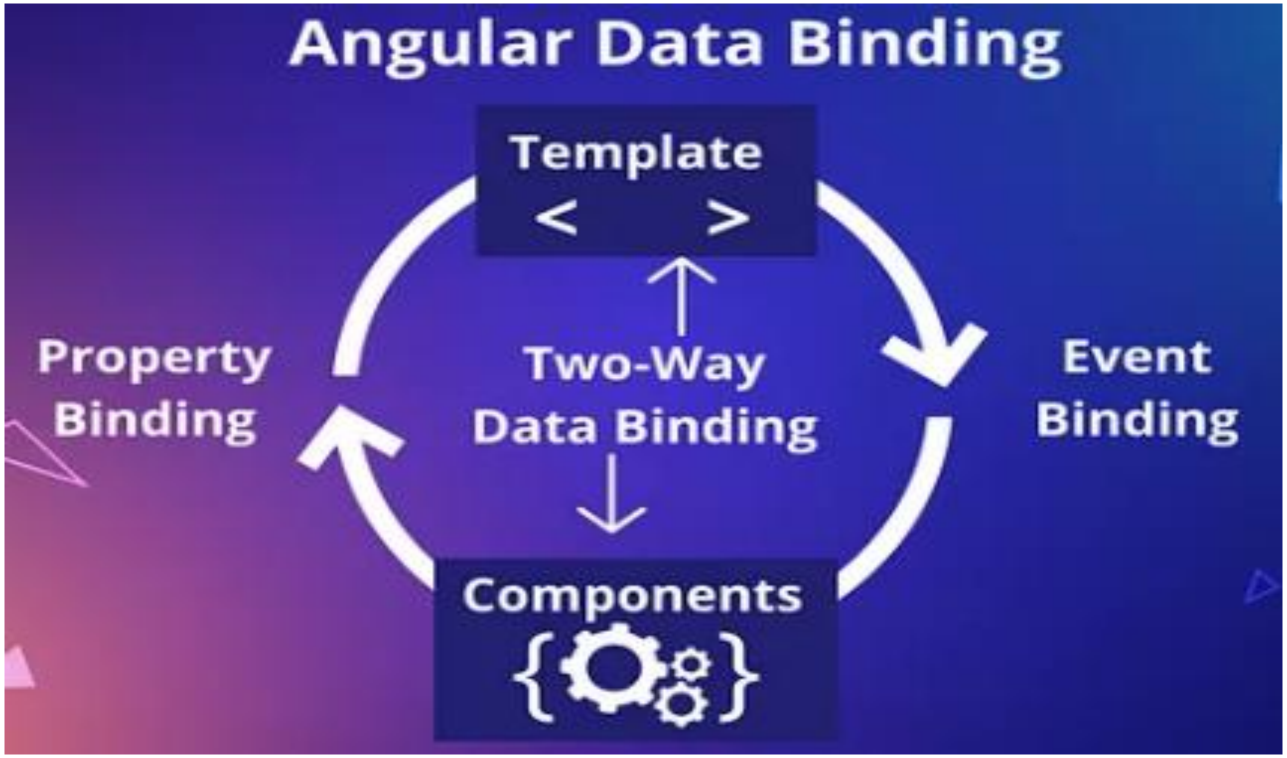
Advanced Application Development (CSE-214)

week-4 : Data binding, Angular Component Lifecycle Hook, Routing

Dr. Alper ÖZCAN
Akdeniz University
alper.ozcan@gmail.com

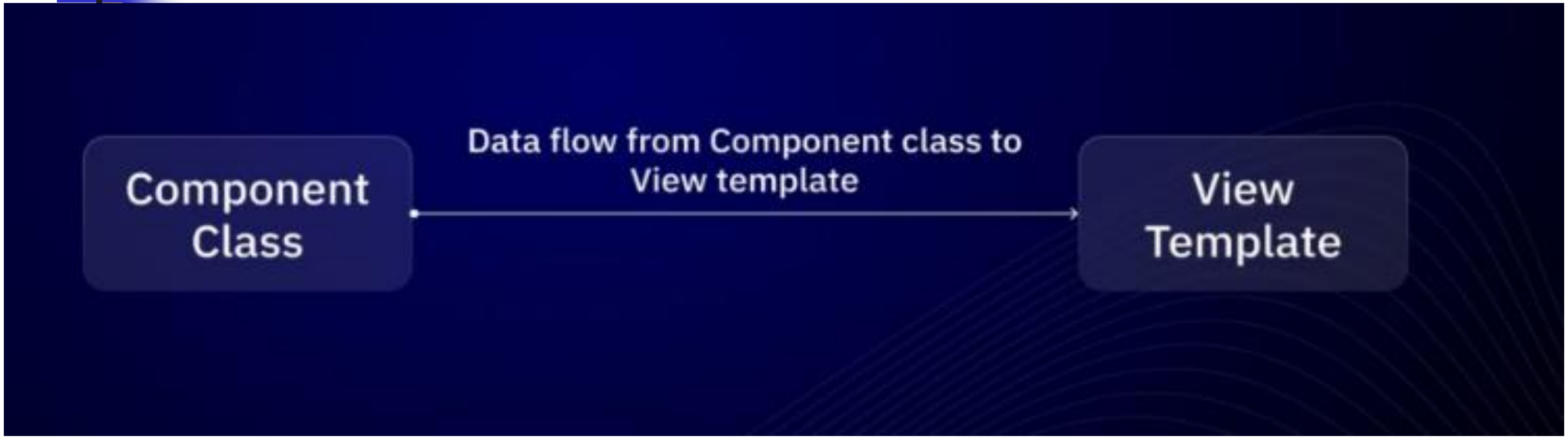
Data Binding in Angular: Connecting Components and Templates



Data binding is a fundamental concept in Angular that allows you to establish a connection between your **component's data** and its **template**.

Data binding enables **dynamic interaction** between your **UI (views)** and the **data layer (models)** of your application.

1.Interpolation: {{ }}

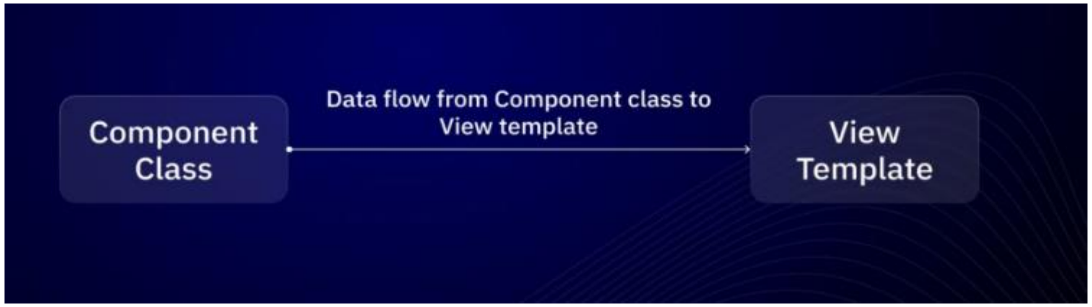


Interpolation is a type of Angular **one-way data binding** that enables the **insertion of a component's property value into the view template**.

To achieve Interpolation binding, you must use double curly braces **{{}}**

Any data or **expressions enclosed** within these curly braces will be **automatically converted** into a **string** for display **in the view**.

Interpolation: {{ }} Example



typescript

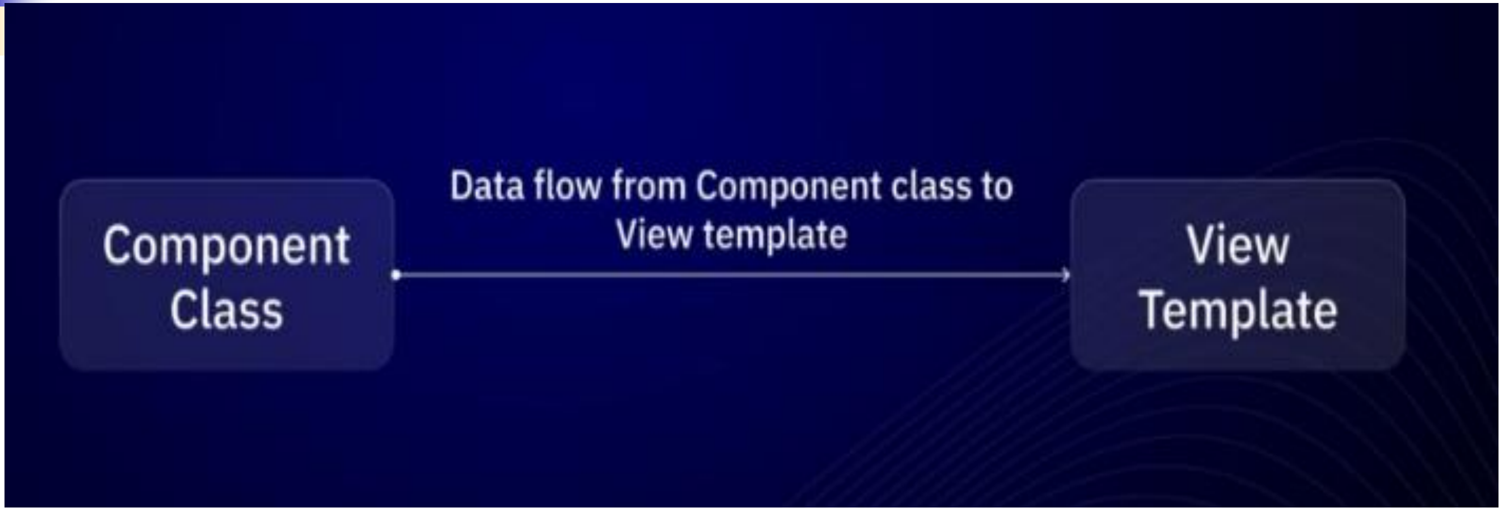
```typescript
export class AppComponent {
  title = "Angular Data Binding";
}
```

html

```html
<h1>{{ title }}</h1>
```

The title property from the component is injected into the template using {{ title }}

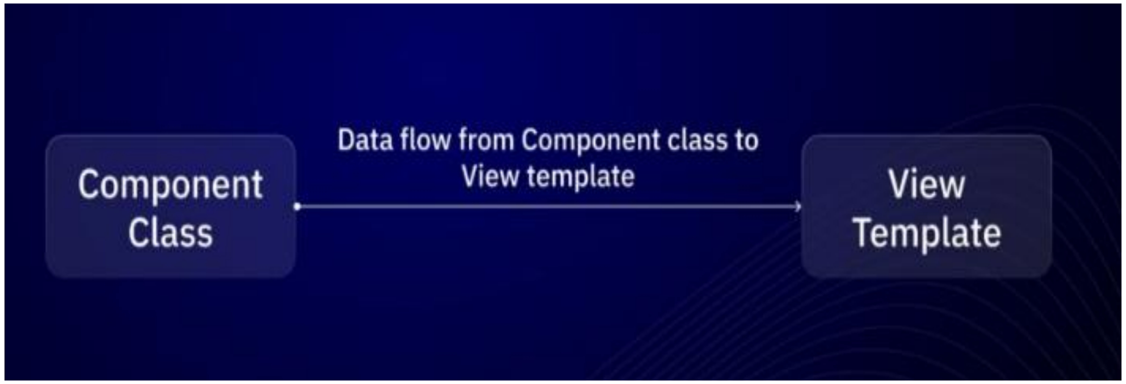
2.Property Binding: [property]="value"



Property binding is a form of Angular **one-way data binding** that allows you to **assign the value of an HTML element's property** based on a **component property**.

You can **dynamically associate** a **component property** with an **HTML element's property**, ensuring that changes in the **component property** are reflected in the **HTML element's property value**

Property Binding: [property]="value" example 1



typescript

```typescript
export class AppComponent {
  isDisabled = true;
}
```

html

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

The [disabled] property is bound to isDisabled, dynamically controlling whether the button is disabled.

Property Binding: [property]="value" example 2

typescript

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

@Component({
  selector: 'app-child',
  template: `<p>The message is: {{ message }}</p>`,
})
export class ChildComponent {
  @Input() message: string = ''; // Custom property for property binding
}
```

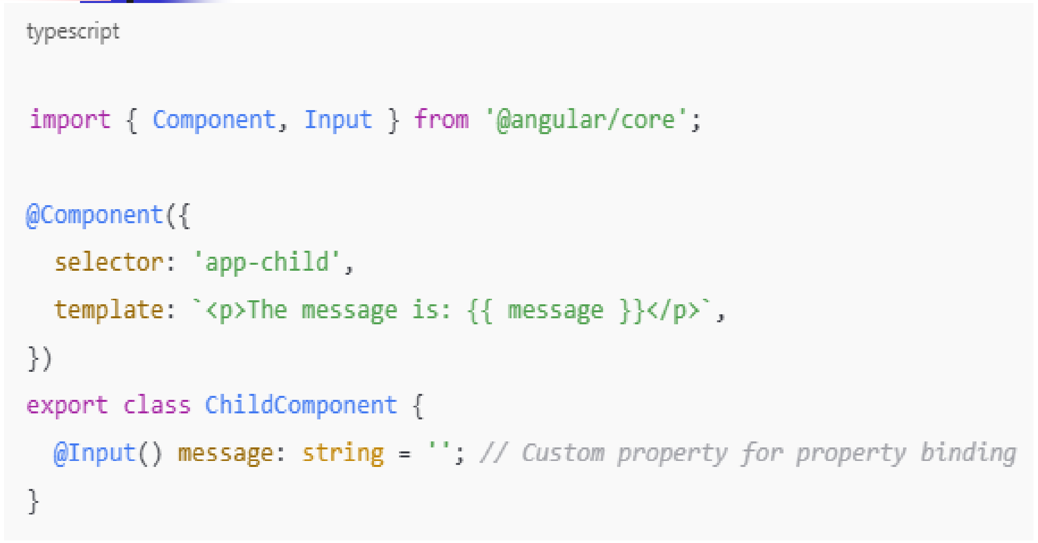
typescript

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

@Component({
  selector: 'app-parent',
  template: `
    <h2>Parent Component</h2>
    <app-child [message]="parentMessage"></app-child>
  `,
})
export class ParentComponent {
  parentMessage = 'Hello from Parent!';
}
```

@Input() makes message a bindable property in the child component.

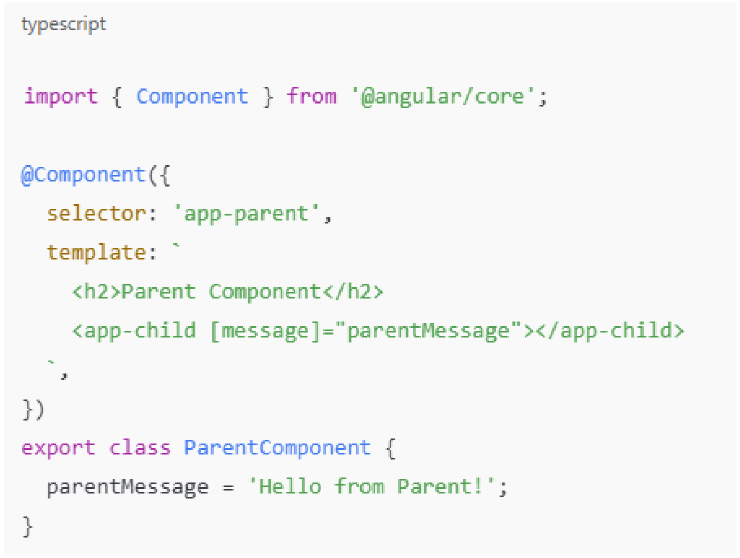
The **[message]** property binding passes **parentMessage** from the **parent to the child**.

Property Binding: [property]="value" example 2



typescript

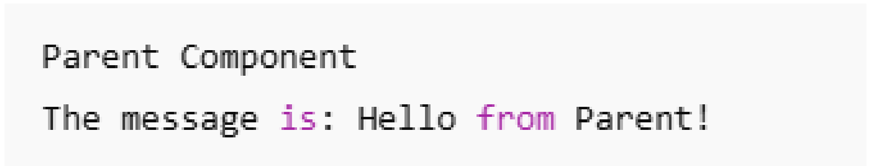
```typescript
import { Component, Input } from '@angular/core';

@Component({
  selector: 'app-child',
  template: `<p>The message is: {{ message }}</p>`,
})
export class ChildComponent {
  @Input() message: string = ''; // Custom property for property binding
}
```



typescript

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

@Component({
  selector: 'app-parent',
  template: `
    <h2>Parent Component</h2>
    <app-child [message]="parentMessage"></app-child>
  `,
})
export class ParentComponent {
  parentMessage = 'Hello from Parent!';
}
```

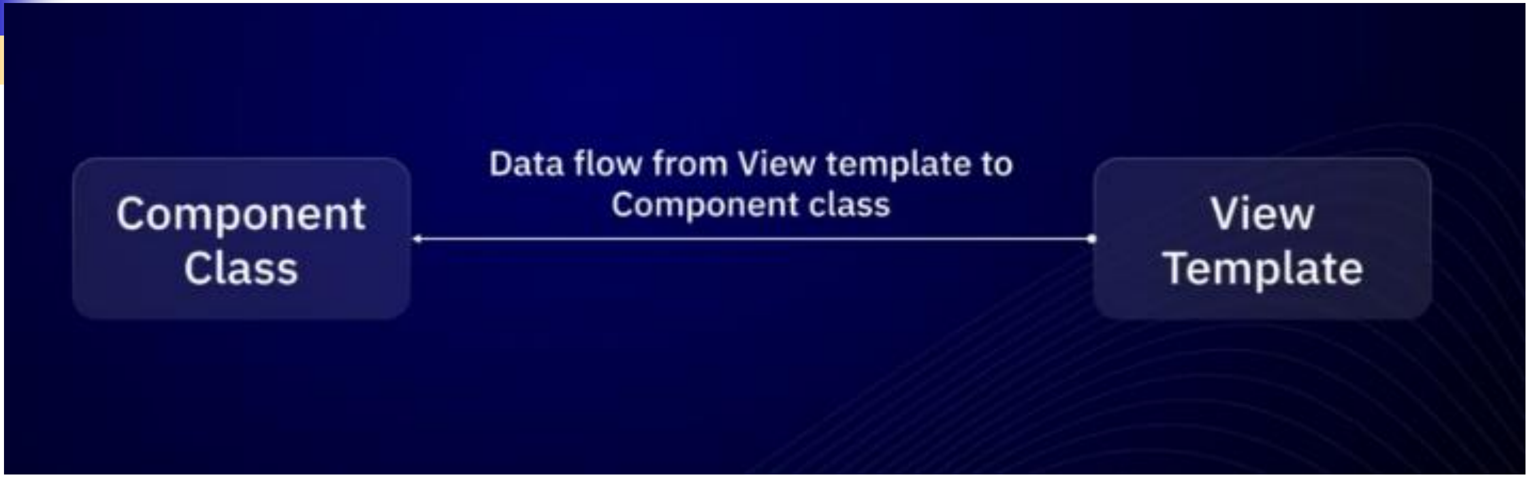
Expected Output :



```
Parent Component
The message is: Hello from Parent!
```

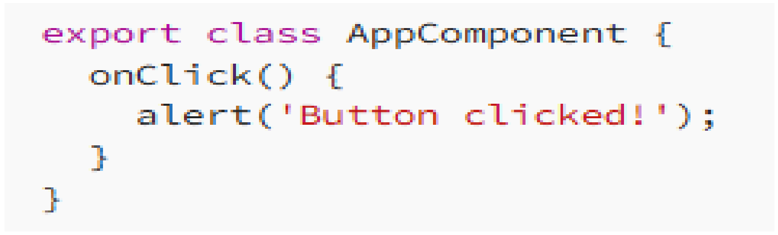
1. **Custom property (@Input())** in **child components** allows **dynamic property binding.**
2. Property binding **([message]="parentMessage")** ensures data flow from **parent to child.**
3. The child component dynamically **updates whenever the parent's property changes.**

3. Event Binding: (event)="handler"

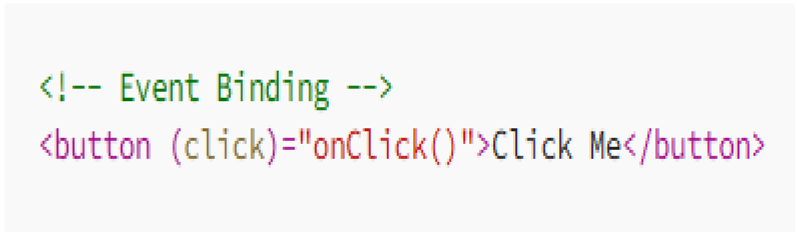


Event binding enables the **connection** of **component methods** to **various user events such as clicks, hovers, and touches**. This creates a **one-way flow** from the **view** to the **component**.

This allows us to update the component’s model and perform necessary validations, providing a seamless and responsive user experience.

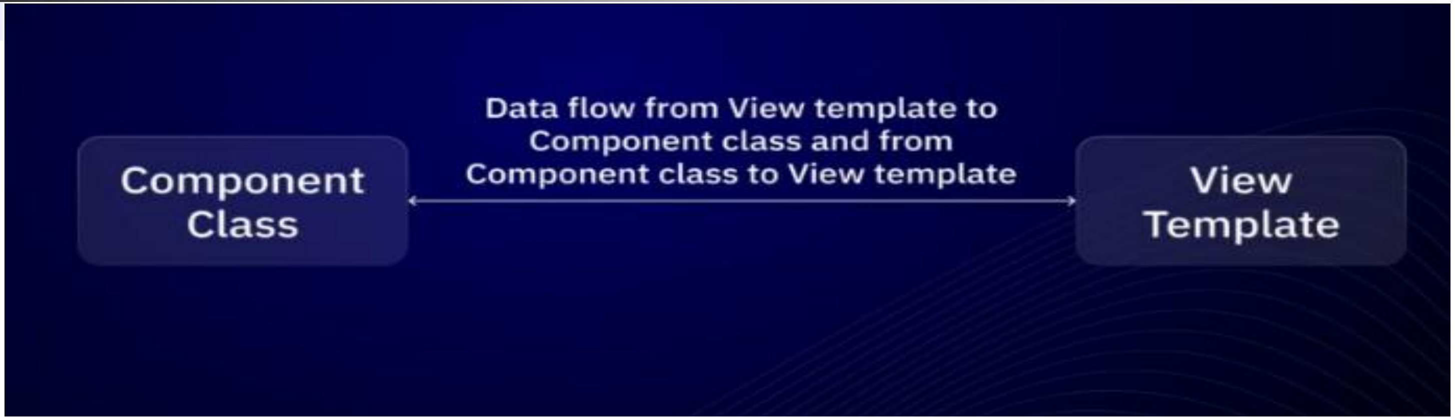


```
export class AppComponent {
  onClick() {
    alert('Button clicked!');
  }
}
```



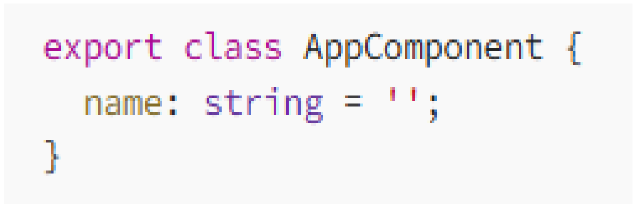
```
<!-- Event Binding -->
<button (click)="onClick()">Click Me</button>
```

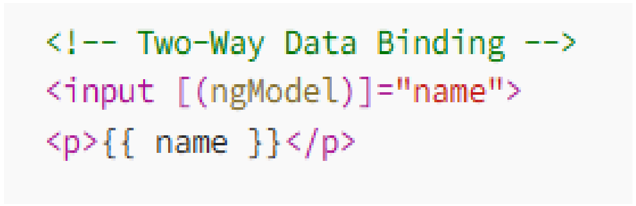
4. Two-Way Binding: [(ngModel)]="property"


Component Class
Data flow from View template to Component class and from Component class to View template
View Template

Two-way data binding in Angular is a powerful feature that combines both **property binding** and **event binding**. With **two-way binding**, you can establish a **connection** between a **component property** and an **input element**, enabling real-time **synchronization** between the two.

You can use the **"ngModel"** directive. **"ngModel"** directive (1) binds the component property to the input element's value, (2) listens for any changes to the input element, updating the component property


```
export class AppComponent {
  name: string = '';
}
```


```
<!-- Two-Way Data Binding -->
<input [(ngModel)]="name">
<p>{{ name }}</p>
```

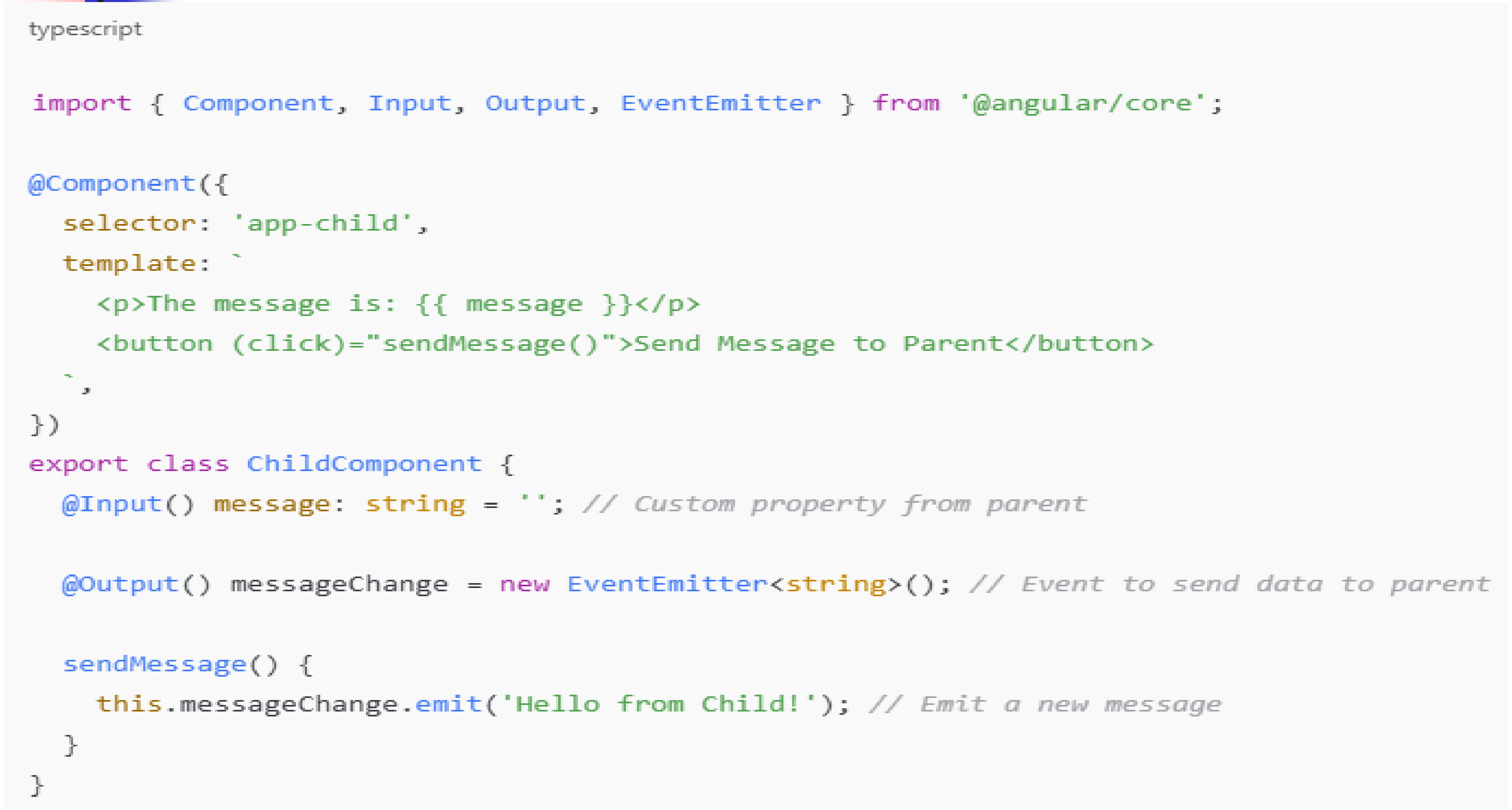
Two-Way Binding example

Extending Property Binding with **Event Binding** for **Two-Way Communication**

Modify our example so that the **child component** can send data back to the **parent component** using **event binding** and **@Output()**

Use **@Output()** and **EventEmitter** to send data to **the parent** when the button is clicked.

Two-Way Binding example

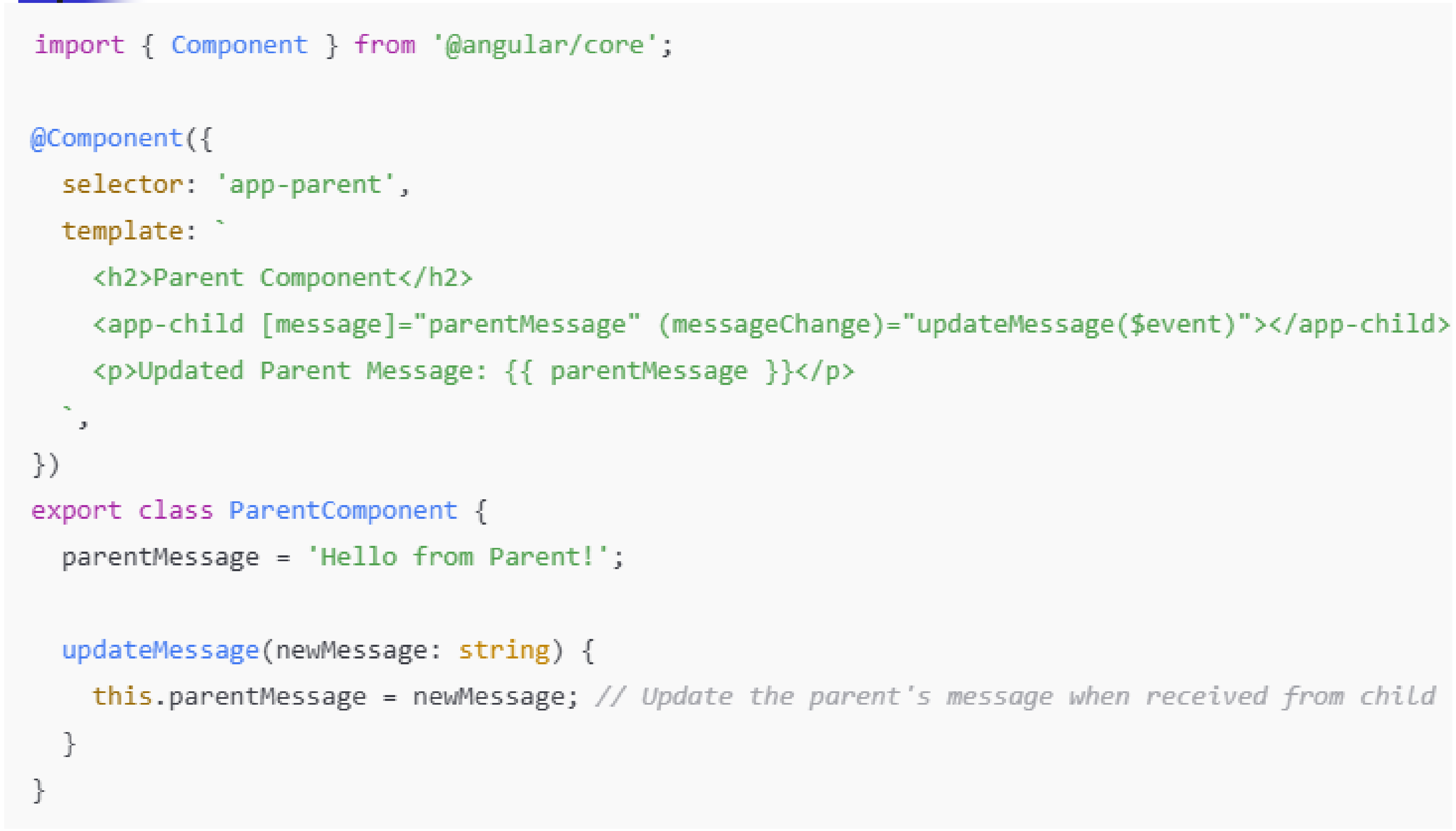


typescript

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

@Component({
  selector: 'app-child',
  template: `
    <p>The message is: {{ message }}</p>
    <button (click)="sendMessage()">Send Message to Parent</button>
  `,
})
export class ChildComponent {
  @Input() message: string = ''; // Custom property from parent

  @Output() messageChange = new EventEmitter<string>(); // Event to send data to parent

  sendMessage() {
    this.messageChange.emit('Hello from Child!'); // Emit a new message
  }
}
```

@Input() receives message from the parent.
@Output() and EventEmitter<string>() allow sending new data back.
sendMessage() triggers **messageChange.emit('Hello from Child!')** when the button is clicked.

Two-Way Binding example

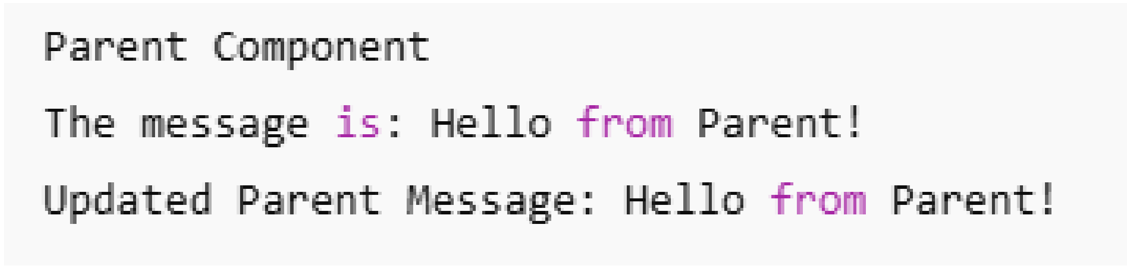


```
import { Component } from '@angular/core';

@Component({
  selector: 'app-parent',
  template: `
    <h2>Parent Component</h2>
    <app-child [message]="parentMessage" (messageChange)="updateMessage($event)"></app-child>
    <p>Updated Parent Message: {{ parentMessage }}</p>
  `,
})
export class ParentComponent {
  parentMessage = 'Hello from Parent!';

  updateMessage(newMessage: string) {
    this.parentMessage = newMessage; // Update the parent's message when received from child
  }
}
```

[message]="parentMessage" binds the parent’s message to the child.
(messageChange)="updateMessage($event)" listens for emitted messages and updates the parent.
updateMessage(newMessage: string) changes parentMessage dynamically.

Two-Way Binding example

The parent initially displays:

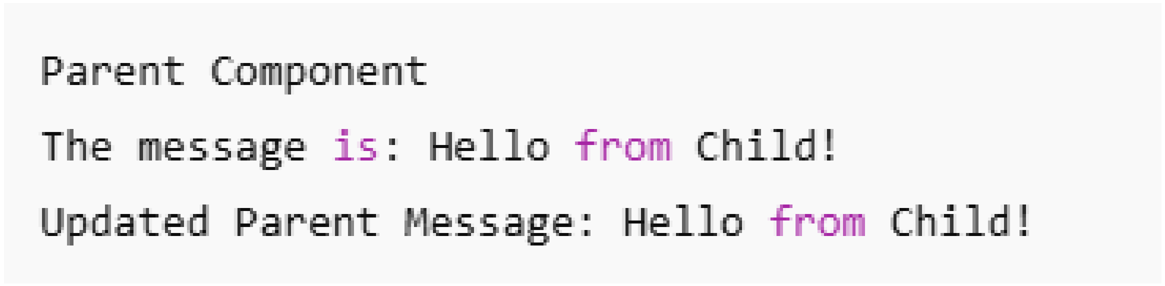


```
Parent Component

The message is: Hello from Parent!

Updated Parent Message: Hello from Parent!
```

When the user clicks **"Send Message to Parent"** in the child component:

- The child sends 'Hello from Child!' to the parent.
- The parent updates its parentMessage variable.
- The displayed text updates dynamically.

After Clicking the Button:



```
Parent Component

The message is: Hello from Child!

Updated Parent Message: Hello from Child!
```

Two-Way Binding example

Property Binding ([message]="parentMessage") passes data **from Parent → Child**.
Event Binding ((messageChange)="updateMessage($event)") allows **Child → Parent** communication.
Combining them simulates two-way binding without ngModel.

This is how Angular achieves two-way communication without ngModel by combining property and event binding!

Decorators in Angular

Decorators are special functions that modify the **behaviour** of **classes** or **class members** (**properties**, **methods**, etc.). **Decorators** are used to add **metadata** to **classes** so that the Angular compiler can process them appropriately.

Decorators can be broadly categorized into four main types:

1. Class decorators
2. Property decorators
3. Method decorators

1. Class Decorators

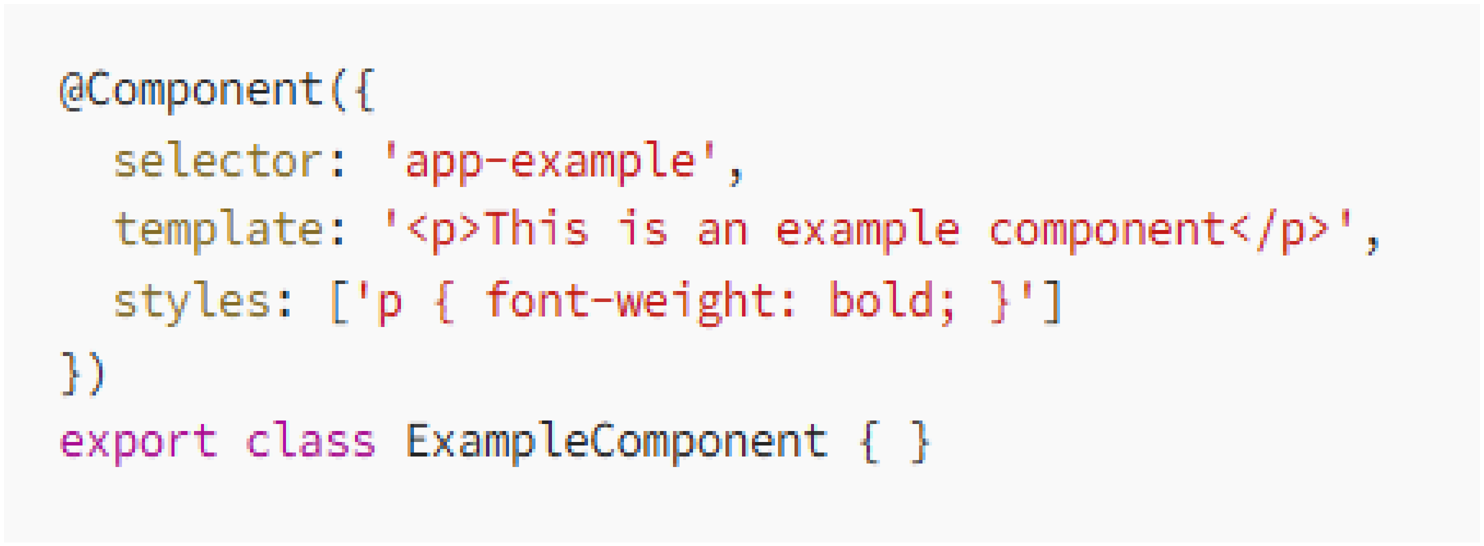
In Angular, class decorators are used to annotate and modify classes. Class decorators are functions that can be applied to a class declaration, typically prefixed with the @ symbol.

@Component,@Injectable,@Directive and **@NgModule** are widely used class decorators.

@Component

•Decorates a class to become an Angular component.

•It specifies **metadata** for the **component**, such as its **selector**, **template**, **styles**

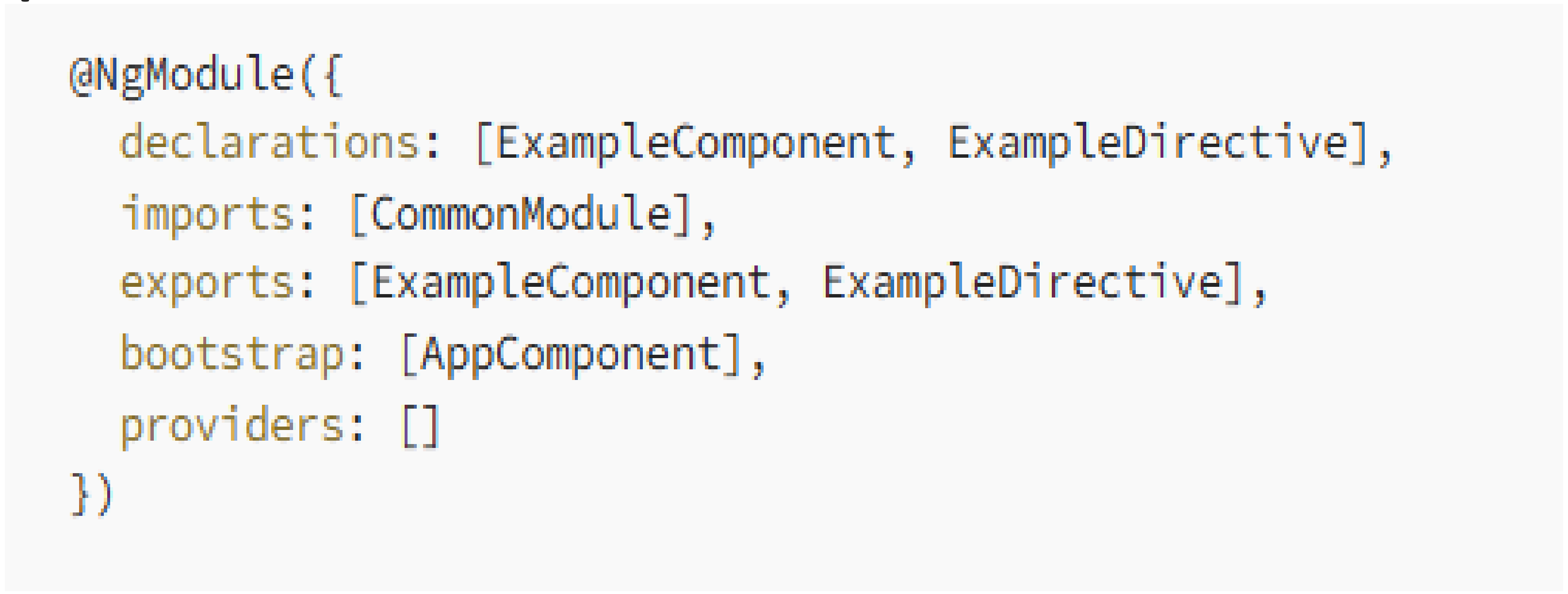


```
@Component({
  selector: 'app-example',
  template: '<p>This is an example component</p>',
  styles: ['p { font-weight: bold; }']
})
export class ExampleComponent { }
```

@NgModule

•Decorates a class to become an Angular module.

•It provides metadata for configuring the module, including **declarations**, **imports**, **exports**, **providers**, and **bootstrap components**.



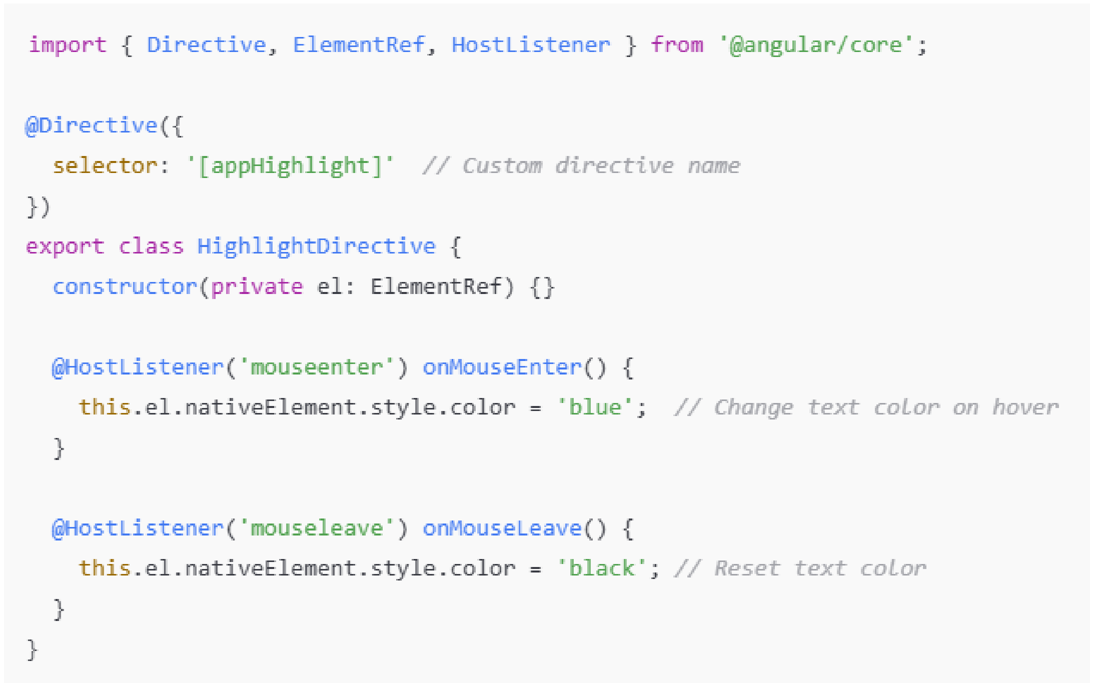
```
@NgModule({
  declarations: [ExampleComponent, ExampleDirective],
  imports: [CommonModule],
  exports: [ExampleComponent, ExampleDirective],
  bootstrap: [AppComponent],
  providers: []
})
```

@Directive

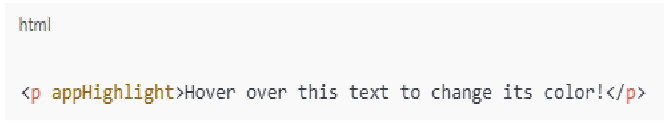
•Used to create **custom directives** in Angular.

•**Directives** are markers on a DOM element that tell Angular to attach a behaviour to that element (*ngIf, *ngFor).

•**Directives** are used to create **reusable components** or add **behaviour to elements** in the DOM.



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

@Directive({
  selector: '[appHighlight]'  // Custom directive name
})
export class HighlightDirective {
  constructor(private el: ElementRef) {}

  @HostListener('mouseenter') onMouseEnter() {
    this.el.nativeElement.style.color = 'blue';  // Change text color on hover
  }

  @HostListener('mouseleave') onMouseLeave() {
    this.el.nativeElement.style.color = 'black'; // Reset text color
  }
}
```



html

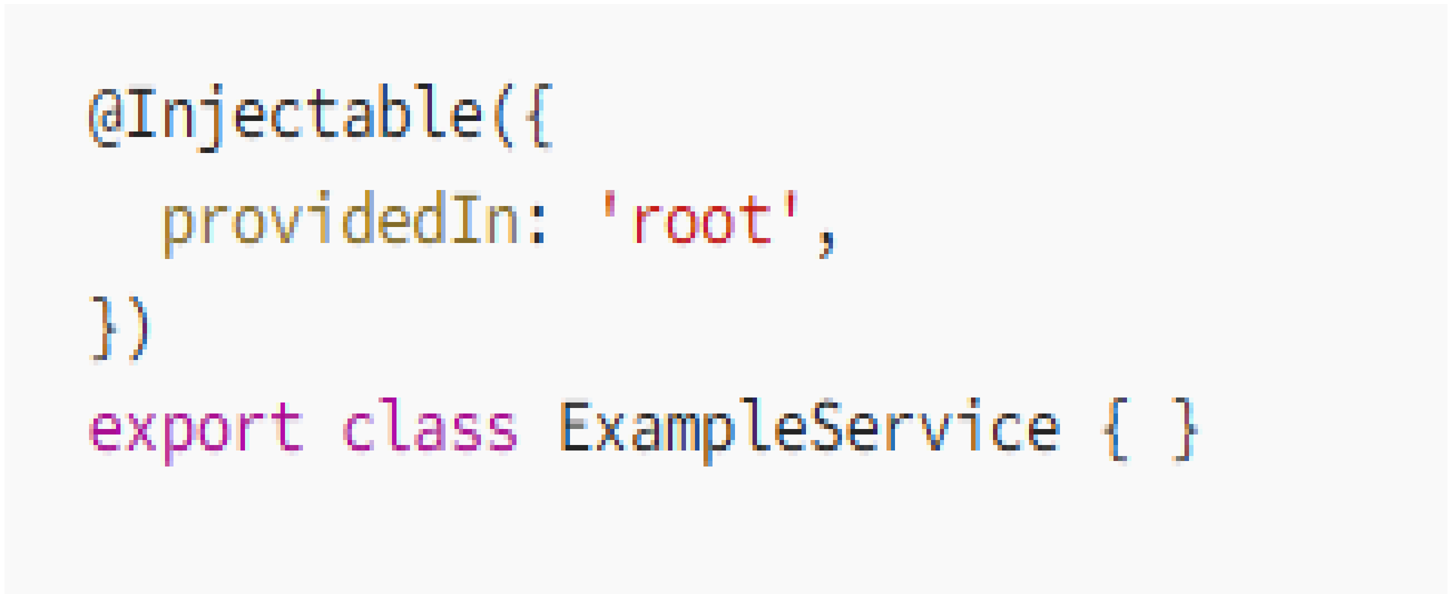
```html
<p appHighlight>Hover over this text to change its color!</p>
```

@Directive, enhance elements by adding custom behaviors, making the UI more interactive and dynamic.

@Injectable

•Used to declare a class as injectable, making it eligible for dependency injection.

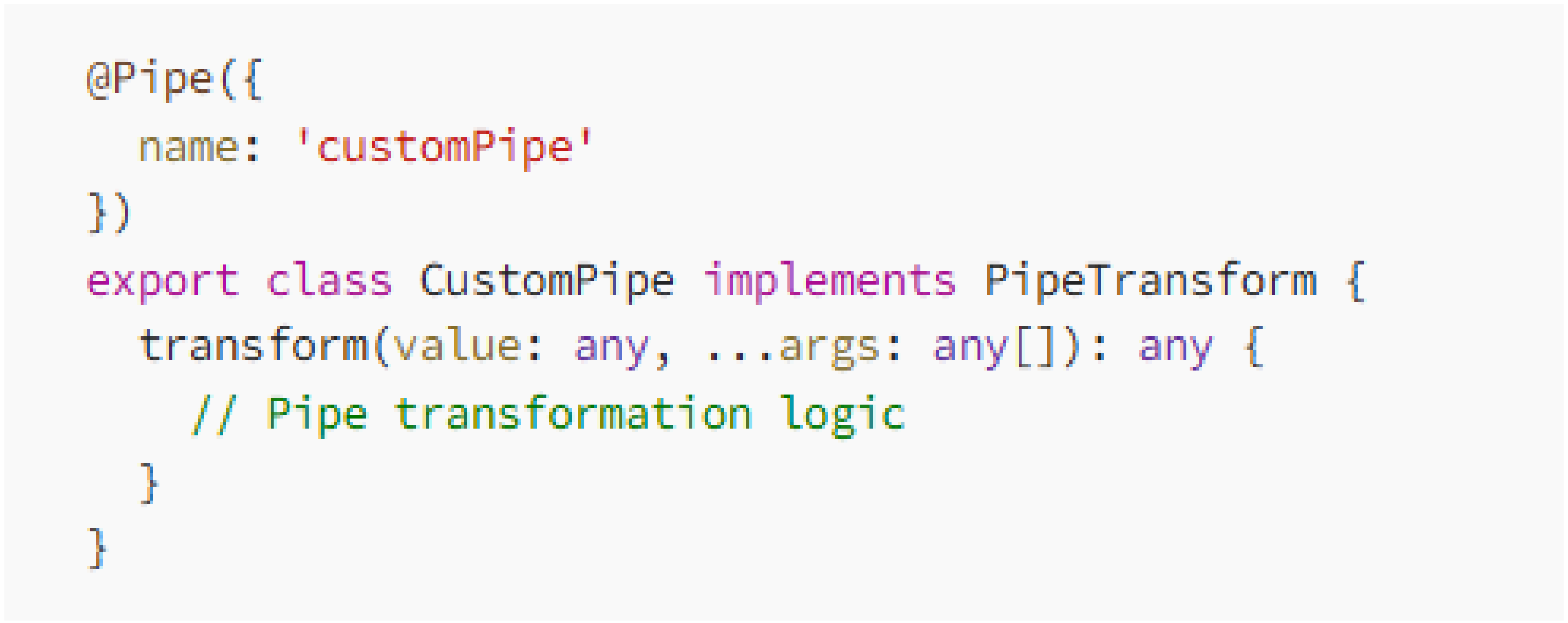
•Typically used with services.



```
@Injectable({
  providedIn: 'root',
})
export class ExampleService { }
```

@Pipe

- Used to create custom pipes for transforming data in templates.



```
@Pipe({
  name: 'customPipe'
})
export class CustomPipe implements PipeTransform {
  transform(value: any, ...args: any[]): any {
    // Pipe transformation logic
  }
}
```

2. Property Decorators

- **Property decorators** in angular are used to **annotate** and **modify class properties**. They are applied to the property declarations directly.

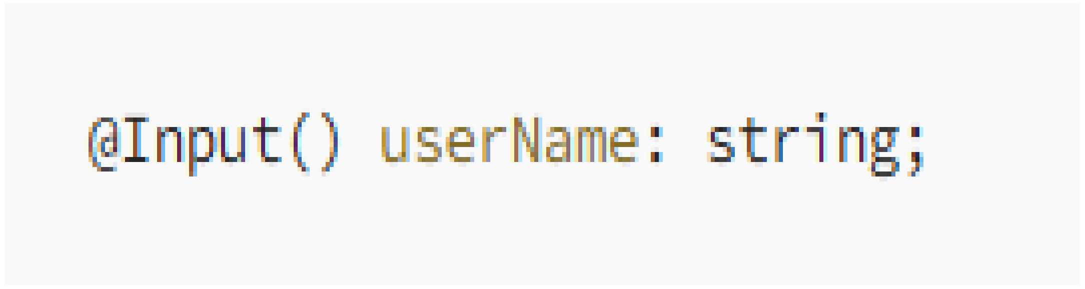
- Property decorators are **invoked at runtime** with the target object (instance of the class) and property name.

- **@Input(),@Output()** are widely used property decorators

@Input

•Used to declare an input property in a component.

•Allows the parent component to pass data to the child component.

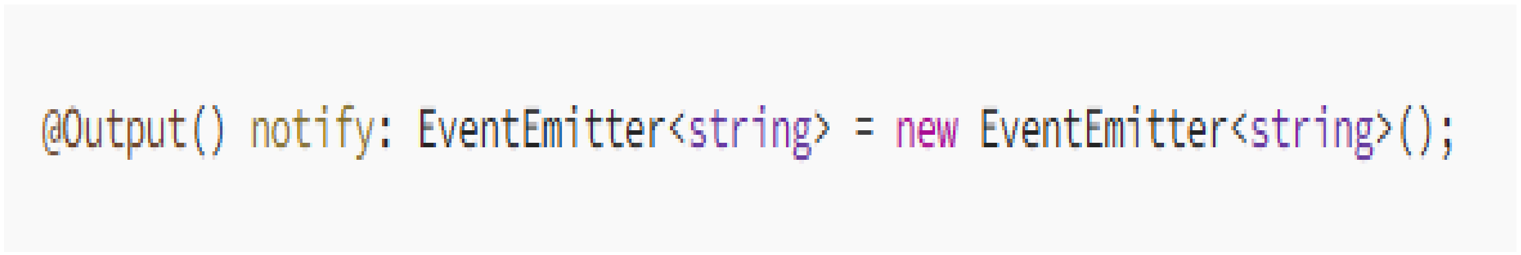


```
@Input() userName: string;
```

@Output

•Used to declare an output property in a component.

•Allows a child component to emit events that the parent component can listen to.



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

3. Method Decorators

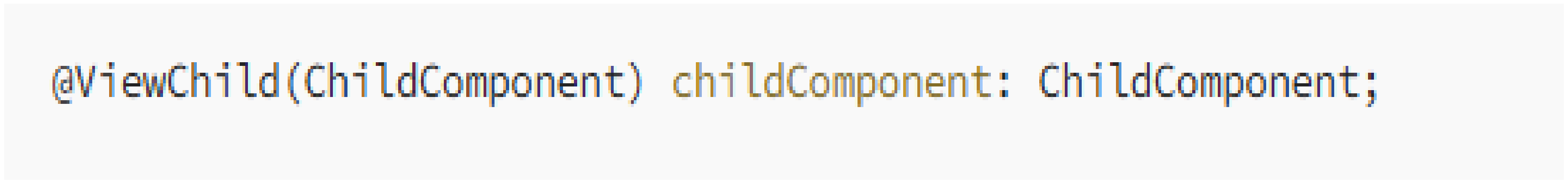
Method decorators in angular are used to **annotate** and **modify class methods**. They are applied to the method declarations directly. **Method decorators are invoked at runtime** with the **target object (instance of the class)**, **the property name (method name)**, and a **property descriptor**.

Angular provides **method decorators** that allow interaction with **components**, **DOM elements**, and **events**.

@ViewChild,@ContentChild,@HostListener are widely used method decorators.

@ViewChild and @ViewChildren

•Used to access child components or elements from the parent component.



```
@ViewChild(ChildComponent) childComponent: ChildComponent;
```

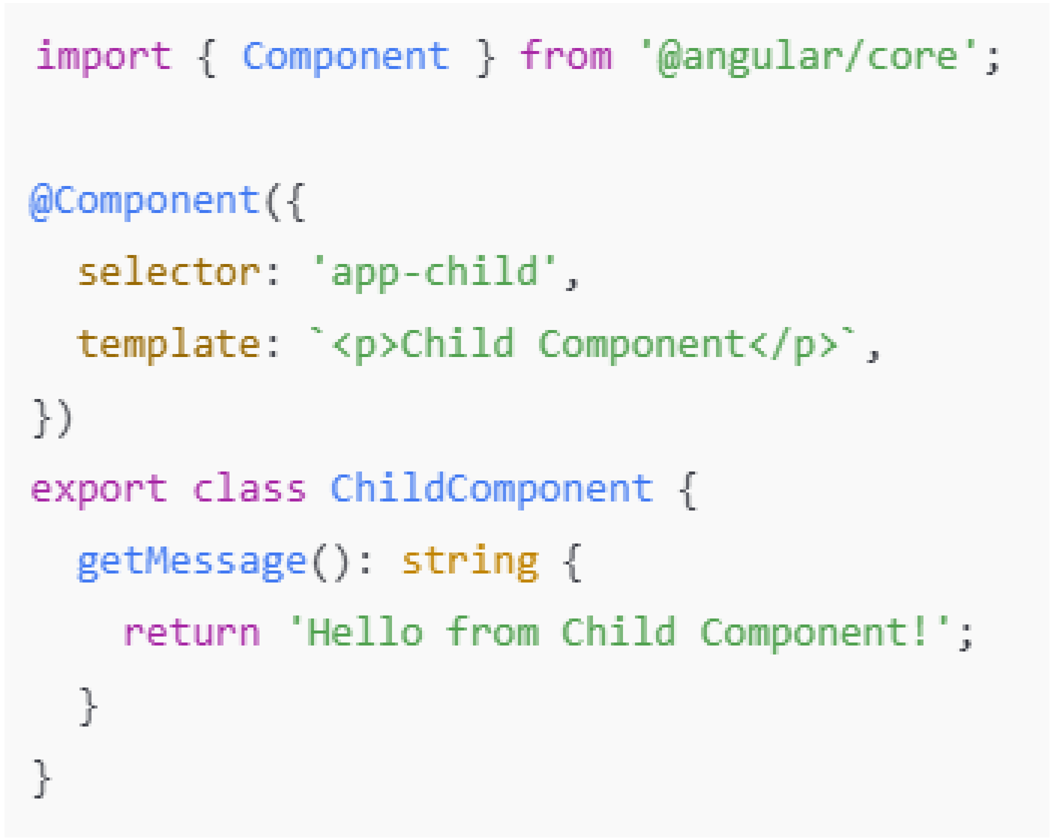
@ViewChild – Access a Single Child Element or Component

@ViewChild retrieves a **single instance** of a component, directive, or element from the template.

It is used when you need to **manipulate or interact** with a child component or an HTML element in the DOM.

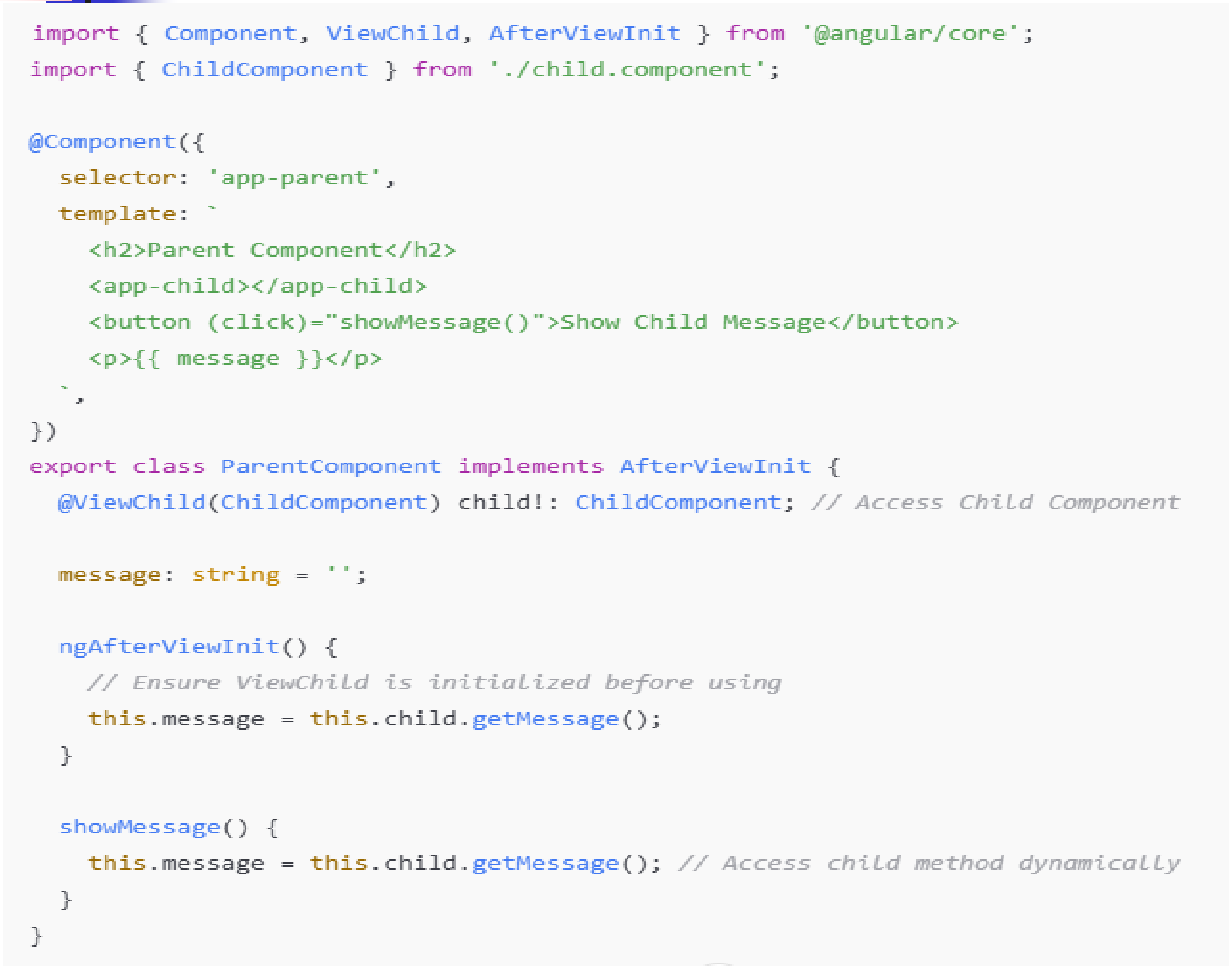
Example: Access a Child Component

Child Component (child.component.ts)



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

@Component({
  selector: 'app-child',
  template: `<p>Child Component</p>`,
})
export class ChildComponent {
  getMessage(): string {
    return 'Hello from Child Component!';
  }
}
```

Example: Access a Child Component



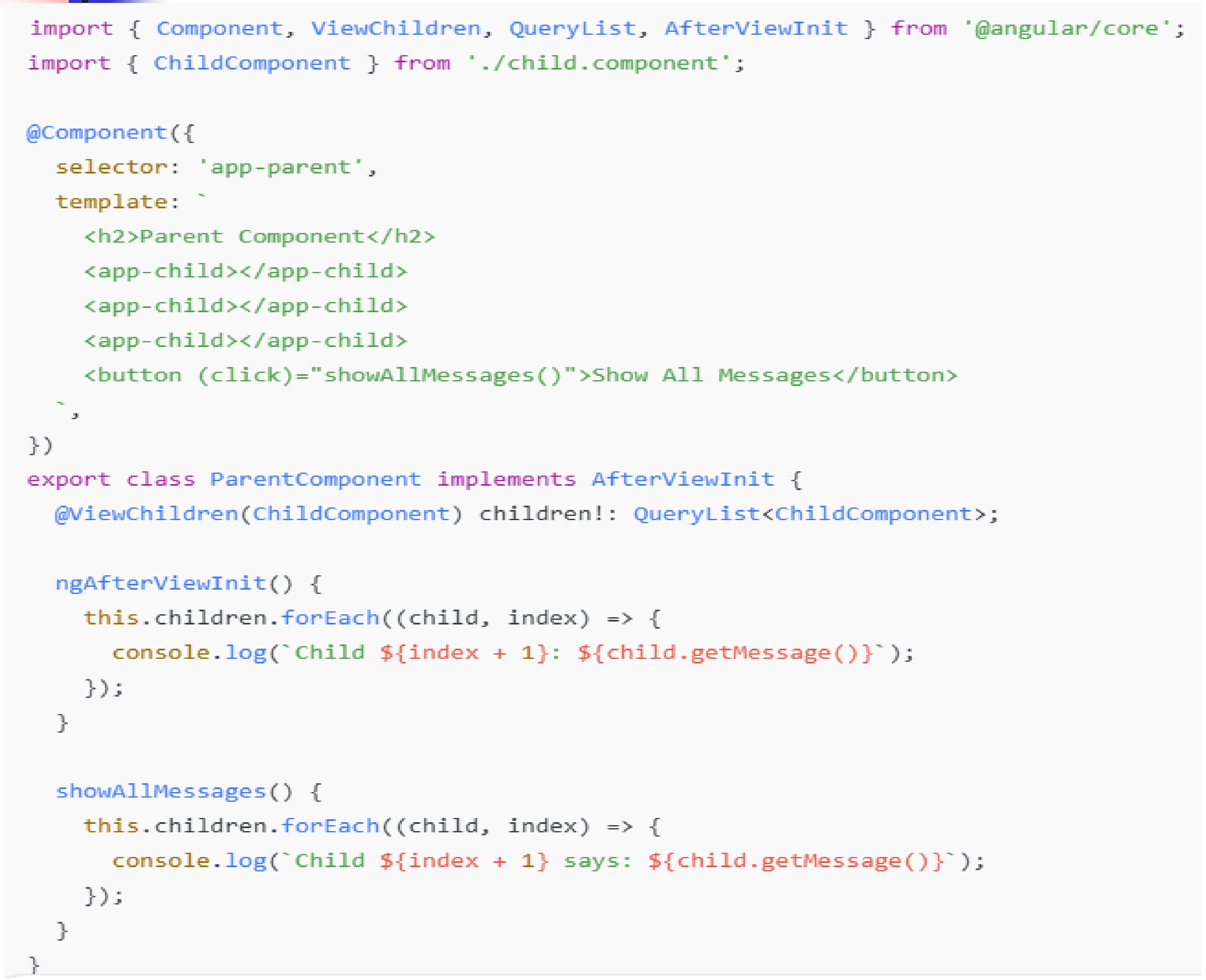
```
import { Component, ViewChild, AfterViewInit } from '@angular/core';
import { ChildComponent } from './child.component';

@Component({
  selector: 'app-parent',
  template: `
    <h2>Parent Component</h2>
    <app-child></app-child>
    <button (click)="showMessage()">Show Child Message</button>
    <p>{{ message }}</p>
  `,
})
export class ParentComponent implements AfterViewInit {
  @ViewChild(ChildComponent) child!: ChildComponent; // Access Child Component

  message: string = '';

  ngAfterViewInit() {
    // Ensure ViewChild is initialized before using
    this.message = this.child.getMessage();
  }

  showMessage() {
    this.message = this.child.getMessage(); // Access child method dynamically
  }
}
```

Parent Component (parent.component.ts)

@ViewChild(ChildComponent) gets a reference to the child component.

ngAfterViewInit() ensures that the child component is available before using it.

Example: @ViewChildren – Access Multiple Child Component



```
import { Component, ViewChildren, QueryList, AfterViewInit } from '@angular/core';
import { ChildComponent } from './child.component';

@Component({
  selector: 'app-parent',
  template: `
    <h2>Parent Component</h2>
    <app-child></app-child>
    <app-child></app-child>
    <app-child></app-child>
    <button (click)="showAllMessages()">Show All Messages</button>
  `,
})
export class ParentComponent implements AfterViewInit {
  @ViewChildren(ChildComponent) children!: QueryList<ChildComponent>;

  ngAfterViewInit() {
    this.children.forEach((child, index) => {
      console.log(`Child ${index + 1}: ${child.getMessage()}`);
    });
  }

  showAllMessages() {
    this.children.forEach((child, index) => {
      console.log(`Child ${index + 1} says: ${child.getMessage()}`);
    });
  }
}
```

@ViewChildren is used to query multiple elements or components inside the view.

It returns a QueryList<T>, which is an iterable list of elements.

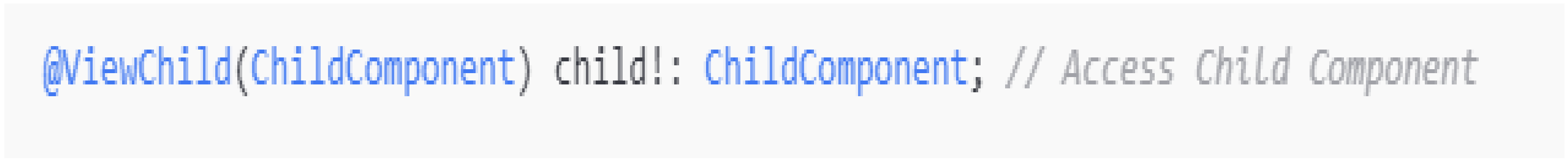
The data becomes available **after view initialization (ngAfterViewInit())**.

(!) non-null assertion operator

The **exclamation mark (!)** is called the **non-null assertion operator**. It tells TypeScript that a variable **will definitely be assigned a value** before it is used, even if TypeScript cannot verify it at compile time.

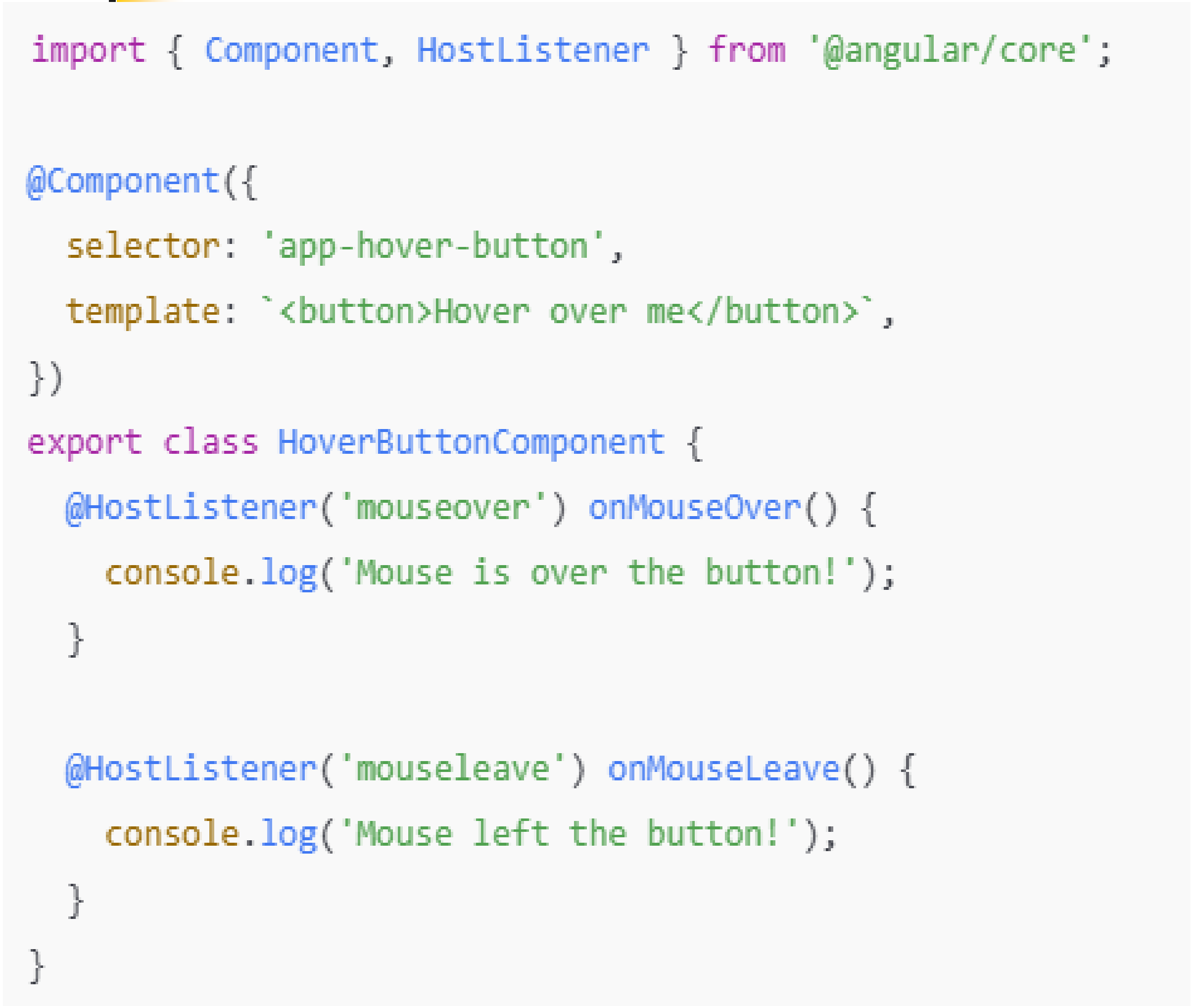
The **! operator** tells TypeScript, *“Trust me, this will be initialized before use.”*

Without !, TypeScript may throw an error because @ViewChild is not explicitly initialized:



```
@ViewChild(ChildComponent) child!: ChildComponent; // Access Child Component
```

Example: @HostListener – Listen to DOM Events



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


@Component({
  selector: 'app-hover-button',
  template: `<button>Hover over me</button>`,
})
export class HoverButtonComponent {
  @HostListener('mouseover') onMouseOver() {
    console.log('Mouse is over the button!');
  }

  @HostListener('mouseleave') onMouseLeave() {
    console.log('Mouse left the button!');
  }
}
```

@HostListener('eventName') listens for **events on the host element**.

The event name can be any **DOM event** (e.g., click, mouseover, resize).

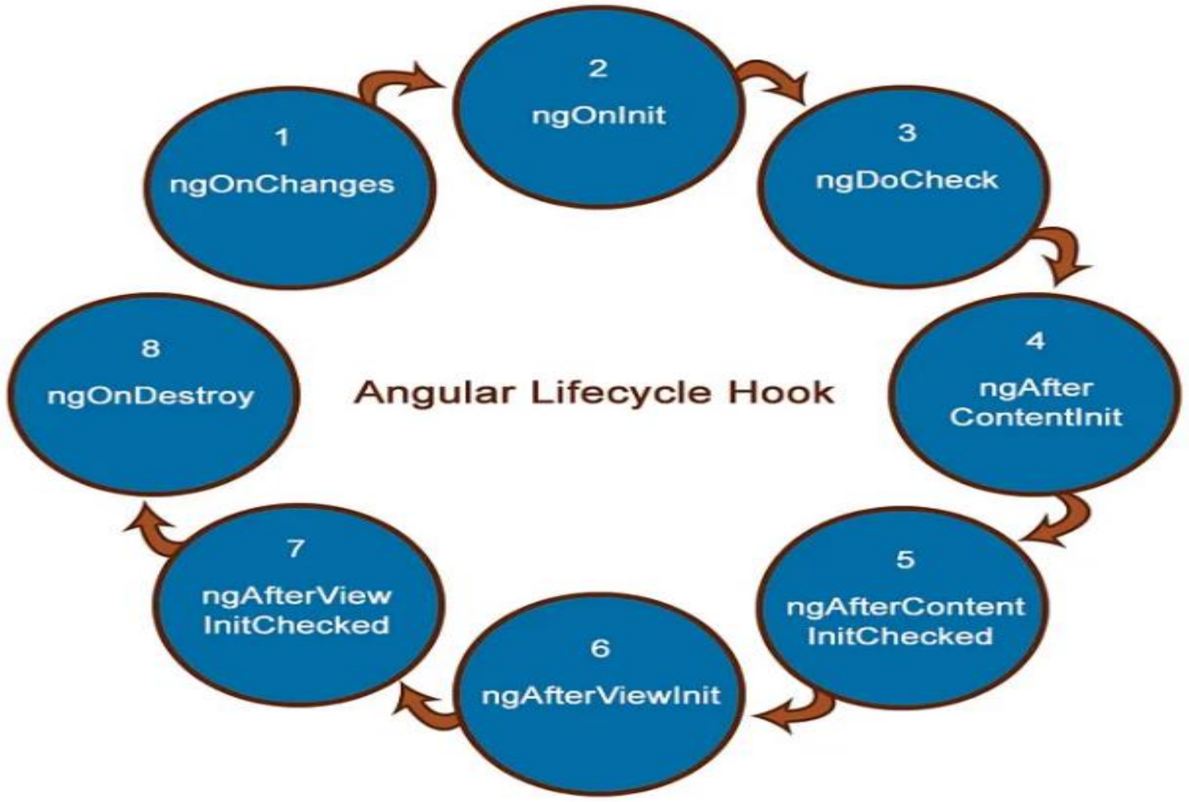
@HostListener allows a directive or component to listen to DOM events on its host element.

It helps in handling user interactions without directly using event binding ((event)="handler()").

Detecting Mouse Hover on a Button

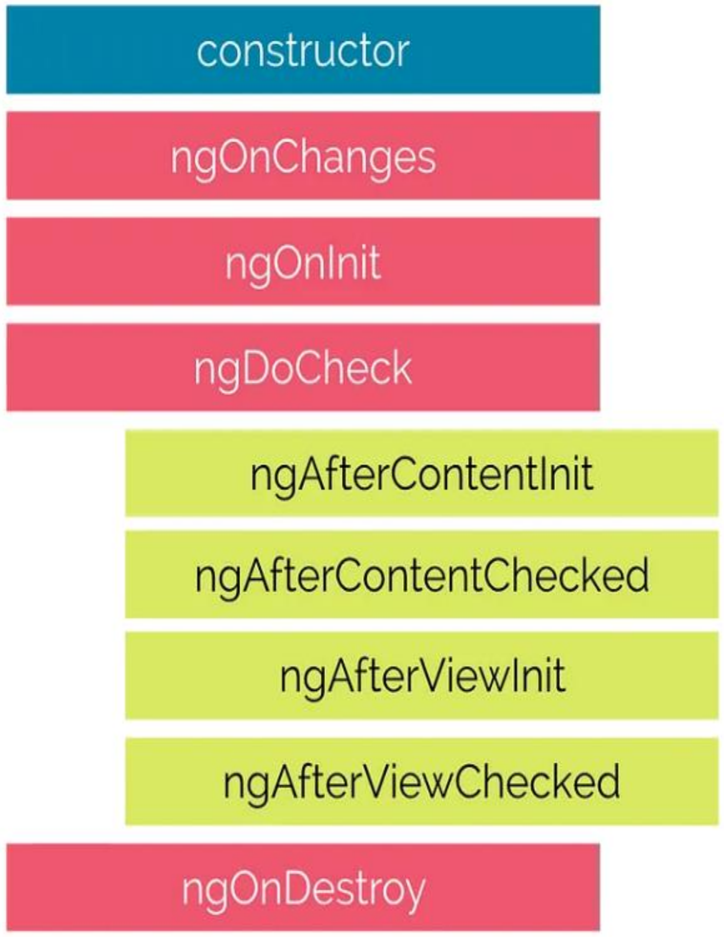
Angular Component Lifecycle Hook

A **component's lifecycle** is the **sequence of steps** that happen between the component's **creation** and its **destruction**. Each step represents a different part of Angular's process for **rendering components** and **checking** them for updates over time.



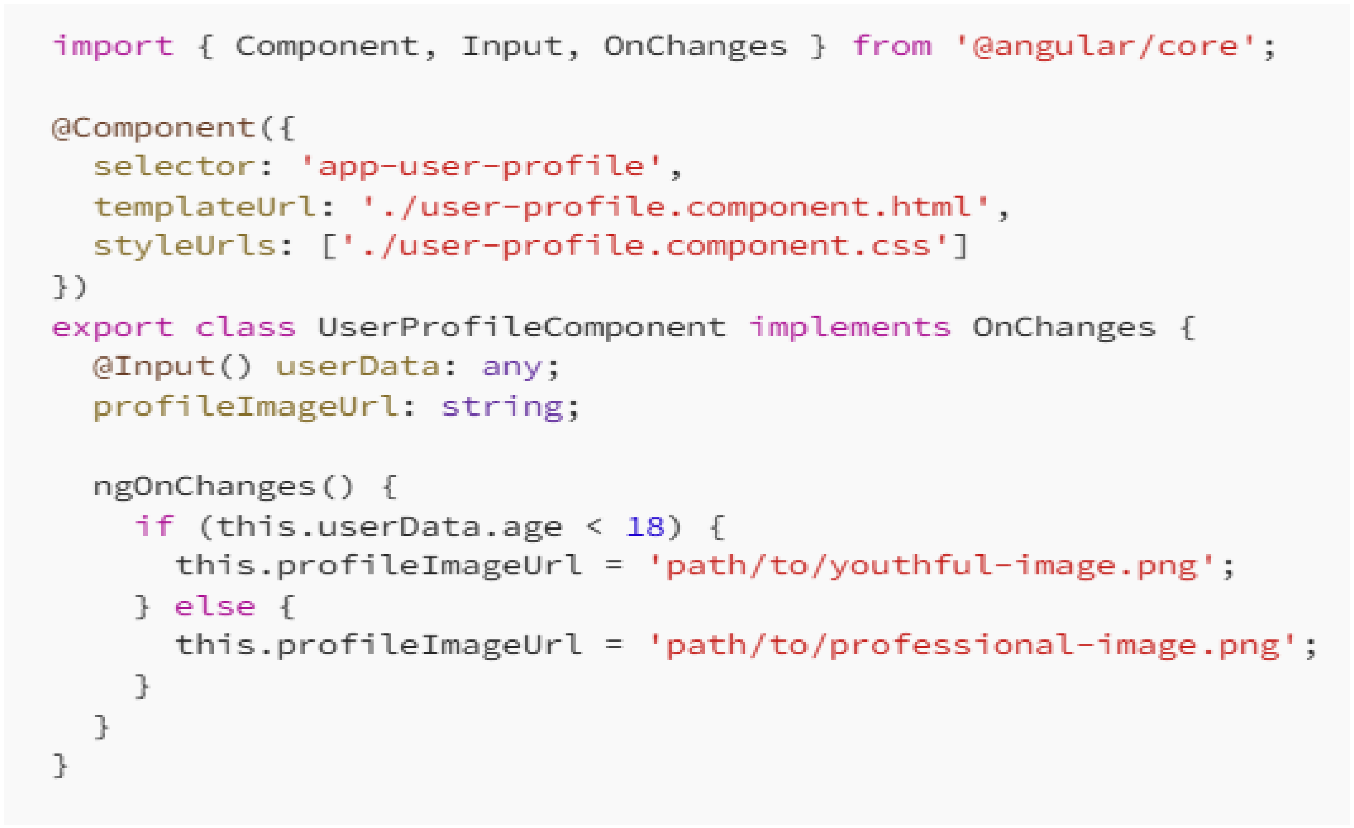
Angular Component Lifecycle Hook

- In Angular, every component has a life-cycle, a number of different stages it goes through from initializing to destroying.
- There are **8 different stages** in the component lifecycle.
- Every stage is called **life cycle hook events**
- Since a component is a typescript class, for that reason every component must have a **constructor** method. The constructor of the component class executes **first before the execution of any other life cycle hook event**.



1. ngOnChanges - Runs When Input Properties Change

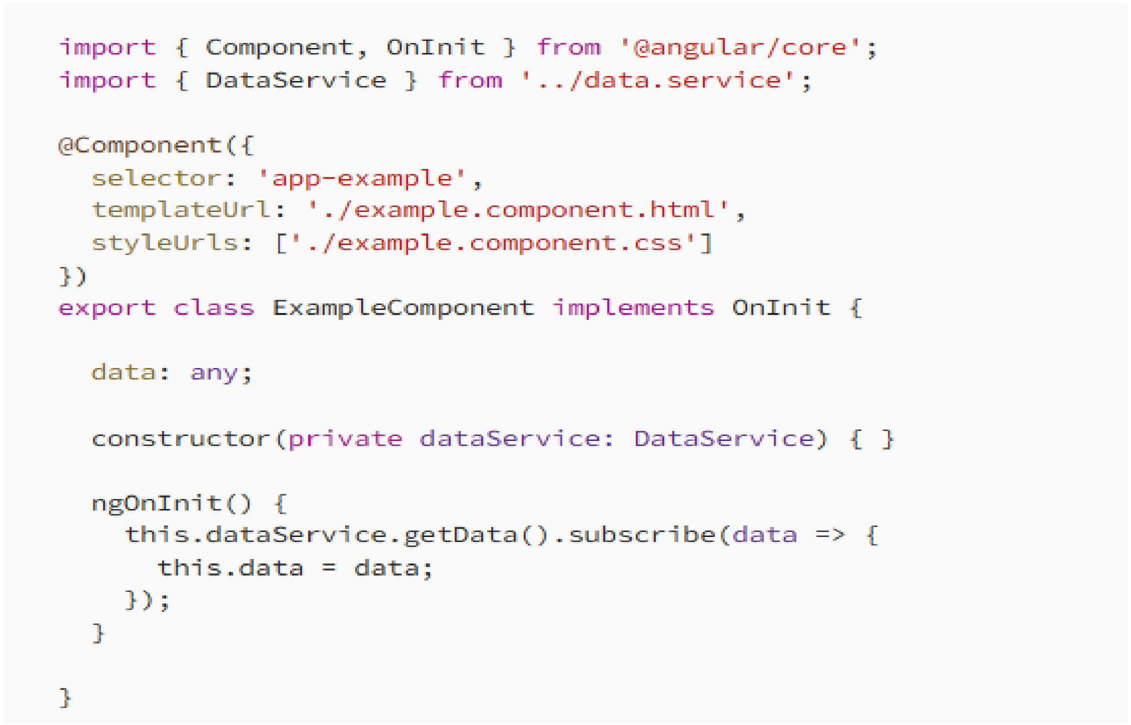
This method is a lifecycle hook in Angular that is **invoked every time** there is a **change in one of the input properties of the component**. Called before ngOnInit() and whenever one or more of the component's input properties changes.



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

@Component({
  selector: 'app-user-profile',
  templateUrl: './user-profile.component.html',
  styleUrls: ['./user-profile.component.css']
})
export class UserProfileComponent implements OnChanges {
  @Input() userData: any;
  profileImageUrl: string;

  ngOnChanges() {
    if (this.userData.age < 18) {
      this.profileImageUrl = 'path/to/youthful-image.png';
    } else {
      this.profileImageUrl = 'path/to/professional-image.png';
    }
  }
}
```

2. ngOnInit - Runs Once After Component Initialization

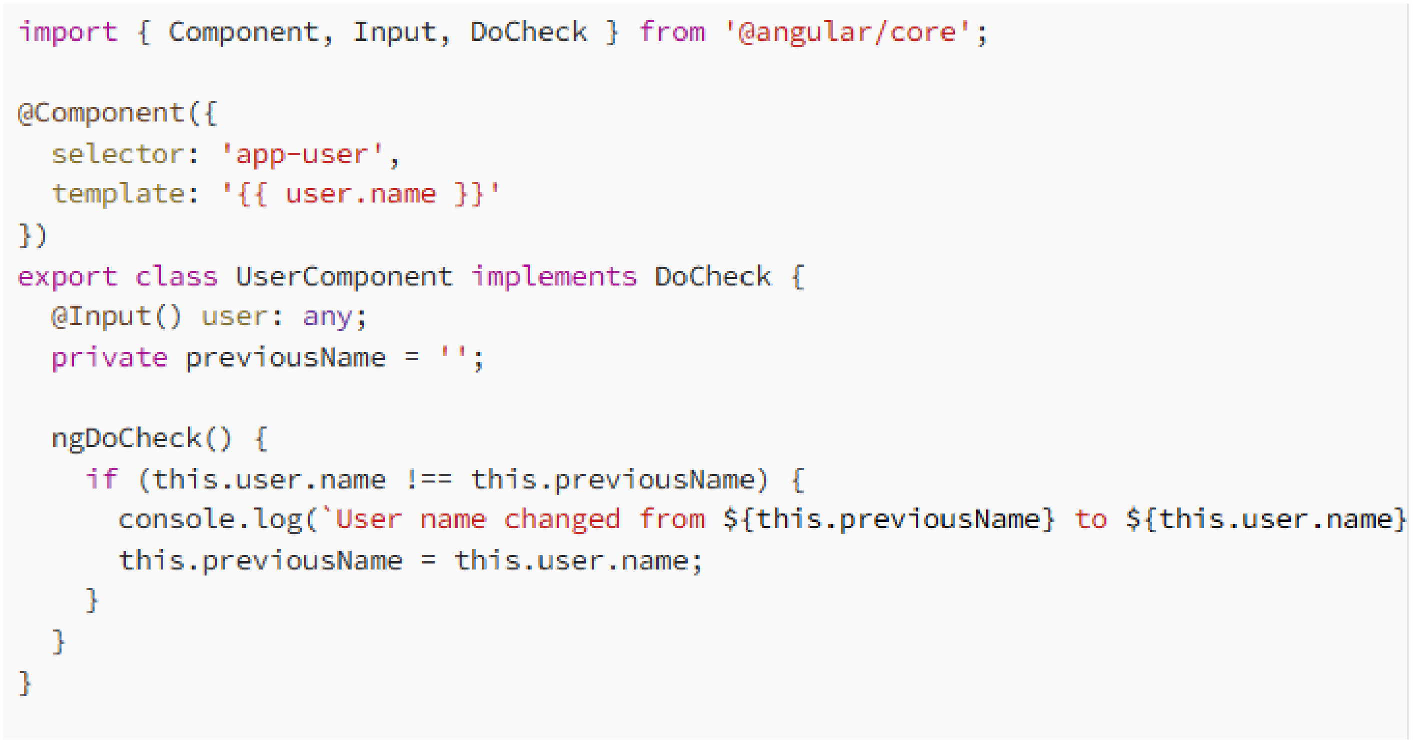
This method is a lifecycle hook that is called once **after the ngOnChanges method**, and it is invoked **when the component has been initialized**. Since **ngOnInit** runs **after** the **constructor** method, **all injected dependencies will be resolved** and **all class members** will be defined. In the example below, **ngOnInit()** hook uses a **dataService** to get the data. This method runs only once during **the initial loading** of the **component** and is used to **initialize the state of the component.**



```
import { Component, OnInit } from '@angular/core';
import { DataService } from '../data.service';

@Component({
  selector: 'app-example',
  templateUrl: './example.component.html',
  styleUrls: ['./example.component.css']
})
export class ExampleComponent implements OnInit {

  data: any;

  constructor(private dataService: DataService) { }

  ngOnInit() {
    this.dataService.getData().subscribe(data => {
      this.data = data;
    });
  }

}
```

3. ngDoCheck - Runs on Every Change Detection Cycle

This method is another lifecycle hook in Angular and it is invoked when the **change detector of the given component is invoked**. It allows us to implement **our own change detection algorithm** for the given component.
The **ngDoCheck** lifecycle hook is called every time Angular runs **change detection** for a component.



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

@Component({
  selector: 'app-user',
  template: '{{ user.name }}'
})
export class UserComponent implements DoCheck {
  @Input() user: any;
  private previousName = '';

  ngDoCheck() {
    if (this.user.name !== this.previousName) {
      console.log(`User name changed from ${this.previousName} to ${this.user.name}
      this.previousName = this.user.name;
    }
  }
}
```

4. ngAfterContentInit - Runs After Content Projection (<ng-content>)

ngAfterContentInit() is a lifecycle hook in Angular that **runs once after the component's projected content (via <ng-content>) has been initialized**.

It **runs only once after Angular projects content from a parent component into a child component.**

It is used when we need to interact with projected content after it has been initialized.

It is different from ngOnInit(), which runs when a component itself initializes.

4. What is <ng-content> (Content Projection)?

Content Projection allows you to **pass content from a parent component to a child component** without using @Input().

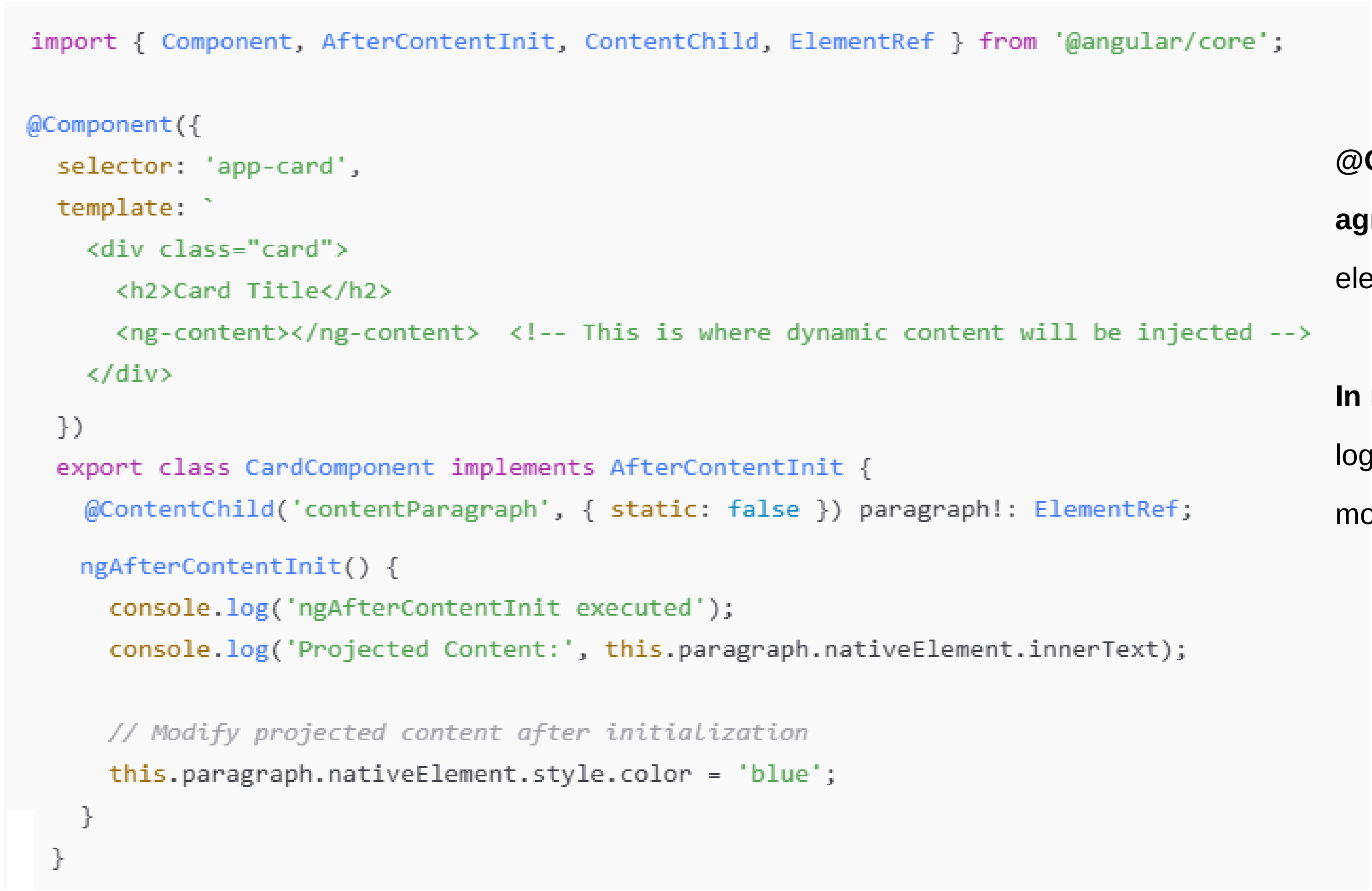
<ng-content> acts as a placeholder for **dynamic content in the child component**.

This is useful when building **reusable and flexible components**.

4. ngAfterContentInit - Runs After Content Projection (<ng-content>)

Child Component (app-card)

The **<ng-content>** inside **the app-card** component **will receive content dynamically.**



```
import { Component, AfterContentInit, ContentChild, ElementRef } from '@angular/core';

@Component({
  selector: 'app-card',
  template: `
    <div class="card">
      <h2>Card Title</h2>
      <ng-content></ng-content>  <!-- This is where dynamic content will be injected -->
    </div>
  })
  export class CardComponent implements AfterContentInit {
    @ContentChild('contentParagraph', { static: false }) paragraph!: ElementRef;

   ngAfterContentInit() {
     console.log('ngAfterContentInit executed');
     console.log('Projected Content:', this.paragraph.nativeElement.innerText);

     // Modify projected content after initialization
     this.paragraph.nativeElement.style.color = 'blue';
   }
 }
```

@ContentChild('contentParagraph') finds the <p> element inside <ng-content>.

In ngAfterContentInit(), we log the projected content and modify its color

4. ngAfterContentInit - Runs After Content Projection (<ng-content>)

Use **the Card Component** in a **Parent Component**

use **<app-card>** inside **Parent component**

app.component.html (Parent Component)

```
<app-card>
  <p #contentParagraph> This is dynamic projected content inside the card! </p>
</app-card>
```

The **<p> element** inside <app-card> is **dynamically inserted** into **<ng-content>.**

The **#contentParagraph** is passed into **@ContentChild()** in the **child component**.

When **ngAfterContentInit()** runs, it will **log the content** and **change its text color**.

The # symbol in Angular is used to create a **template reference variable**. It allows you to reference a **DOM element, directive, or component** inside an Angular template.

4. ngAfterContentInit - Runs After Content Projection (<ng-content>)

- **<ng-content> enables content projection** (passing custom content from parent to child).
-

  ngAfterContentInit() runs after projected content is initialized (useful for manipulating it).
-

  @ContentChild() lets us access projected elements inside the child component.
-

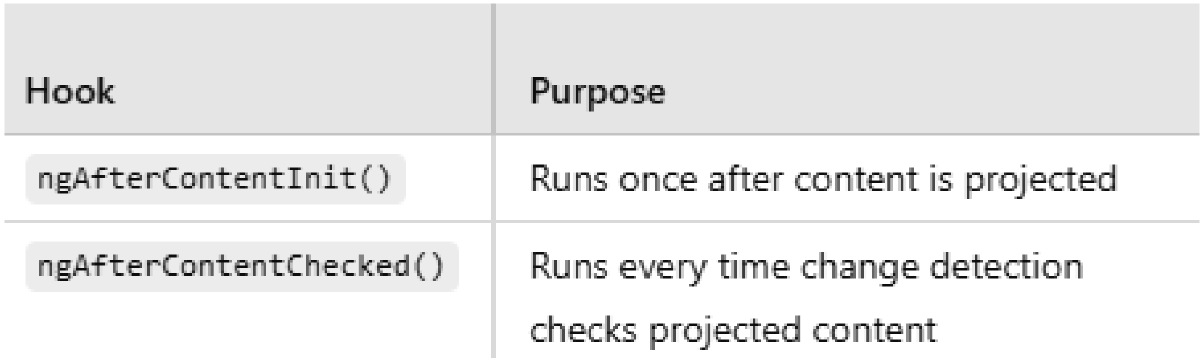
  Content Projection makes Angular components reusable and flexible.

5. ngAfterContentChecked - Runs After Content Projection is Checked

ngAfterContentChecked() is a lifecycle hook in Angular that **runs after every change detection cycle**, whenever **projected content inside <ng-content> is checked** for changes.

When does it execute?

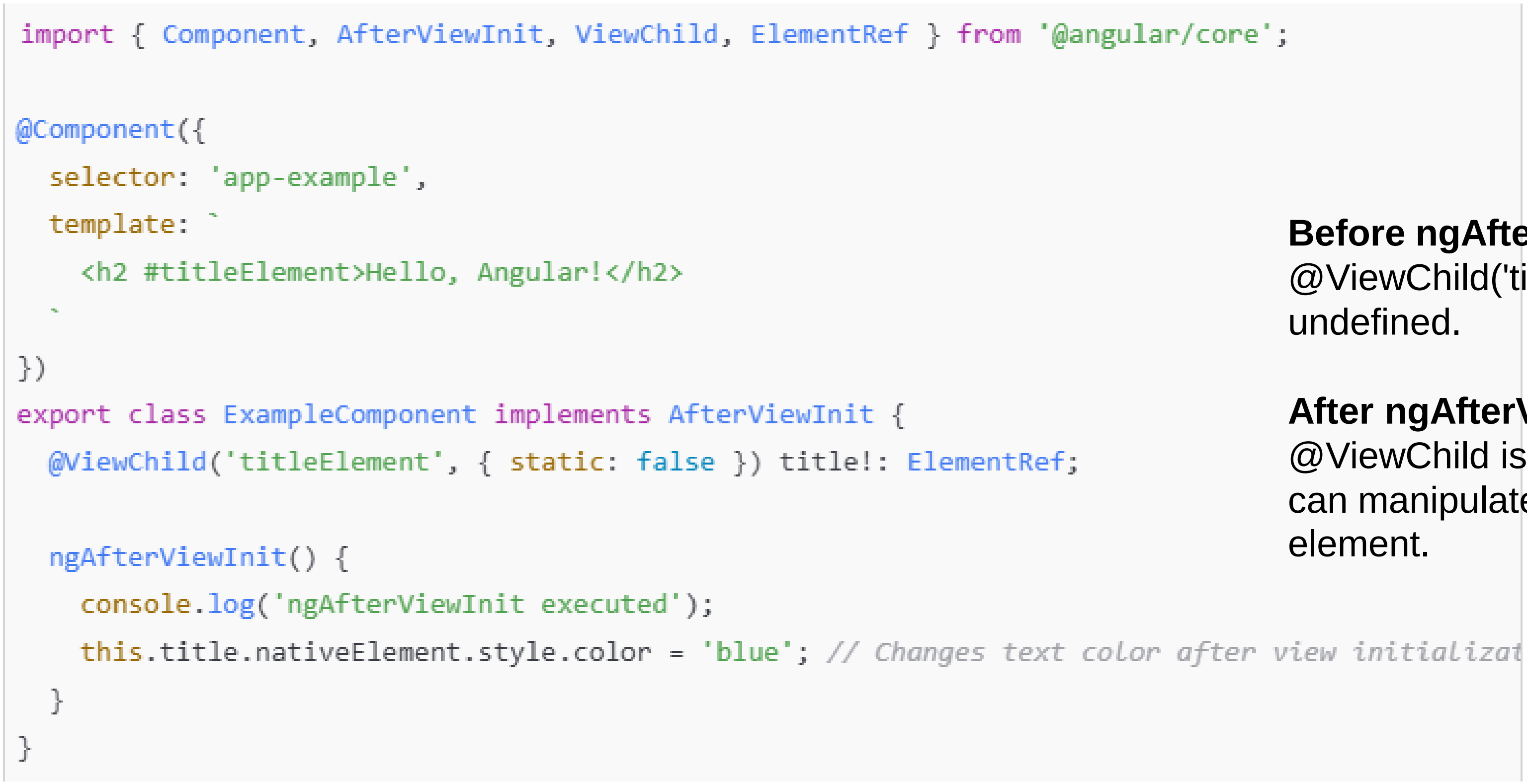
- It runs **after ngAfterContentInit()** and **on every change detection cycle**.
- It is triggered whenever **Angular detects changes** in **projected content inside <ng-content>.**
- It can be used to **respond to dynamic changes** in projected content.



Hook	Purpose
`ngAfterContentInit()`	Runs once after content is projected
`ngAfterContentChecked()`	Runs every time change detection checks projected content

6. ngAfterViewInit - Runs After View Initialization

Runs once **after Angular initializes the component's view** (including child components).
Useful when you **need to access and manipulate child components** or **DOM elements** after they are available.
It ensures that **@ViewChild()** and **@ViewChildren() properties** are populated.

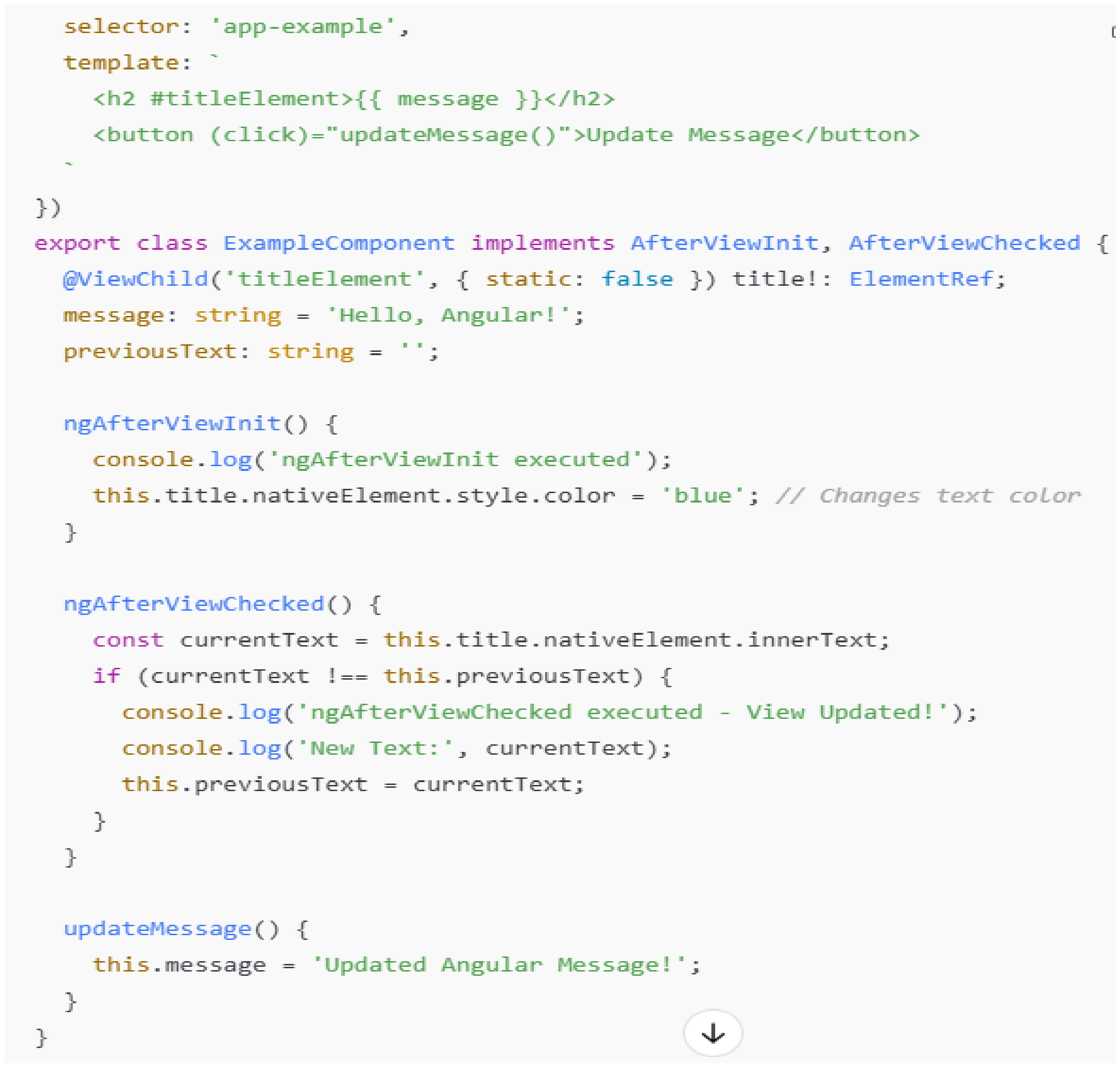


```
import { Component, AfterViewInit, ViewChild, ElementRef } from '@angular/core';

@Component({
  selector: 'app-example',
  template: `
    <h2 #titleElement>Hello, Angular!</h2>
  `
})
export class ExampleComponent implements AfterViewInit {
  @ViewChild('titleElement', { static: false }) title!: ElementRef;

  ngAfterViewInit() {
    console.log('ngAfterViewInit executed');
    this.title.nativeElement.style.color = 'blue'; // Changes text color after view initializat
  }
}
```

Before ngAfterViewInit() → @ViewChild('titleElement') is undefined.

After ngAfterViewInit() → @ViewChild is available, and we can manipulate the DOM element.

7. ngAfterViewChecked - Runs After View is Checked

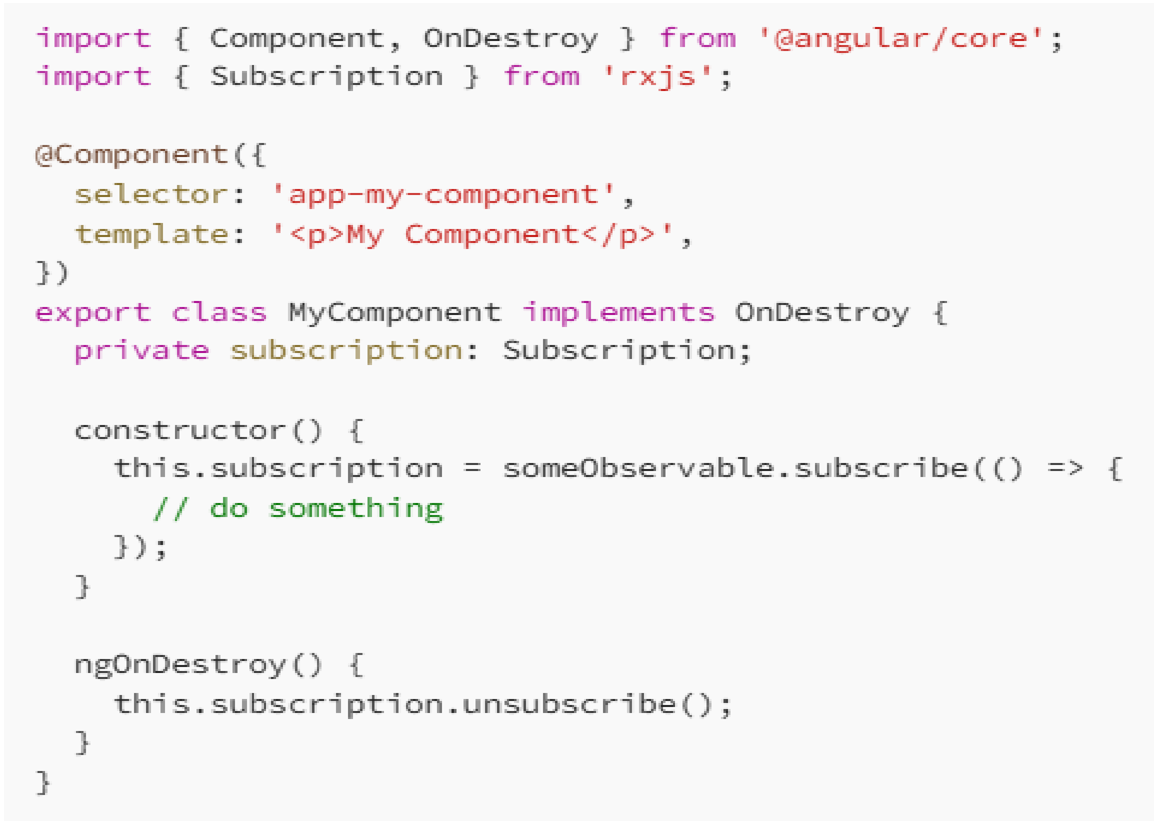


```
  selector: 'app-example',
  template: `
    <h2 #titleElement>{{ message }}</h2>
    <button (click)="updateMessage()">Update Message</button>
  `
})
export class ExampleComponent implements AfterViewInit, AfterViewChecked {
  @ViewChild('titleElement', { static: false }) title!: ElementRef;
  message: string = 'Hello, Angular!';
  previousText: string = '';

  ngAfterViewInit() {
    console.log('ngAfterViewInit executed');
    this.title.nativeElement.style.color = 'blue'; // Changes text color
  }

  ngAfterViewChecked() {
    const currentText = this.title.nativeElement.innerText;
    if (currentText !== this.previousText) {
      console.log('ngAfterViewChecked executed - View Updated!');
      console.log('New Text:', currentText);
      this.previousText = currentText;
    }
  }

  updateMessage() {
    this.message = 'Updated Angular Message!';
  }
}
```

- Runs **after every change detection cycle**, whenever the view is checked for updates.
- Useful for **detecting changes in the component's view** that occur **dynamically**.
- Can be **used to respond to changes** in the view **after Angular updates the DOM**.

8.ngOnDestroy - Runs Before Component Destruction

This is a lifecycle hook that is invoked just **before** Angular destroys the component. You can use this hook to **unsubscribe observables** and **detach event handlers** to **avoid memory leaks**.



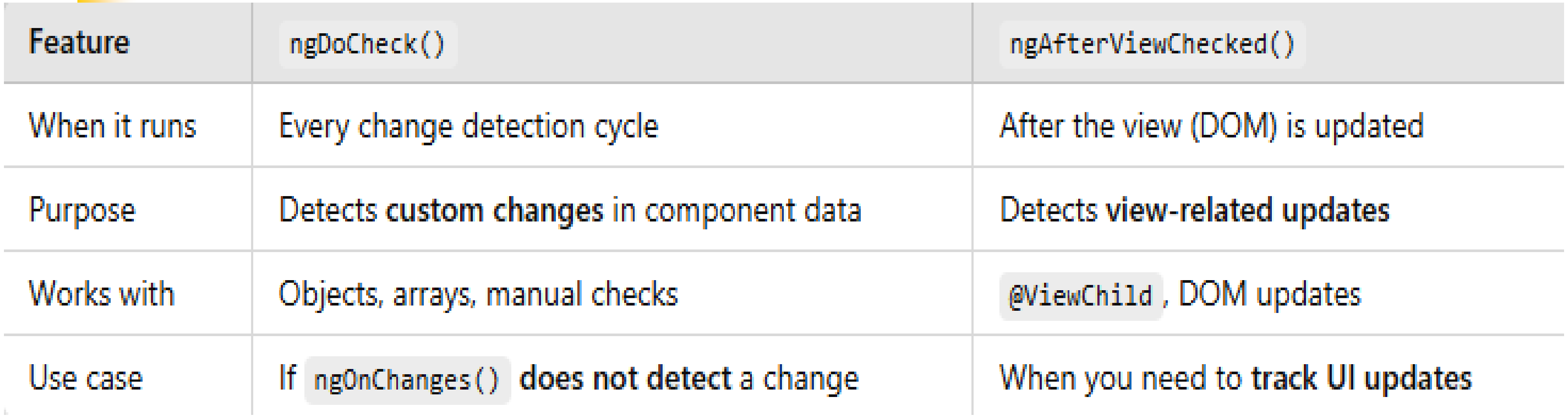
```
import { Component, OnDestroy } from '@angular/core';
import { Subscription } from 'rxjs';

@Component({
  selector: 'app-my-component',
  template: '<p>My Component</p>',
})
export class MyComponent implements OnDestroy {
  private subscription: Subscription;

  constructor() {
    this.subscription = someObservable.subscribe(() => {
      // do something
    });
  }

  ngOnDestroy() {
    this.subscription.unsubscribe();
  }
}
```

Component Lifecycle Hook

- Use **ngOnInit()** for **component initialization**.
- Use **ngOnChanges()** to **react to @Input() property changes**.
- Use **ngDoCheck()** for **custom change detection** (use carefully).
- Use **ngAfterContentInit() & ngAfterContentChecked()** for **content projection**.
- Use **ngAfterViewInit() & ngAfterViewChecked()** for **view-related logic**.
- Use **ngOnDestroy()** to **clean up resources and subscriptions**.

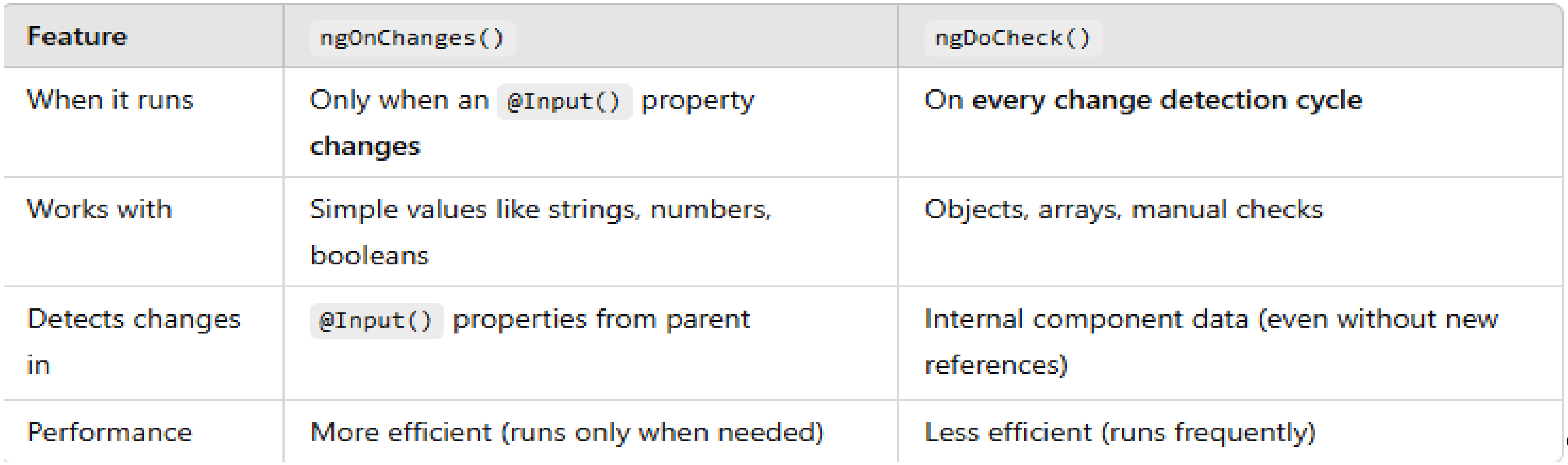
Component Lifecycle Hook

Key **Differences** Between **ngDoCheck()** and **ngAfterViewChecked()**



Feature	`ngDoCheck()`	`ngAfterViewChecked()`
When it runs	Every change detection cycle	After the view (DOM) is updated
Purpose	Detects **custom changes** in component data	Detects **view-related updates**
Works with	Objects, arrays, manual checks	`@ViewChild`, DOM updates
Use case	If `ngOnChanges()` **does not detect** a change	When you need to **track UI updates**

Key **Differences** Between **ngOnChanges()** and **ngDoCheck()**



Feature	`ngOnChanges()`	`ngDoCheck()`
When it runs	Only when an `@Input()` property **changes**	On **every change detection cycle**
Works with	Simple values like strings, numbers, booleans	Objects, arrays, manual checks
Detects changes in	`@Input()` properties from parent	Internal component data (even without new references)
Performance	More efficient (runs only when needed)	Less efficient (runs frequently)

Routing (Angular Routing)

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**.

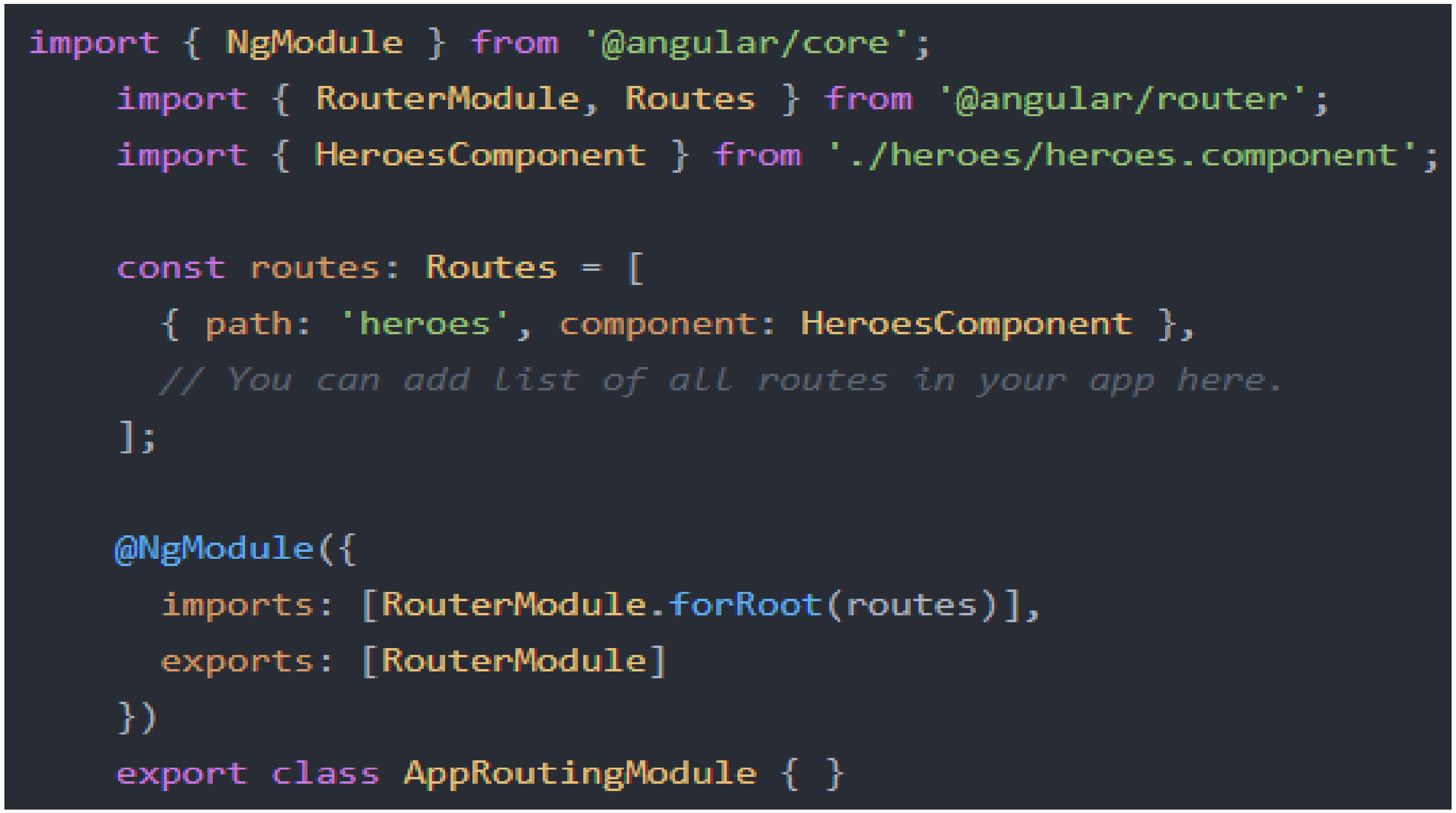
Routes tell the Angular Router which view to display when a user clicks a link.

A typical Angular Route has 2 properties

1. **path**: A string that matches the URL in the browser address bar.
2. **component**: The component that the router should create when navigating to this route

Routing (Angular Routing) app-routing.module.ts

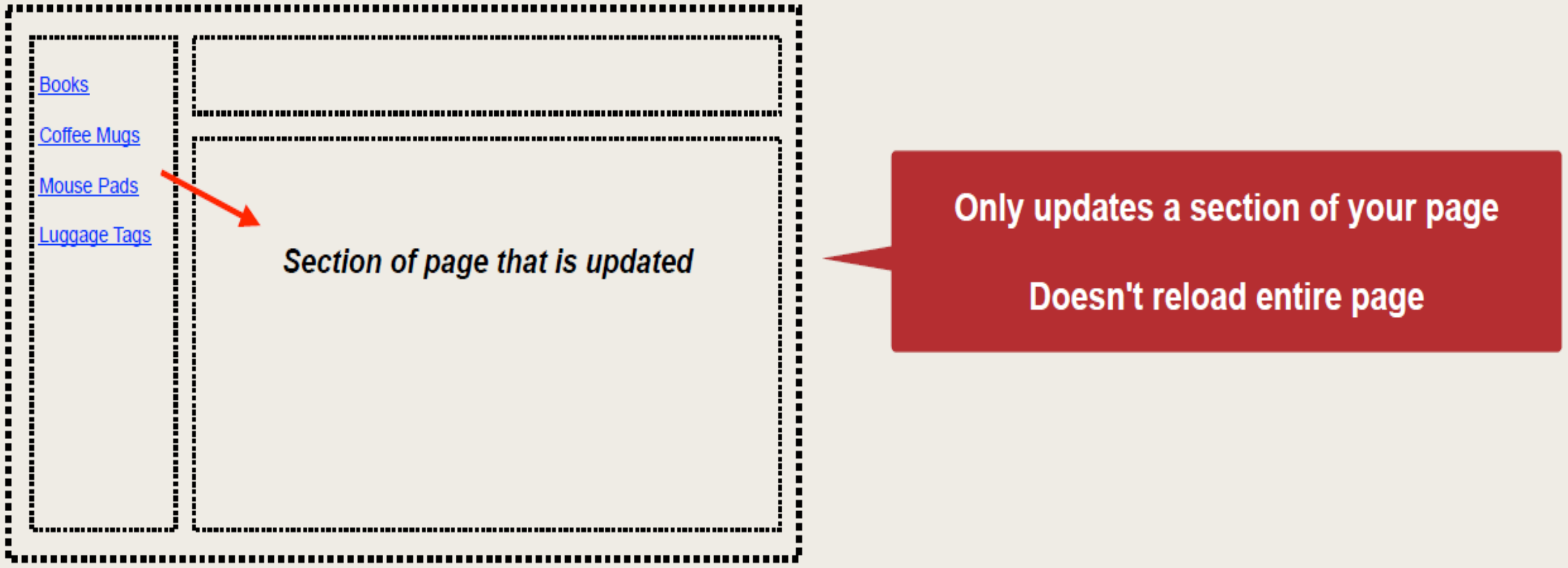
This tells the **router** to match that URL to path: ‘**heroes**’ and display the **HeroesComponent** when the URL is something like **http://localhost:4200/heroes**



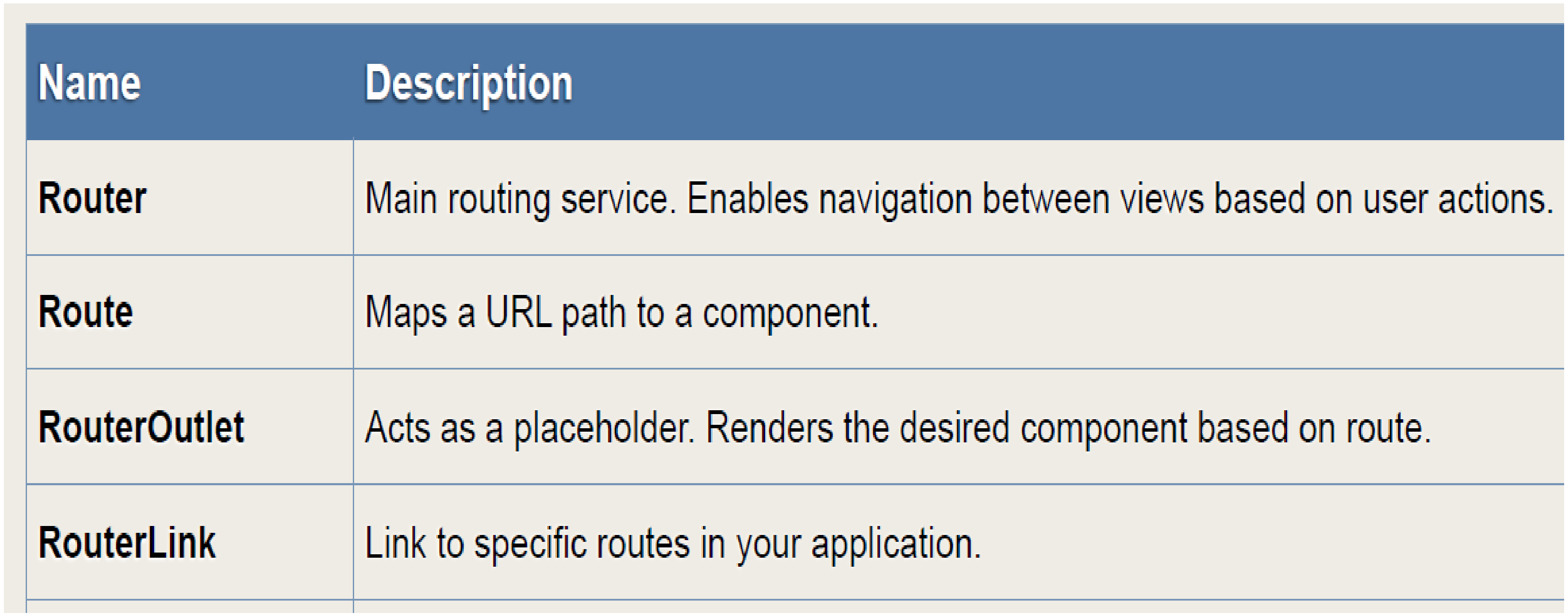
```
import { NgModule } from '@angular/core';
    import { RouterModule, Routes } from '@angular/router';
    import { HeroesComponent } from './heroes/heroes.component';

    const routes: Routes = [
      { path: 'heroes', component: HeroesComponent },
      // You can add list of all routes in your app here.
    ];

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

Angular Routing

- In Angular, you can add links in your application
- The links will route to other components in your application
- Angular routing will handle updating a view of your application


Books
Coffee Mugs
Mouse Pads
Luggage Tags
Section of page that is updated
Only updates a section of your page
Doesn't reload entire page

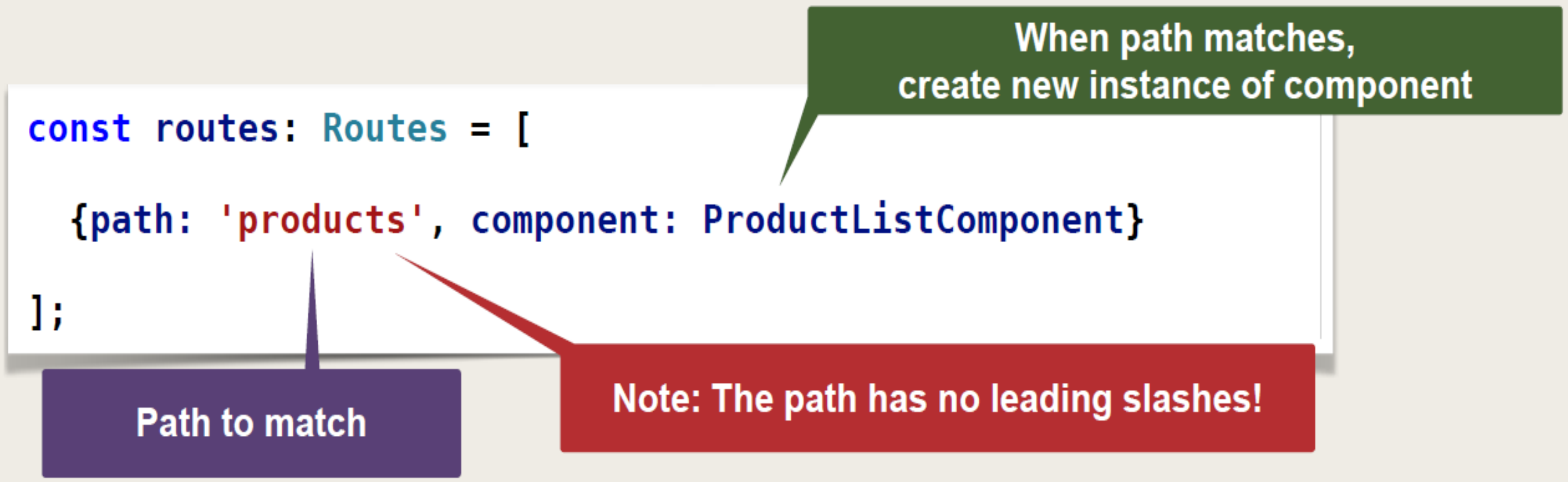
Angular Routing



Name	Description
Router	Main routing service. Enables navigation between views based on user actions.
Route	Maps a URL path to a component.
RouterOutlet	Acts as a placeholder. Renders the desired component based on route.
RouterLink	Link to specific routes in your application.

Define Routes (app-routing.module.ts)

- A route has a **path** and a reference to a **component**
- When the user selects the link for the route path
  - Angular will create a new instance of component



```
const routes: Routes = [

  {path: 'products', component: ProductListComponent}

];
```

When path matches,
create new instance of component
Path to match
Note: The path has no leading slashes!

Define Routes (app-routing.module.ts)

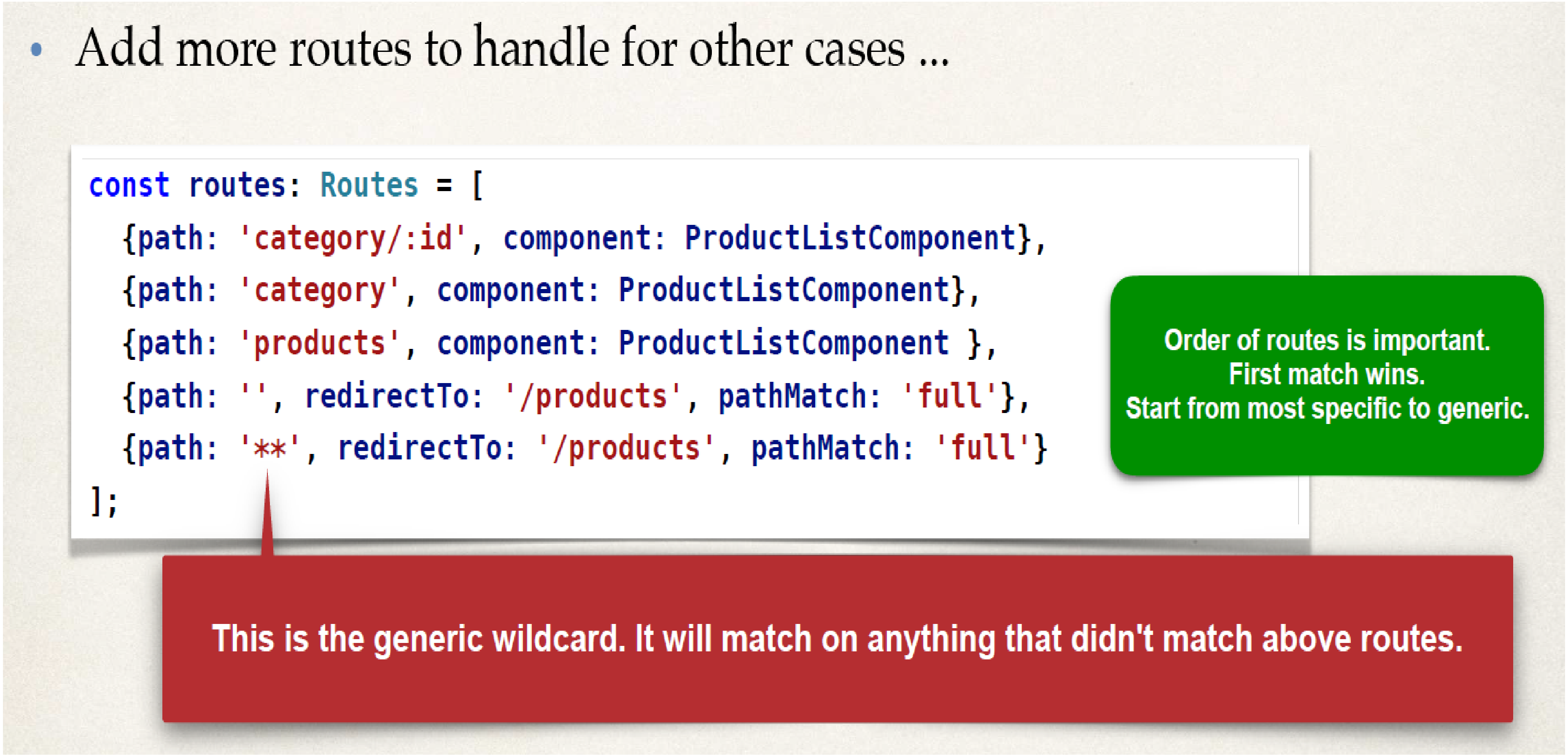
- Add route to show products for a given category **id**

```
const routes: Routes = [

  {path: 'category/:id', component: ProductListComponent},
  {path: 'products', component: ProductListComponent}

];
```

Category `id` parameter

The component can read this later and show products for this category

Define Routes (app-routing.module.ts)



- Add more routes to handle for other cases …

```
const routes: Routes = [
  {path: 'category/:id', component: ProductListComponent},
  {path: 'category', component: ProductListComponent},
  {path: 'products', component: ProductListComponent },
  {path: '', redirectTo: '/products', pathMatch: 'full'},
  {path: '**', redirectTo: '/products', pathMatch: 'full'}
];
```

Order of routes is important.
First match wins.
Start from most specific to generic.

This is the generic wildcard. It will match on anything that didn't match above routes.

Define Routes (app-routing.module.ts)

- Can add a custom PageNotFoundComponent ... for 404s

```
const routes: Routes = [
 ...
 {path: '**', component: PageNotFoundComponent}
];
```

Be sure to place this last.

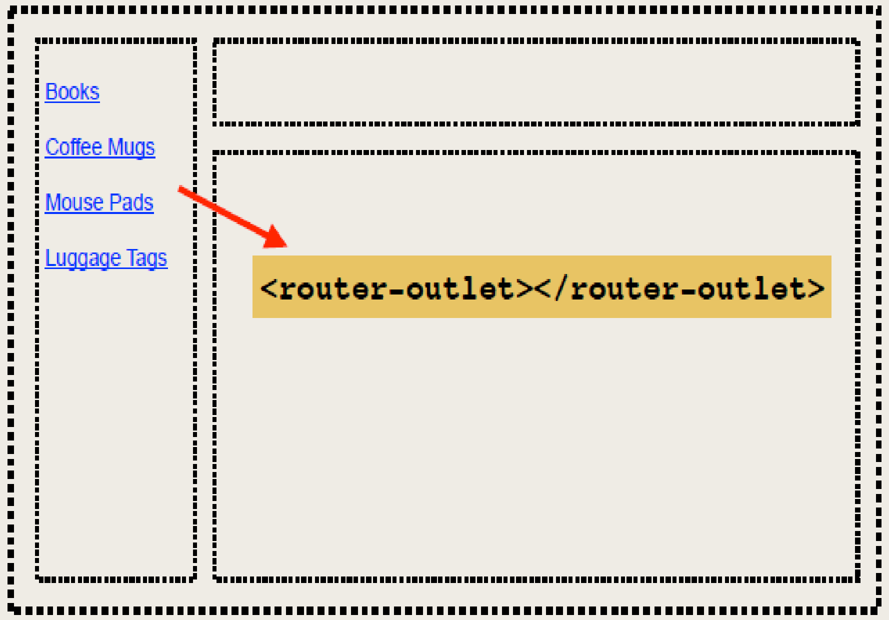
Order of routes is important.
First match wins.
Start from most specific to generic.

Custom component that you create

Can give any name and
provide your own custom view

Define the Router Outlet

- **Router Outlet** acts as a placeholder.
- Renders the desired component based on route.

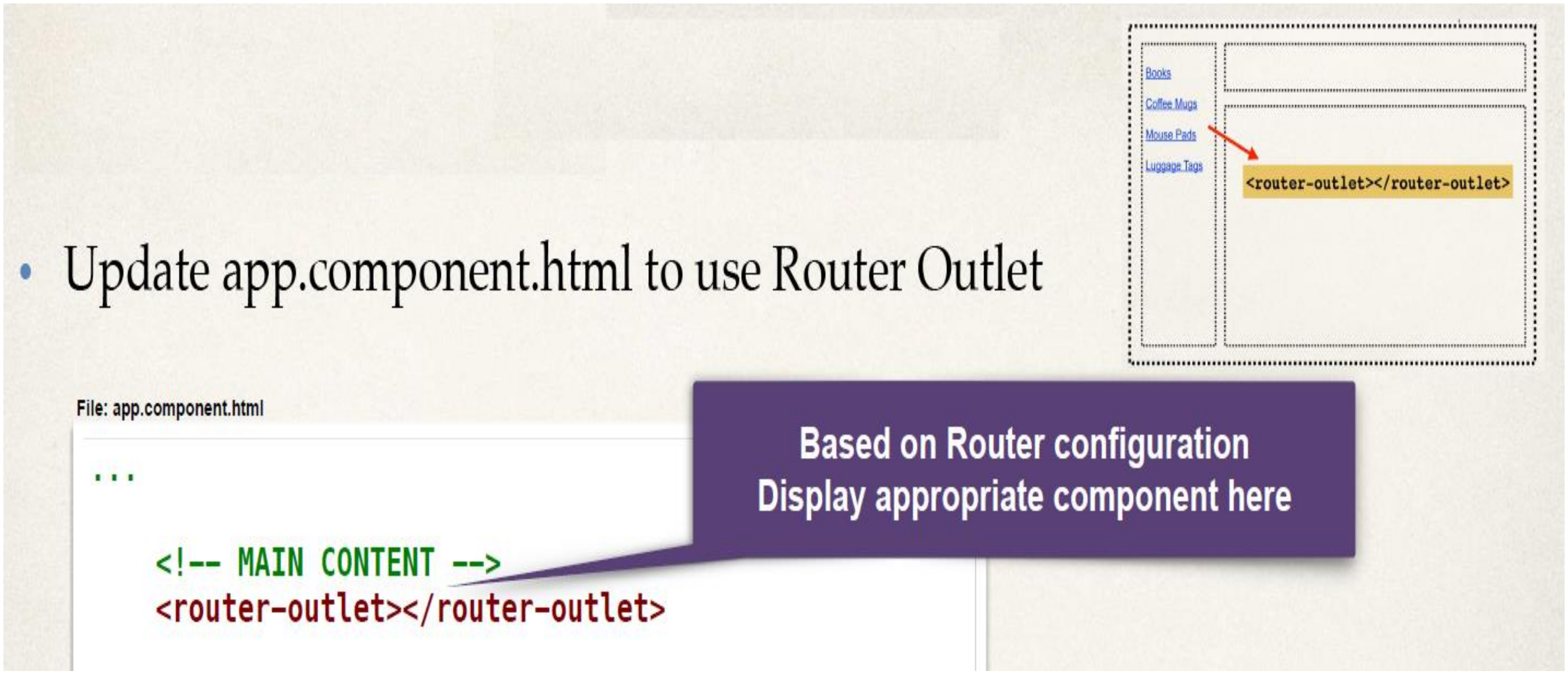


```
<router-outlet></router-outlet>
```

Only updates a section of your page

Doesn't reload entire page

Define the Router Outlet



- Update app.component.html to use Router Outlet

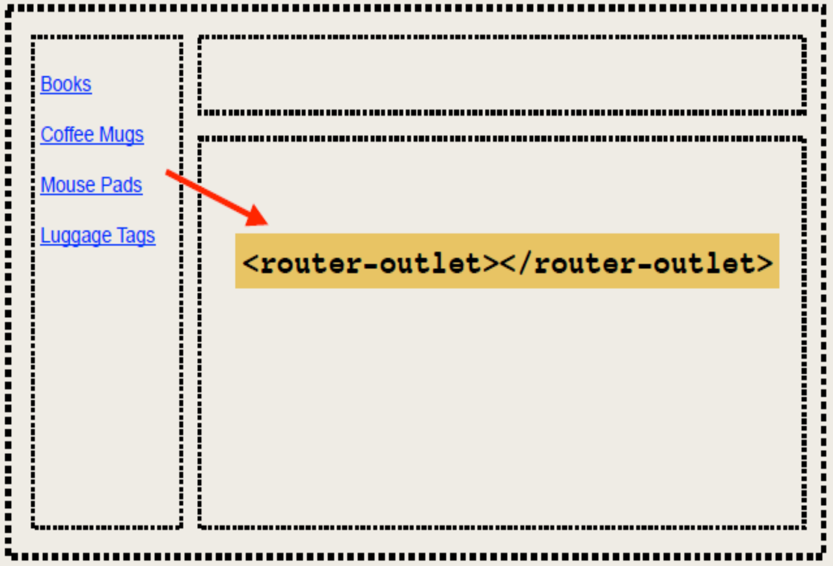
File: app.component.html

```
...

    <!-- MAIN CONTENT -->
    <router-outlet></router-outlet>
```

Based on Router configuration
Display appropriate component here

Define the Router Outlet

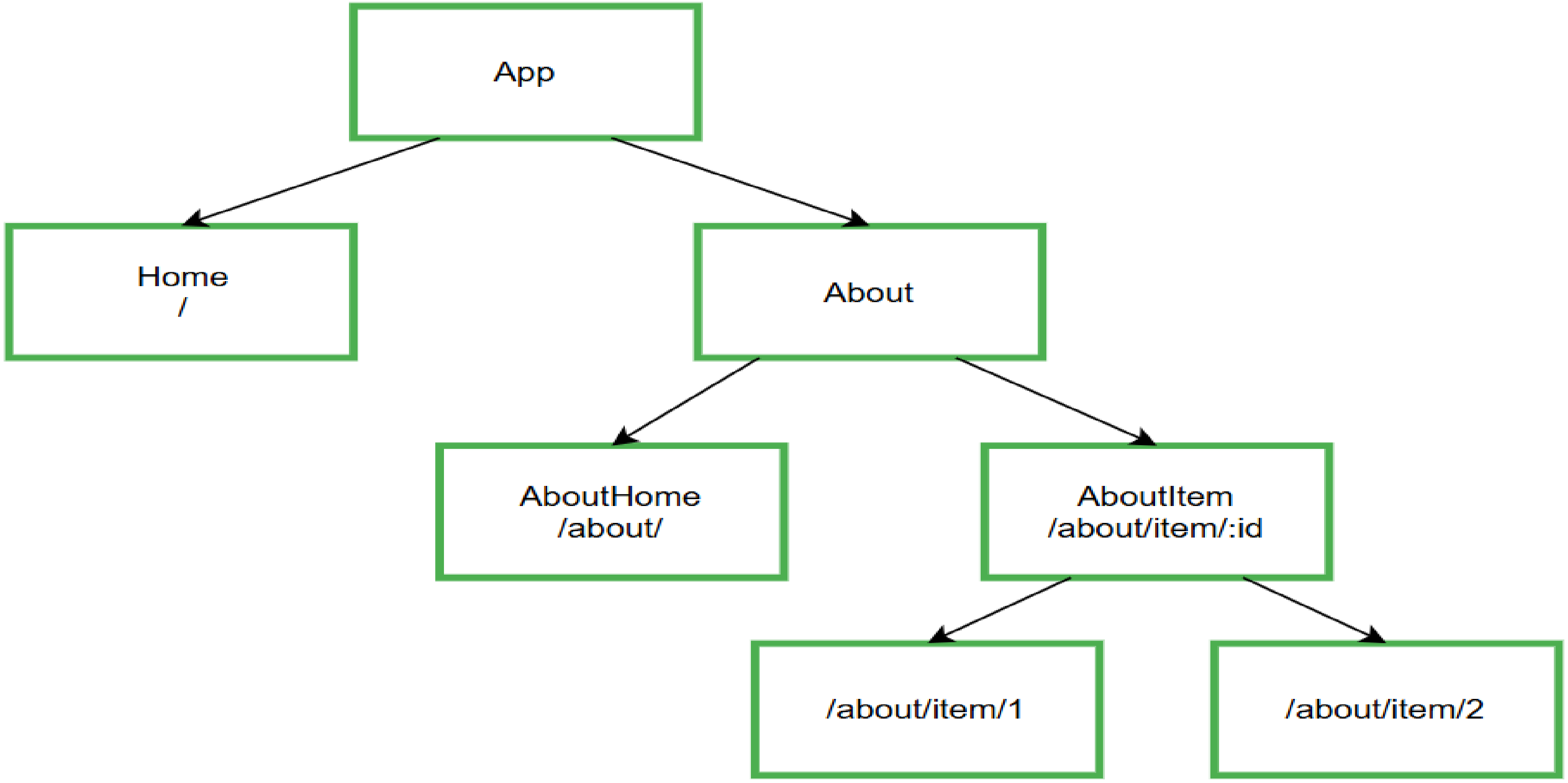
- When user clicks the link
  - ProductListComponent will appear in the location of `router-outlet`



Only updates a section of your page

Doesn't reload entire page

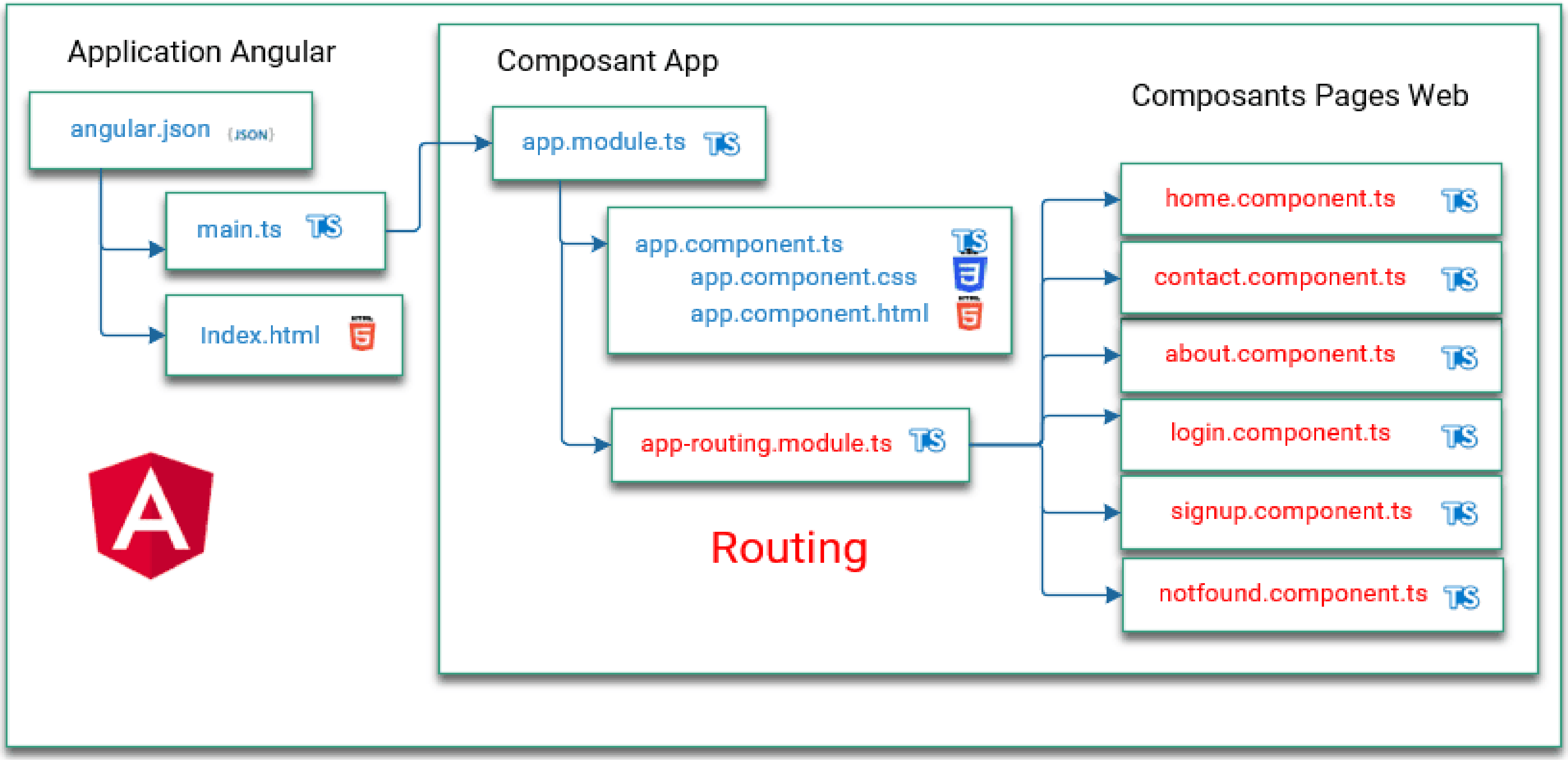
Angular Routing



In Angular which is used to build SPA, **URLs are not served from the server** and **never reload the page**. The URLs are strictly local in the browser and serve the page(components) from local. Every time you request a URL, the **Angular router navigates to the new component renders its template**, and updates the history and URL.

Angular Routing

The **angular@router** library was imported into a **module dedicated** to routing.
This module was named **app-routing.module.ts**
Routing is managed via 3 basic files: **app.module.ts, app.component.ts, app-routing.module.ts**



Angular Routing

src/app/app.routes.ts

```
import { Routes } from '@angular/router';

import { HomeComponent } from './pages/general/home/home.component';

import { LoginComponent } from './pages/general/login/login.component';
import { SignupComponent } from './pages/general/signup/signup.component';
import { NotFoundComponent } from './pages/general/not-found/not-found.component';

import { AboutComponent } from './pages/general/about/about.component';
import { ContactComponent } from './pages/general/contact/contact.component';

export const routes: Routes = [
    { path: '', component: HomeComponent, },

    { path: 'login', component: LoginComponent },
    { path: 'signup', component: SignupComponent },

    { path: 'about', component: AboutComponent },
    { path: 'contact', component: ContactComponent },

    { path: '**', component: NotFoundComponent }
];
```

Angular Routing

src/app/app.component.html

```html
<div class="app">
  <header>
    <section>
      <h1>{{ title }}</h1>
    </section>
    <nav>
      <h2>3 Links with Routes</h2>
      <ul>
        <li><a routerLink="/">Home</a></li>
        <li><a routerLink="/login">Login</a></li>
        <li><a routerLink="/signup">Signup</a></li>
      </ul>
      <h3>2 Links with Child Routes</h3>
      <ul>
        <li><a routerLink="/about">About</a></li>
        <li><a routerLink="/contact">Contact</a></li>
      </ul>
    </nav>
  </header>

  <main>
    <h4>Routes Result</h4>
    <router-outlet></router-outlet>
  </main>
```

Route Guards

In Angular, **guards** are special classes used to control and manage access to different parts of an application.

Guards decide whether a user can navigate to a particular route or perform certain actions based on specific conditions, like checking if the user is logged in or has the necessary permissions.

Guards act as gatekeepers, allowing or preventing access to different parts of the application, **ensuring security** and **controlling the flow of navigation within the app**.

Types of Route Guards in Angular

- (1) CanActivate
- (2) CanActivateChild
- (3) CanDeactivate
- (4) CanLoad

(1) CanActivate

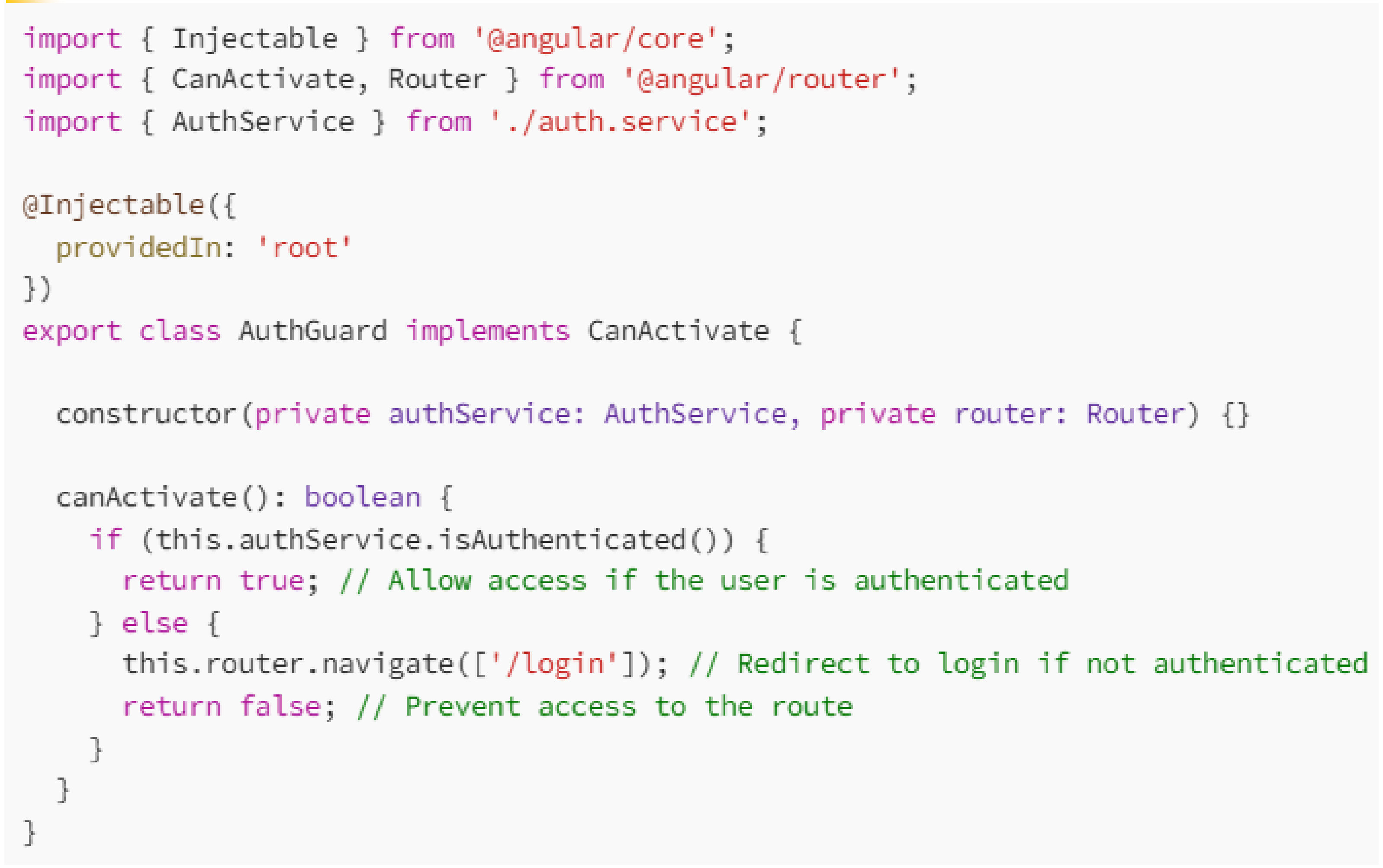
•Determines if a route can be activated and allows navigation based on certain conditions.
•Implemented using CanActivate interface.

Use Case — Authentication Guard:

Let’s say you have an application with a dashboard page that **should only be accessible to authenticated users**. The **CanActivate** guard **ensures that only authenticated users can access this page, redirecting unauthenticated users to the login page**.

Let’s assume that there’s an **AuthService** that provides a method **isAuthenticated()** to check if the user is logged in.

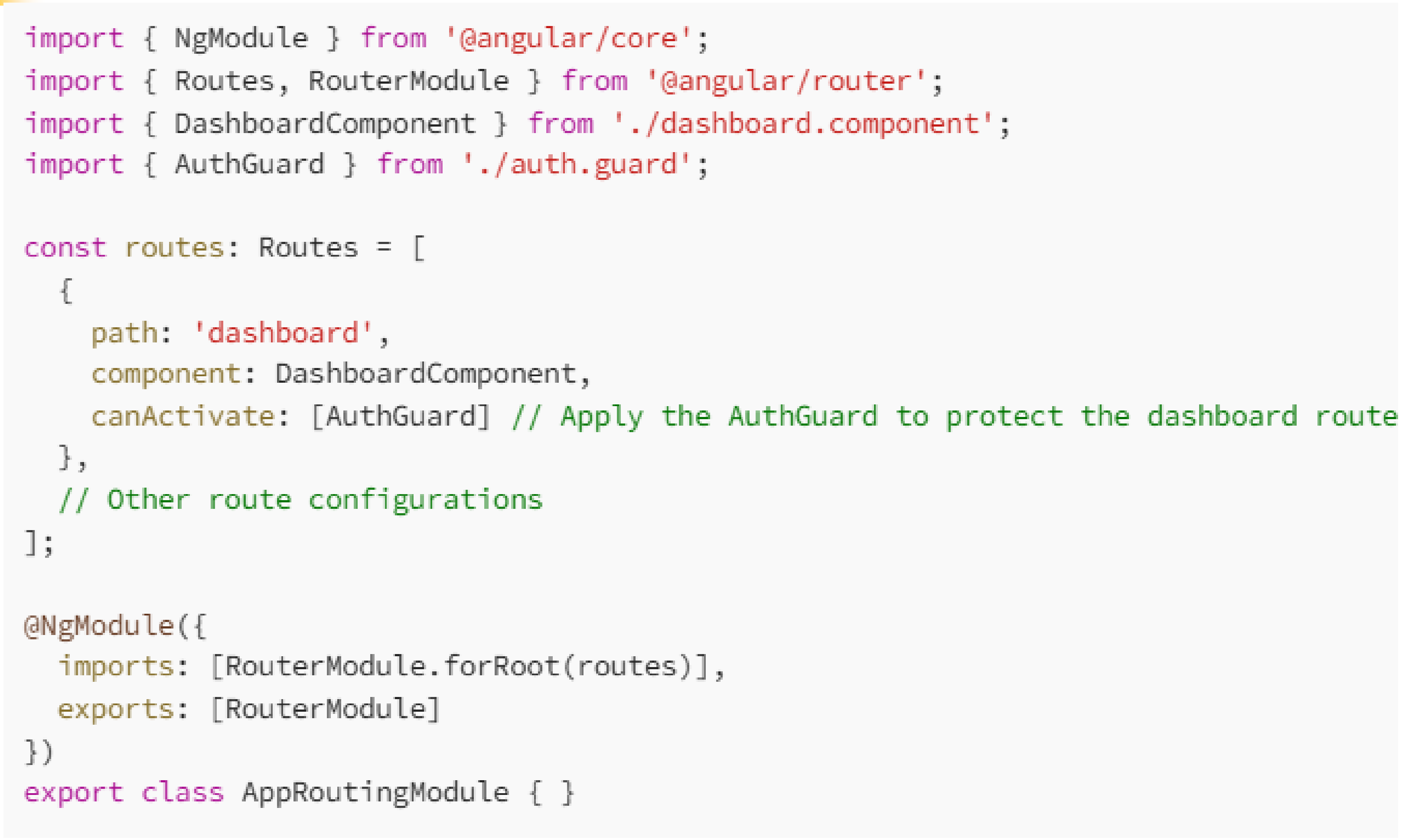
AuthGuard Service Example Code



```
import { Injectable } from '@angular/core';
import { CanActivate, Router } from '@angular/router';
import { AuthService } from './auth.service';

@Injectable({
  providedIn: 'root'
})
export class AuthGuard implements CanActivate {

  constructor(private authService: AuthService, private router: Router) {}

  canActivate(): boolean {
    if (this.authService.isAuthenticated()) {
      return true; // Allow access if the user is authenticated
    } else {
      this.router.navigate(['/login']); // Redirect to login if not authenticated
      return false; // Prevent access to the route
    }
  }
}
```

CanActivate Route Guards

Route Configuration (app-routing.module.ts)



```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { DashboardComponent } from './dashboard.component';
import { AuthGuard } from './auth.guard';

const routes: Routes = [
  {
    path: 'dashboard',
    component: DashboardComponent,
    canActivate: [AuthGuard] // Apply the AuthGuard to protect the dashboard route
  },
  // Other route configurations
];

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

CanActivate Route Guards

(2) CanActivateChild

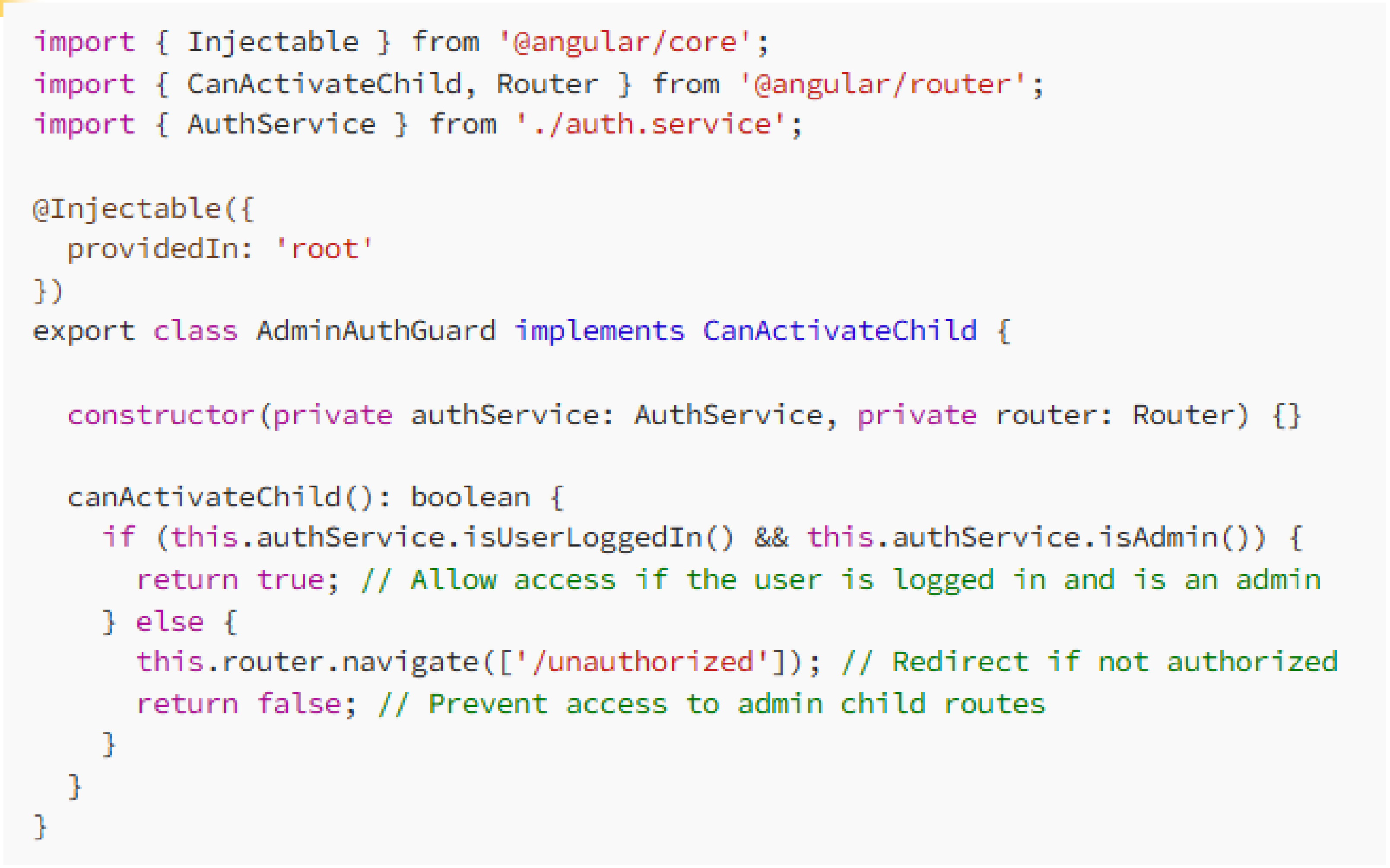
•Similar to CanActivate but controls the activation of child routes.

•Implemented using CanActivateChild interface.

Use Case — Admin Panel Access

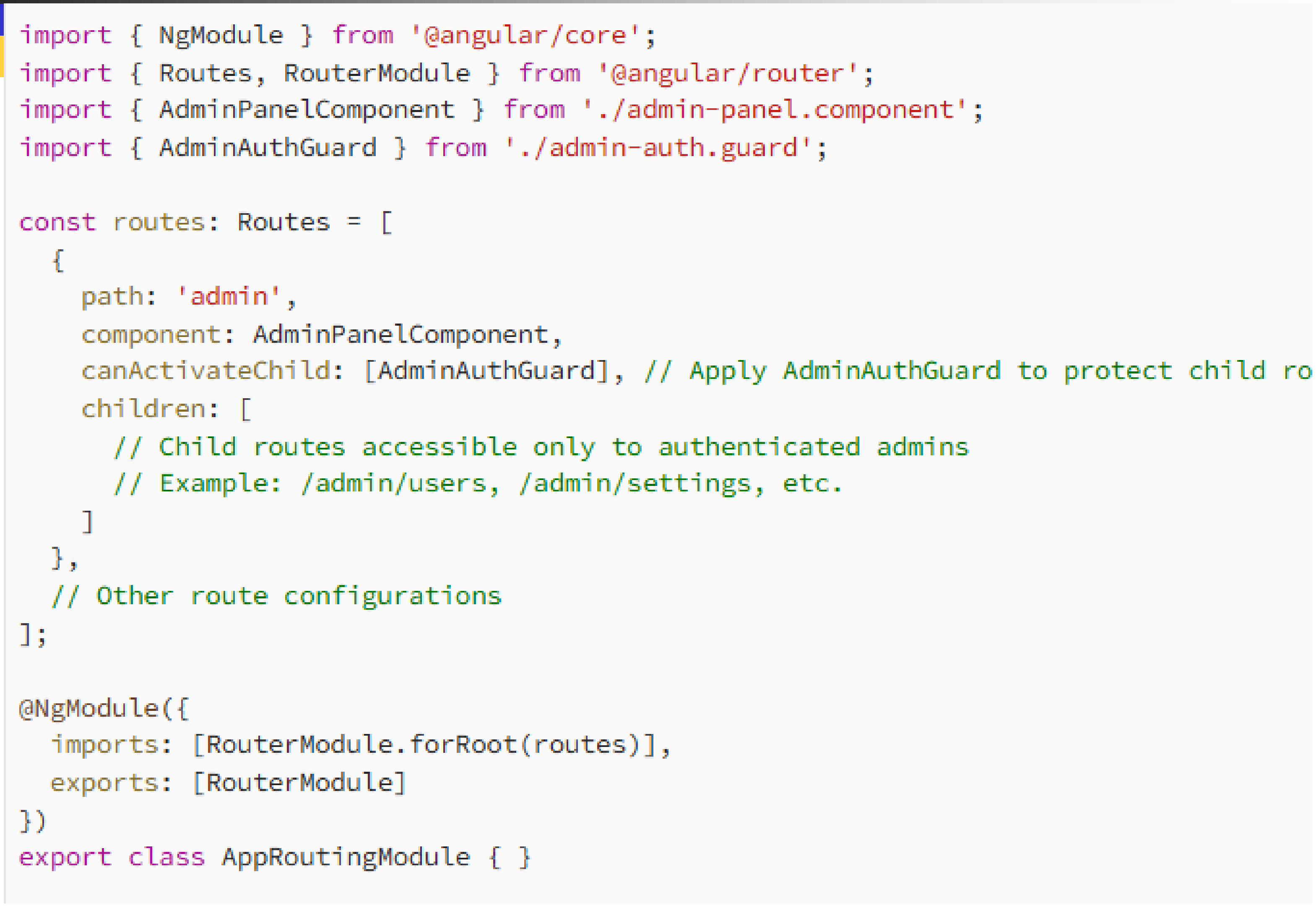
Imagine an **admin panel** within an application **containing multiple child routes** that should only be **accessible to authenticated administrators**.

AdminAuthGuard Service Example Code



```
import { Injectable } from '@angular/core';
import { CanActivateChild, Router } from '@angular/router';
import { AuthService } from './auth.service';

@Injectable({
  providedIn: 'root'
})
export class AdminAuthGuard implements CanActivateChild {

  constructor(private authService: AuthService, private router: Router) {}

  canActivateChild(): boolean {
    if (this.authService.isUserLoggedIn() && this.authService.isAdmin()) {
      return true; // Allow access if the user is logged in and is an admin
    } else {
      this.router.navigate(['/unauthorized']); // Redirect if not authorized
      return false; // Prevent access to admin child routes
    }
  }
}
```

Route Configuration (app-routing.module.ts)



```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { AdminPanelComponent } from './admin-panel.component';
import { AdminAuthGuard } from './admin-auth.guard';

const routes: Routes = [
  {
    path: 'admin',
    component: AdminPanelComponent,
    canActivateChild: [AdminAuthGuard], // Apply AdminAuthGuard to protect child ro
    children: [
      // Child routes accessible only to authenticated admins
      // Example: /admin/users, /admin/settings, etc.
    ]
  },
  // Other route configurations
];

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

CanActivateChild Route Guards

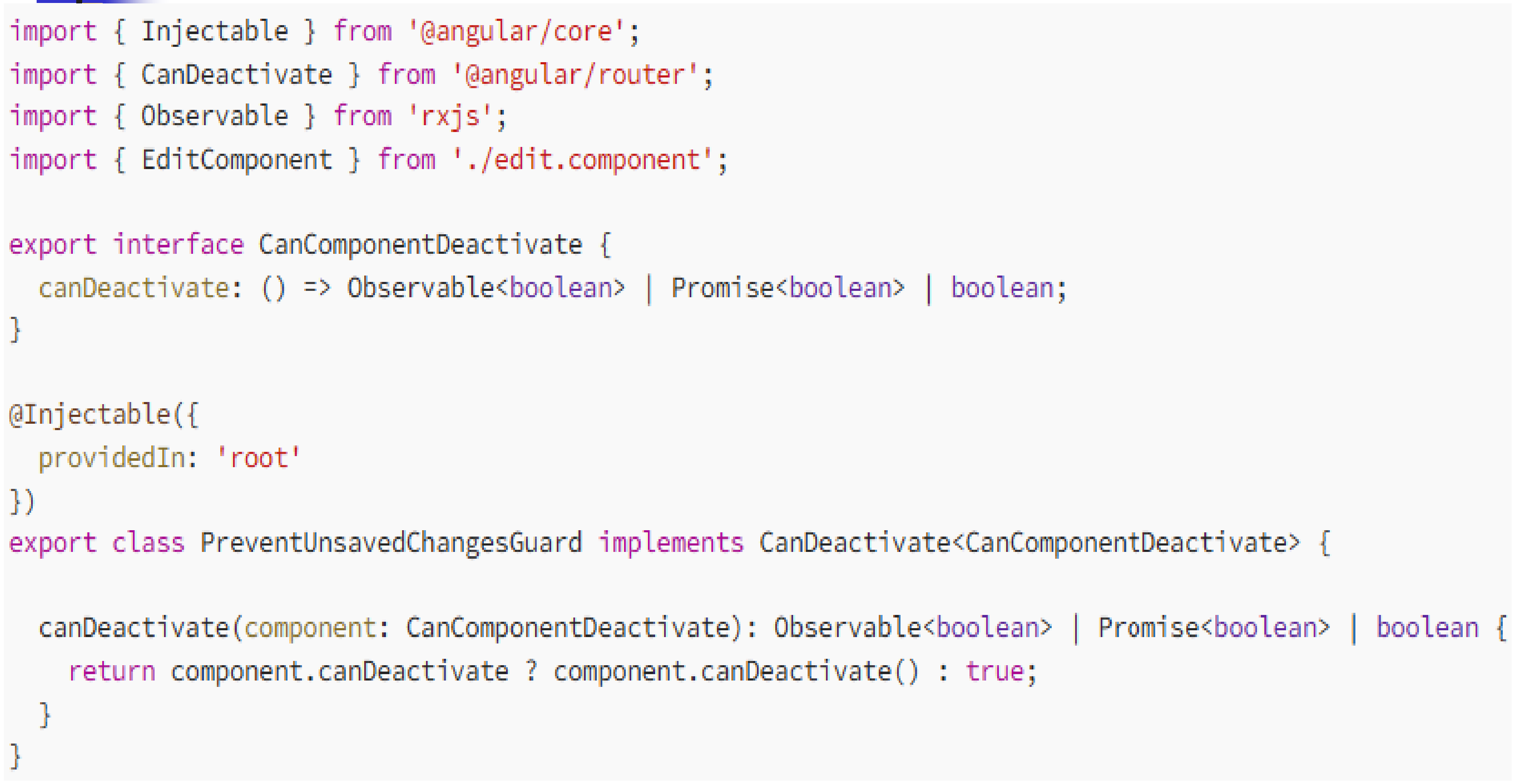
(3) CanDeactivate

•Checks if a route can be deactivated, often used to confirm navigation away from a route.
•Implemented using CanDeactivate interface.

Use Case — Preventing Unsaved Changes:

Consider a **form-editing feature** within an application where users can **modify certain data**. You want to **ensure that users are prompted with a confirmation before leaving the editing page** if they've **made changes that haven't been saved.**

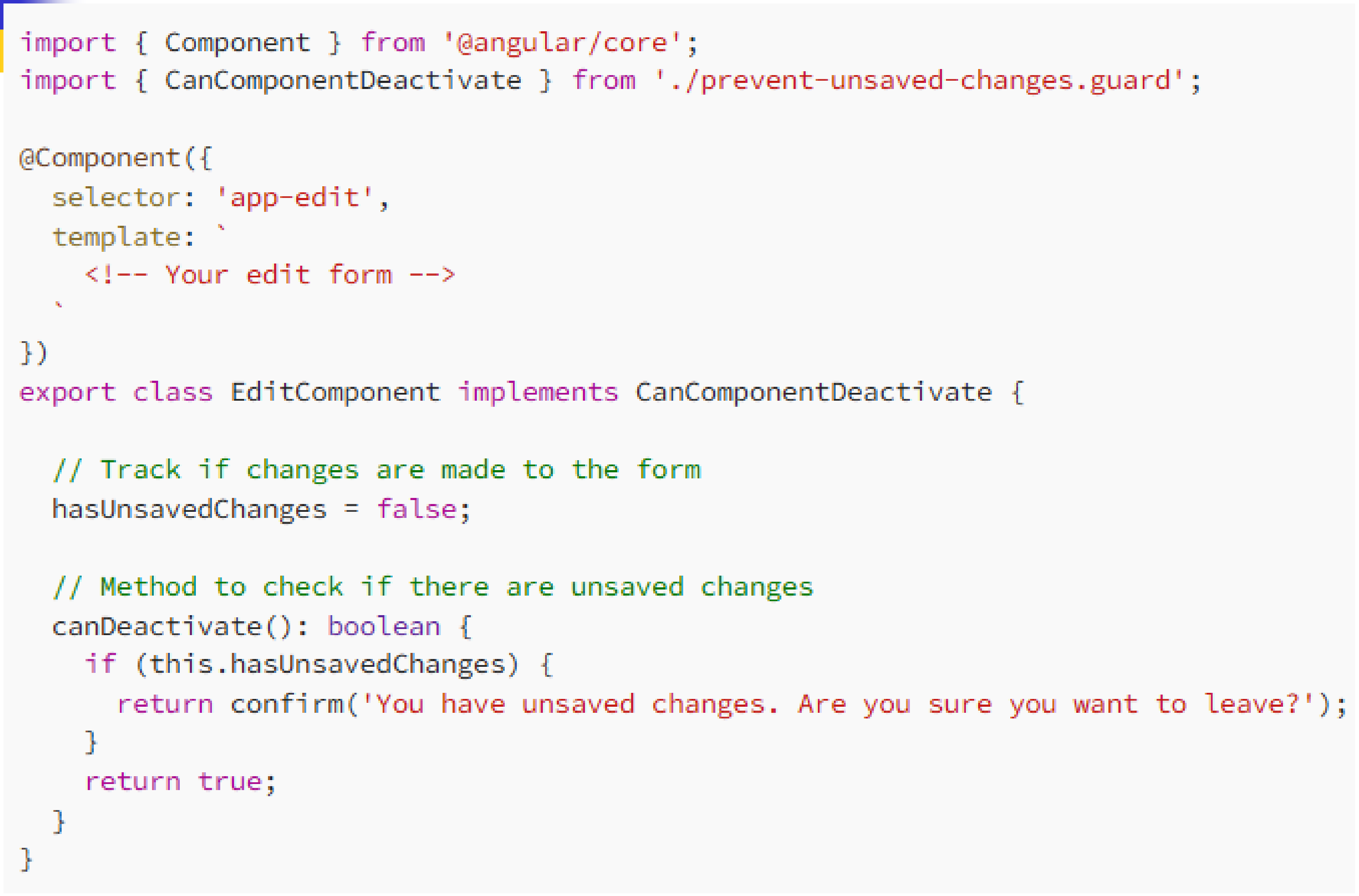
Deactivation Guard Service Example Code



```
import { Injectable } from '@angular/core';
import { CanDeactivate } from '@angular/router';
import { Observable } from 'rxjs';
import { EditComponent } from './edit.component';

export interface CanComponentDeactivate {
  canDeactivate: () => Observable<boolean> | Promise<boolean> | boolean;
}

@Injectable({
  providedIn: 'root'
})
export class PreventUnsavedChangesGuard implements CanDeactivate<CanComponentDeactivate> {

  canDeactivate(component: CanComponentDeactivate): Observable<boolean> | Promise<boolean> | boolean {
    return component.canDeactivate ? component.canDeactivate() : true;
  }
}
```

CanDeactivate Route Guards

Component Implementation



```
import { Component } from '@angular/core';
import { CanComponentDeactivate } from './prevent-unsaved-changes.guard';

@Component({
  selector: 'app-edit',
  template: `
    <!-- Your edit form -->
  `
})
export class EditComponent implements CanComponentDeactivate {

  // Track if changes are made to the form
  hasUnsavedChanges = false;

  // Method to check if there are unsaved changes
  canDeactivate(): boolean {
    if (this.hasUnsavedChanges) {
      return confirm('You have unsaved changes. Are you sure you want to leave?');
    }
    return true;
  }
}
```

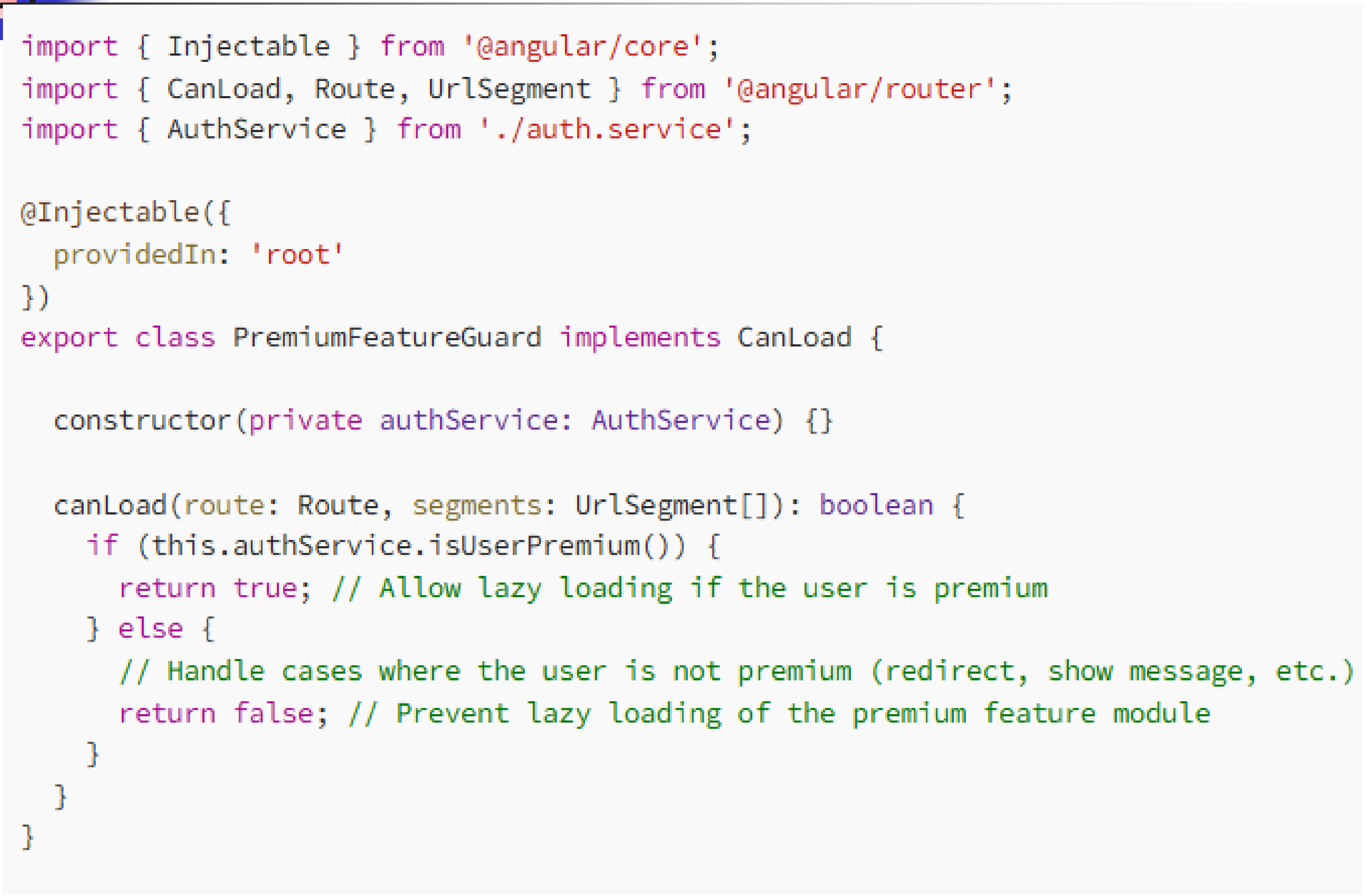
(4) CanLoad

•Prevents a module from being **loaded lazily** until certain conditions are met.
•Implemented using CanLoad interface

Use Case — Role-Based Module Loading:

Consider a scenario where your application **has a premium feature module that should only be loaded for users with a specific subscription or premium status**.

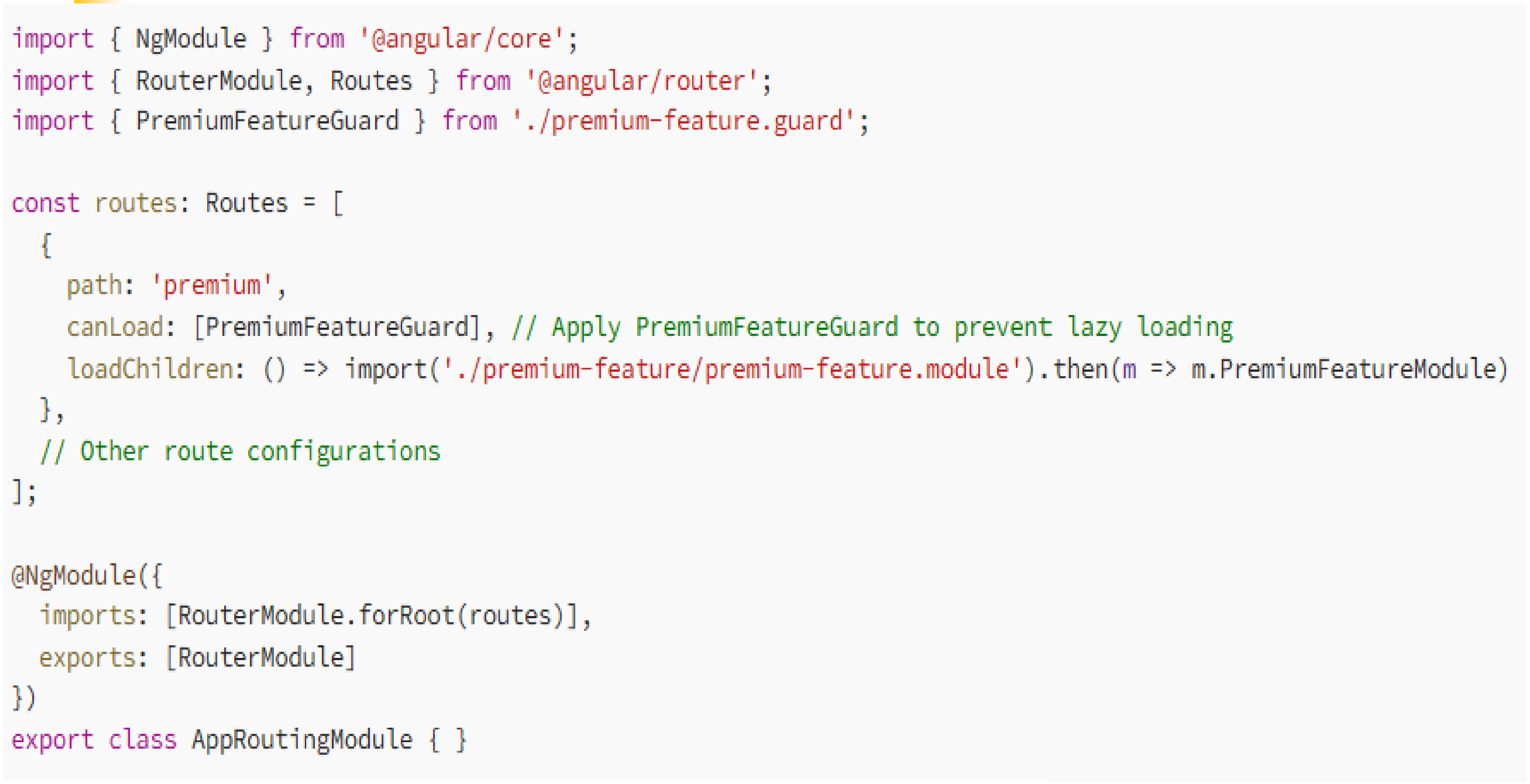
CanLoad Guard Service



```
import { Injectable } from '@angular/core';
import { CanLoad, Route, UrlSegment } from '@angular/router';
import { AuthService } from './auth.service';

@Injectable({
  providedIn: 'root'
})
export class PremiumFeatureGuard implements CanLoad {

  constructor(private authService: AuthService) {}

  canLoad(route: Route, segments: UrlSegment[]): boolean {
    if (this.authService.isUserPremium()) {
      return true; // Allow lazy loading if the user is premium
    } else {
      // Handle cases where the user is not premium (redirect, show message, etc.)
      return false; // Prevent lazy loading of the premium feature module
    }
  }
}
```

CanLoad Route Guards

Lazy Loaded Module Configuration



```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { PremiumFeatureGuard } from './premium-feature.guard';

const routes: Routes = [
  {
    path: 'premium',
    canLoad: [PremiumFeatureGuard], // Apply PremiumFeatureGuard to prevent lazy loading
    loadChildren: () => import('./premium-feature/premium-feature.module').then(m => m.PremiumFeatureModule)
  },
  // Other route configurations
];

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

Lazy Loading

Lazy loading is a technique in Angular that **allows you to load modules, components, or other assets** only when **they are actually needed**, rather **than loading them all at once during the initial application startup**.

This can help **improve the performance and reduce the initial loading time of your Angular application.**

Lazy loading works by **splitting your application into multiple modules**, and then **loading these modules on demand as the user navigates through the application**.

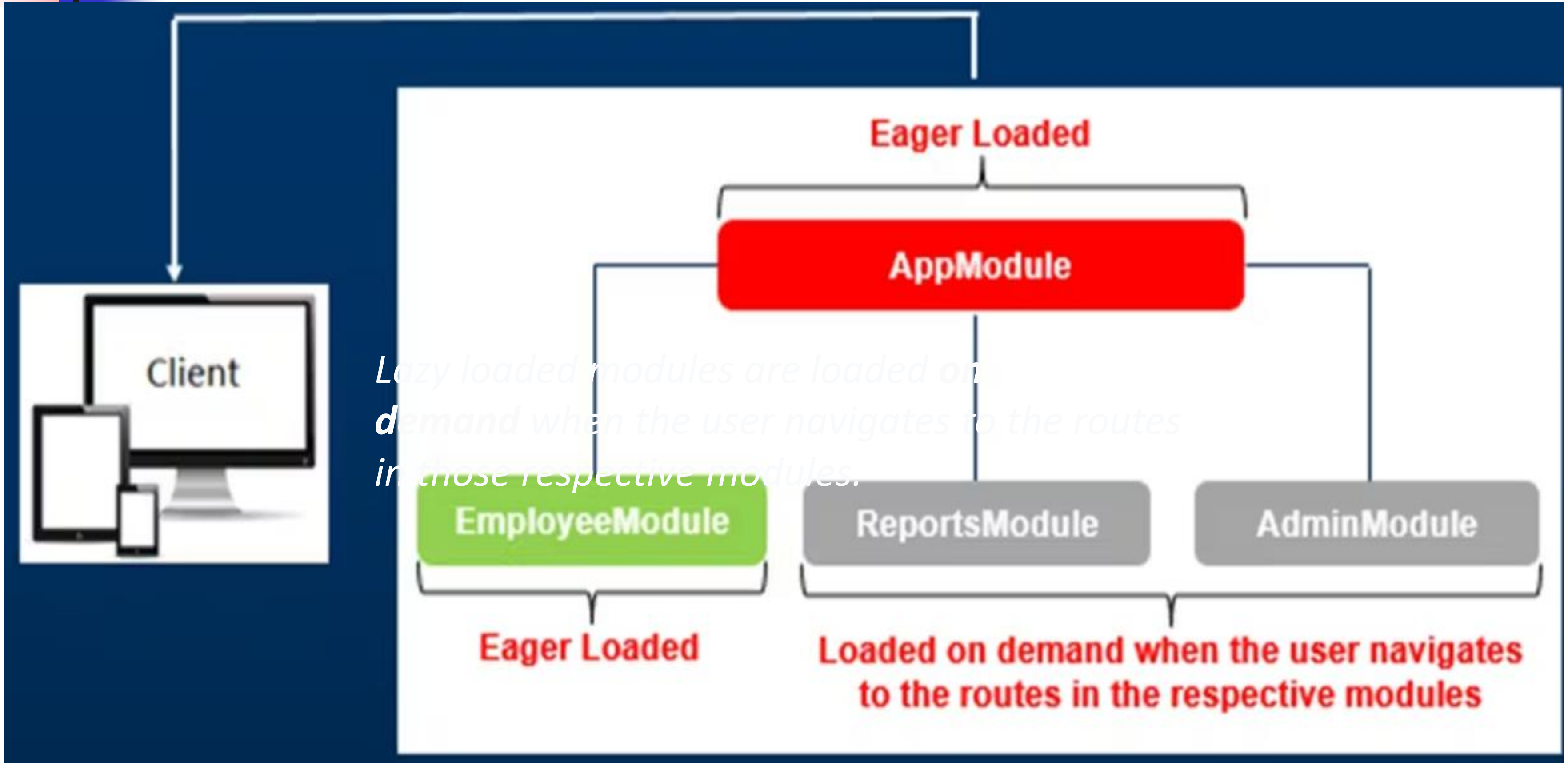
This way, **the initial bundle size is reduced, and the user only downloads the necessary code when it is required.**

Eager Loading

Eager loading is the **default way of loading modules in Angular**. In this approach, **modules are loaded eagerly** when the **main application starts**. All the code from these modules is included in the main bundle, **increasing the initial load time.**

With eager loading, **all the modules must be loaded before the application starts**. So depending on the number of modules and the internet connection speed, **the first request may take a long time,** but **the subsequent requests from that same client will be faster, because all the modules are already downloaded on to the client machine.**

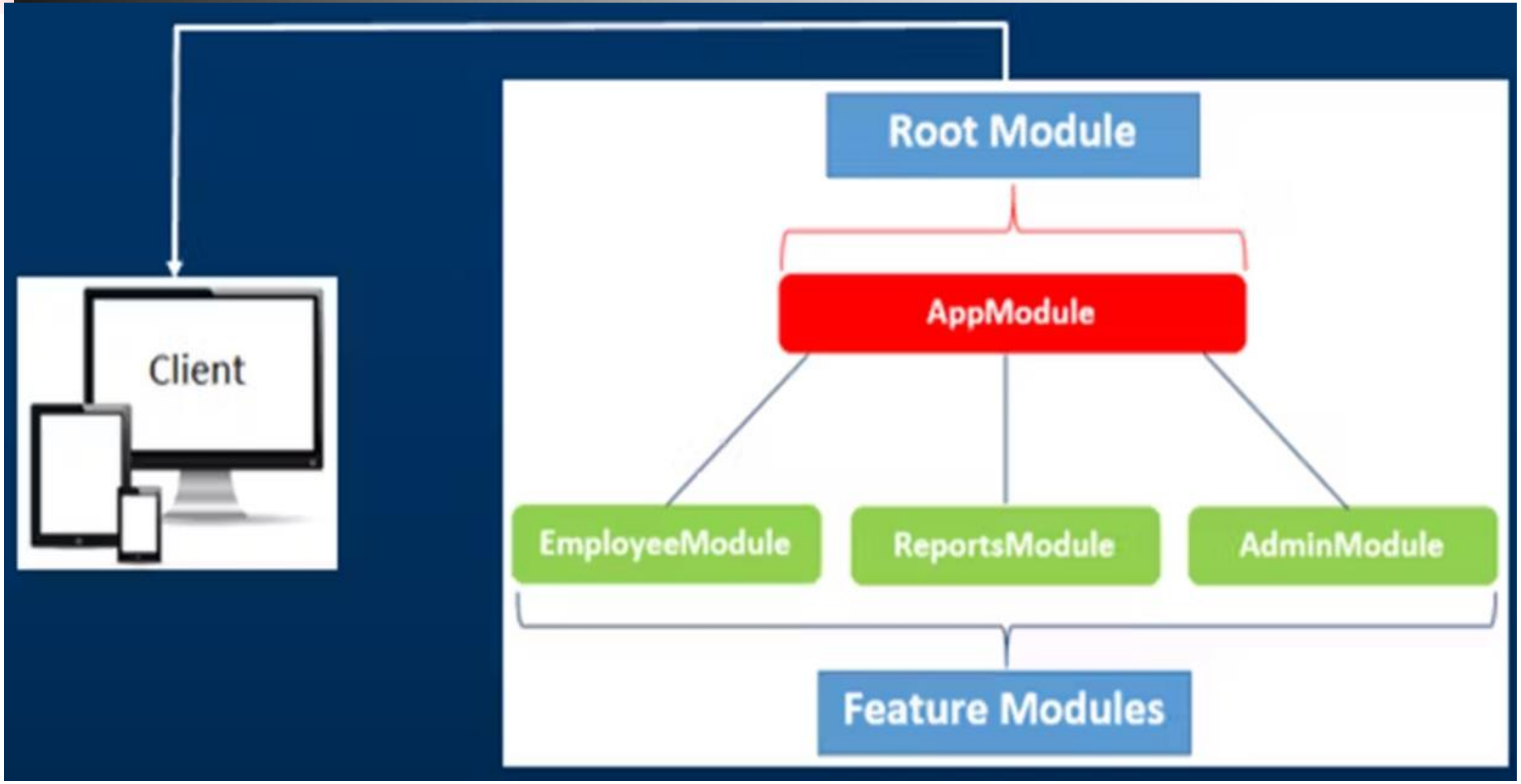
Lazy Loading


Client
Eager Loaded
AppModule
EmployeeModule
ReportsModule
AdminModule
Eager Loaded
Loaded on demand when the user navigates to the routes in the respective modules

Lazy loaded modules are loaded on ***demand*** *when the user navigates to the routes in those respective modules.*

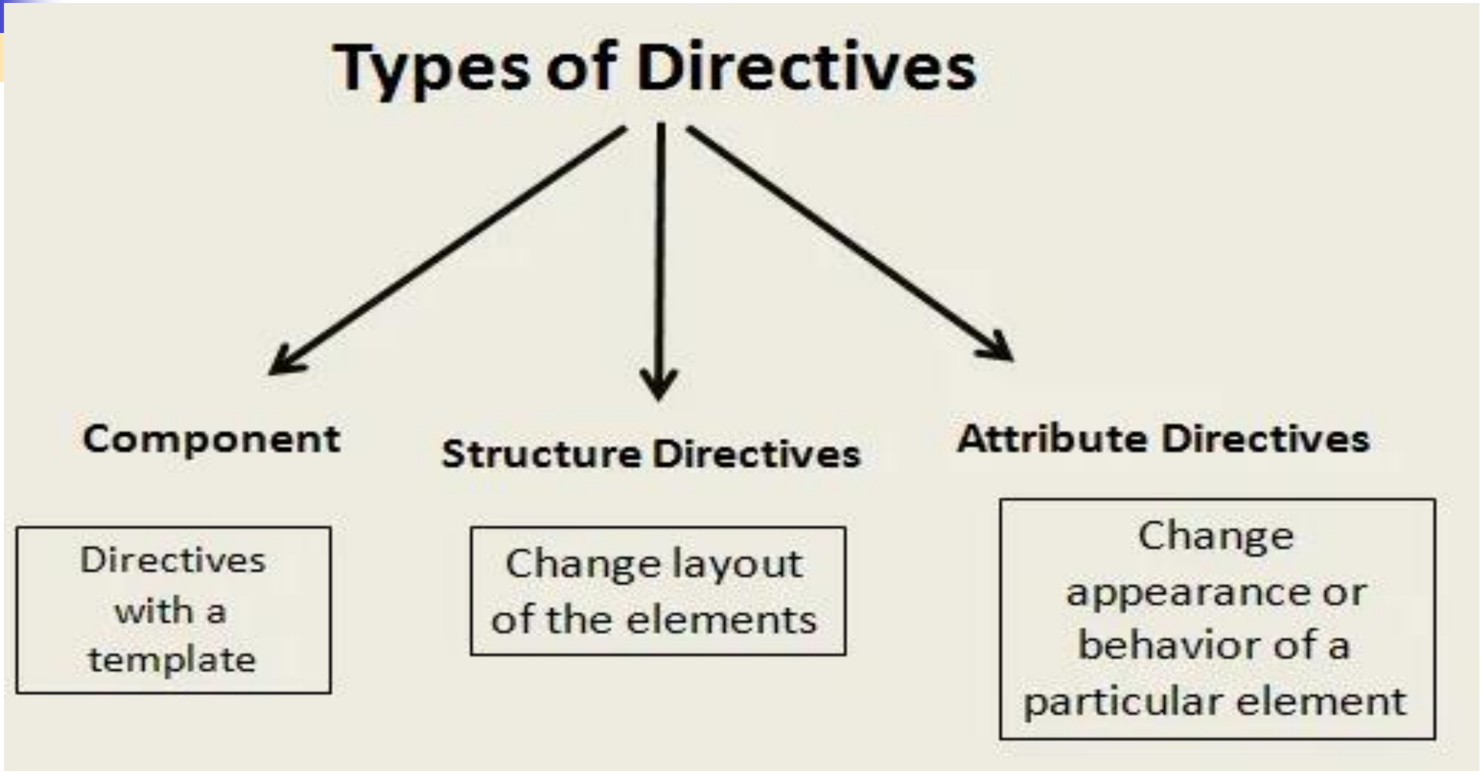
Only the **RootModule** and **EmployeeModule** are loaded **when the application starts**. The rest of the modules i.e **ReportsModule** and **AdminModule** are configured to be **lazy loaded** so **they are not loaded when the application starts**. Because of this reduced download size, the application initial load time is reduced to a great extent

Eager Loading



With **Eager Loading all the modules** must be downloaded onto the client machine before the application starts. By default, the angular modules are eagerly loaded. The root application module (AppModule) is always eagerly loaded

Directives


Types of Directives
Component
Structure Directives
Attribute Directives
Directives with a template
Change layout of the elements
Change appearance or behavior of a particular element

Directives are markers on a DOM element that tells Angular's HTML compiler ($compile) to attach a specified behavior to that DOM element or transform the DOM element

Structural Directives

Structural Directives are responsible for changing the structure of the DOM. They work by adding or removing the elements from the DOM

The Structural Directive's name always starts with an asterisk(*) prefix

The three most popular built-in Structural Directives Angular provides are **NgIf**, **NgFor**, and **NgSwitch**.

NgIf Directive is a structural directive used to add elements into the DOM according to some condition.

```
<p *ngIf="show">I am a Structural Directive</p>
```

```
<div *ngFor="let hero of heroes; let i=index; let odd=odd; trackBy: trackById"
  class="d-flex">
  <p class="text-primary">{{i}} Sinan</p>
        <p class="text-secondary">997</p>
</div>
```

Structural Directives

The **ngFor directive** is a structural directive in Angular that is used to render a list of items based on a collection in the component. The **ngFor** directive iterates over each product in the collection and generates a template for each product.

example of using the **ngFor** directive to display a list of products in a component:

```
//This code goes into the html file
<ul>
  <li *ngFor="let product of products">
    {{product}}
  </li>
</ul>
```

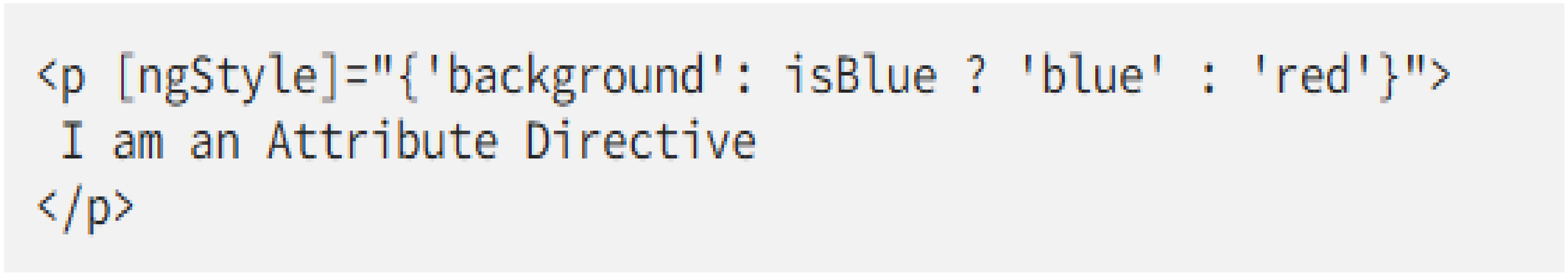
```
//This code goes into the typescript file
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
})
export class AppComponent {

products: string = ["Iphone 13 pro max", "Samsung A51"];

}
```

Attribute Directives

Attribute Directives are responsible for manipulating the appearance and behavior of DOM elements.

We can use **attribute directives** to **change the style of DOM elements**. These directives are also used to hide or show particular DOM elements conditionally.

Angular provides many built-in Attribute Directives like **NgStyle**, **NgClass**, etc.



```
<p [ngStyle]="{'background': isBlue ? 'blue' : 'red'}">
 I am an Attribute Directive
</p>
```

Components

Components are directives with templates. The only difference between Components and the other two types of directives is the Template.

Attribute and Structural Directives don't have Templates.

Creating a Custom Attribute Directive

Creating a custom directive is just like creating an Angular component. To create a custom directive we have to replace **@Component decorator** with **@Directive decorator**.

Directive example: Highlight the selected DOM element by setting an element's background color.

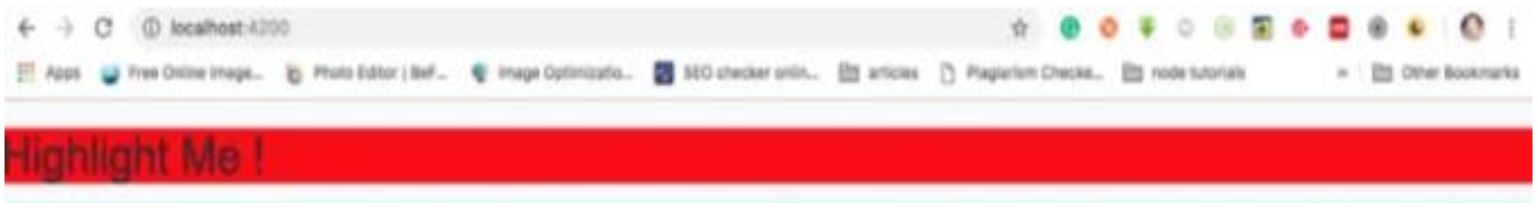
Create an ***app-highlight.directive.ts*** file in ***src/app* folder** and add the code snippet below.

```
import { Directive, ElementRef } from '@angular/core';
@Directive({
    selector: '[appHighlight]'
})
export class HighlightDirective {
    constructor(private eleRef: ElementRef) {
        eleRef.nativeElement.style.background = 'red';
    }
}
```

Creating a Custom Attribute Directive

add the ***appHightlight*** directive in the ***app.component.html*** but you can use it anywhere in the application.

```
<h1 appHightlight>Highlight Me !</h1>
```



Creating a Custom Structural Directive

implement the ***appNot*** directive which will work just opposite of ***ngIf***.

create ***app-not.directive.ts*** file in the ***src/app folder*** and add the code below.

```
import { Directive, Input, TemplateRef, ViewContainerRef } from
'@angular/core';
@Directive({
    selector: '[appNot]'
})
export class AppNotDirective {
constructor(
    private templateRef: TemplateRef<any>,
    private viewContainer: ViewContainerRef) { }
    @Input() set appNot(condition: boolean) {
        if (!condition) {
            this.viewContainer.createEmbeddedView(this.templateRef);
        } else {
            this.viewContainer.clear();
        }
    }
}
```

Creating a Custom Structural Directive

add the ***appNot*** directive in the ***app.component.html*** but you can use it anywhere in the application.

The ****appNot directive*** is designed in a way that it appends the template element into the DOM if the ****appNot*** value is ***false*** just opposite the ****ngIf*** directive.

```
<h1 *appNot="true">True</h1>
<h1 *appNot="false">False</h1>
```

Pipes

Pipes in Angular are a feature that allow you to transform data in your application before displaying it to the user.

Angular comes with a set of **built-in pipes** that you can use in your templates.

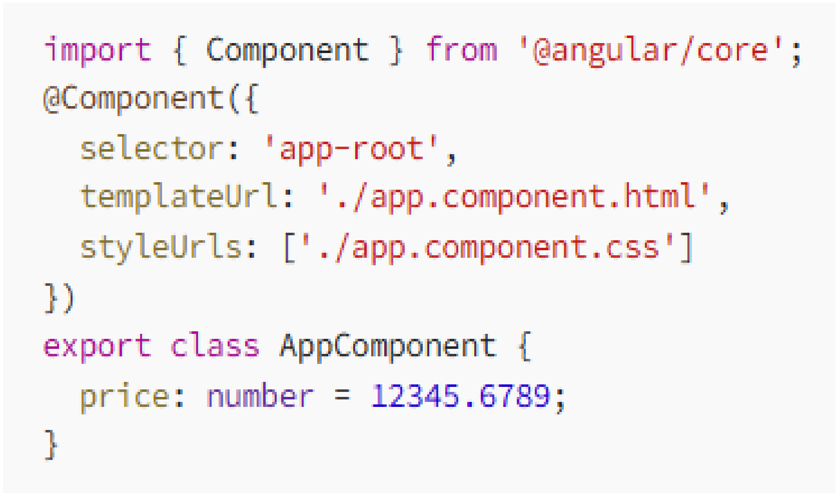
- Currency Pipe
- Date Pipe
- Json Pipe
- LowerCase Pipe
- UpperCase Pipe
- PercentPipe
- SlicePipe
- TitleCasePipe
- AsyncPipe

Currency Pipe

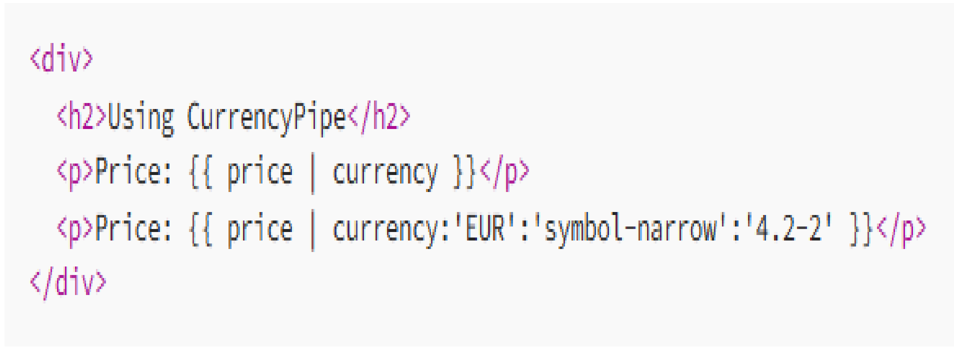
CurrencyPipe is a **built-in pipe** in Angular that is used to **format a number as a currency value.**

app.component.ts

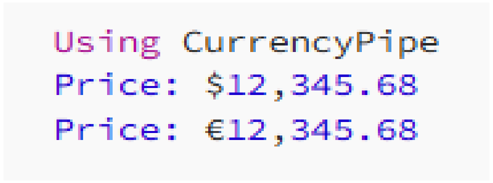
app.component.html



```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  price: number = 12345.6789;
}
```



```
<div>
  <h2>Using CurrencyPipe</h2>
  <p>Price: {{ price | currency }}</p>
  <p>Price: {{ price | currency:'EUR':'symbol-narrow':'4.2-2' }}</p>
</div>
```



```
Using CurrencyPipe
Price: $12,345.68
Price: €12,345.68
```

Date Pipe

DatePipe is a **built-in pipe** in Angular that is used to **format a date object.**

app.component.ts

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

app.component.html

```
<div>
  <h2>Using DatePipe</h2>
  <p>Current date: {{ currentDate | date }}</p>
  <p>Current date: {{ currentDate | date:'fullDate' }}</p>
  <p>Current date: {{ currentDate | date:'short' }}</p>
</div>
```

```
Using DatePipe
Current date: Mar 3, 2023
Current date: Friday, March 3, 2023
Current date: 3/3/23, 12:17 AM
```

Json Pipe

JsonPipe is a **built-in pipe** in Angular that is used to **transform an object into a JSON string.** It provides a way to display object values in a formatted JSON string

app.component.ts

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  myObject: any = {
    name: 'John',
    age: 30,
    email: 'john@example.com'
  };
}
```

app.component.html

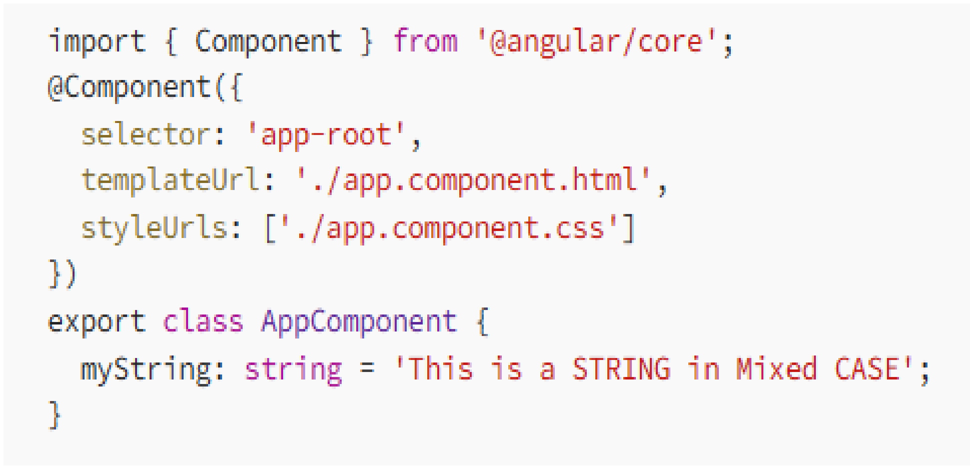
```
<div>
  <h2>Using JsonPipe</h2>
  <pre>{{ myObject | json }}</pre>
</div>
```

```
Using JsonPipe
{
  "name": "John",
  "age": 30,
  "email": "john@example.com"
}
```

LowerCase Pipe

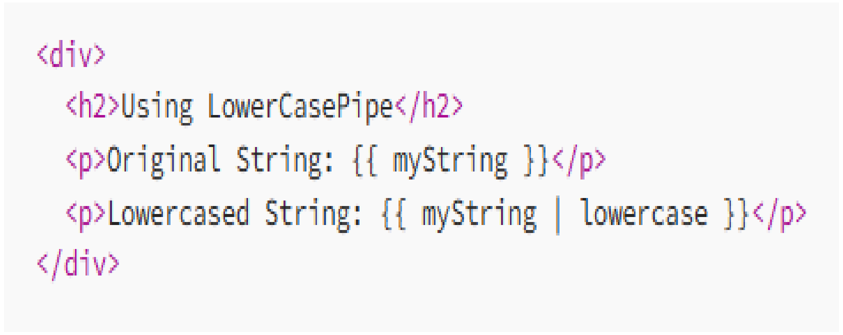
LowerCasePipe is a **built-in pipe** in Angular that is used to **transform a string into a lowercased string**.

app.component.ts

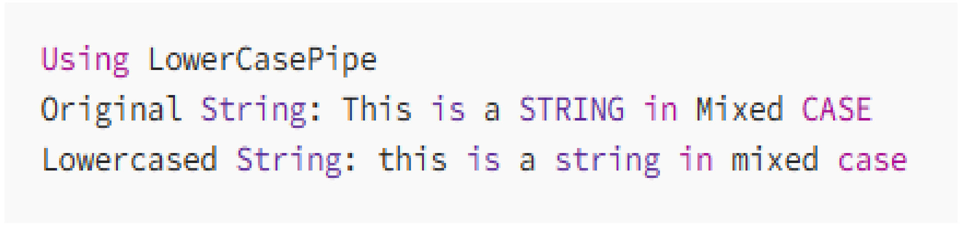


```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  myString: string = 'This is a STRING in Mixed CASE';
}
```

app.component.html



```
<div>
  <h2>Using LowerCasePipe</h2>
  <p>Original String: {{ myString }}</p>
  <p>Lowercased String: {{ myString | lowercase }}</p>
</div>
```

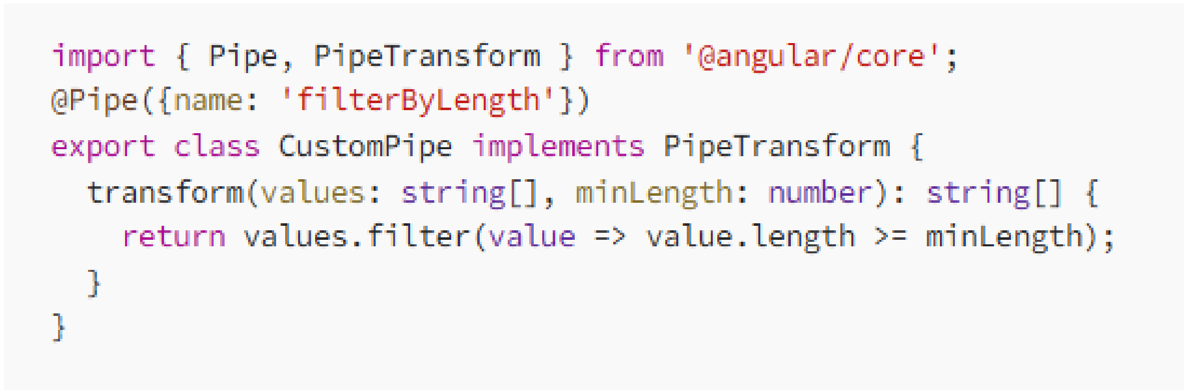


```
Using LowerCasePipe
Original String: This is a STRING in Mixed CASE
Lowercased String: this is a string in mixed case
```

Creating Custom Pipes

Custom pipes are used to **transform the data in an Angular application**

You can create custom pipes in Angular by defining a new class and implementing the ***PipeTransform*** interface. The ***PipeTransform*** interface contains a single method called ***transform*** that **takes an input value** and **returns the transformed value**.



```
import { Pipe, PipeTransform } from '@angular/core';
@Pipe({name: 'filterByLength'})
export class CustomPipe implements PipeTransform {
  transform(values: string[], minLength: number): string[] {
    return values.filter(value => value.length >= minLength);
  }
}
```

custom.pipe.ts

we define a **CustomPipe** class that **implements** the **PipeTransform interface**. The **transform()** method takes two arguments - **an array of strings** and a **minimum length**. It then filters out any strings in the array that **are greater than** or **equal to the specified length**. Decorate the class with the ***@Pipe decorator*** and provide a **name property** to give the pipe a name.

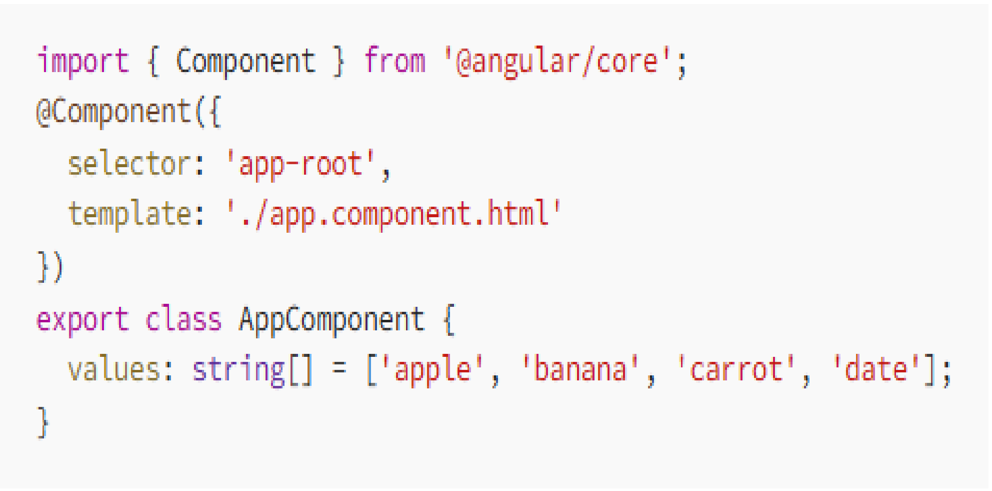
Creating Custom Pipes

app.module.ts

```
import { BrowserModule } from "@angular/platform-browser";
import { NgModule } from "@angular/core";

import { CustomPipe } from "./custom.pipe";

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

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

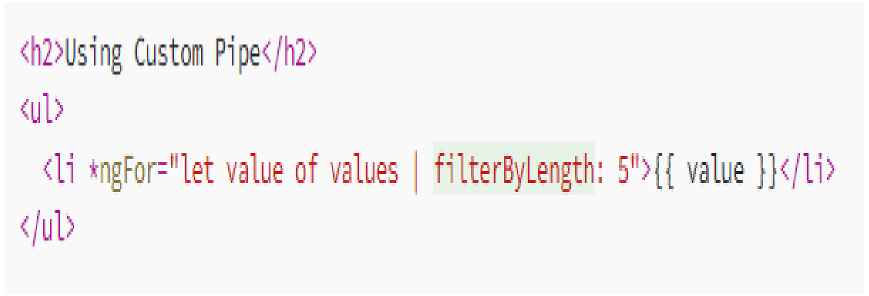
Creating Custom Pipes

app.component.ts

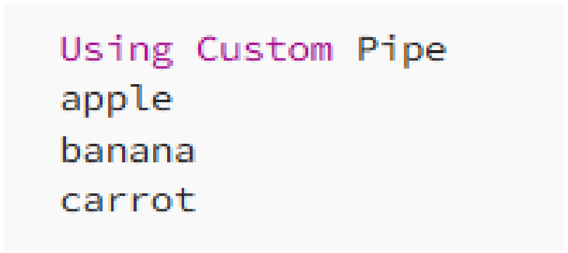


```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  template: './app.component.html'
})
export class AppComponent {
  values: string[] = ['apple', 'banana', 'carrot', 'date'];
}
```

app.component.html



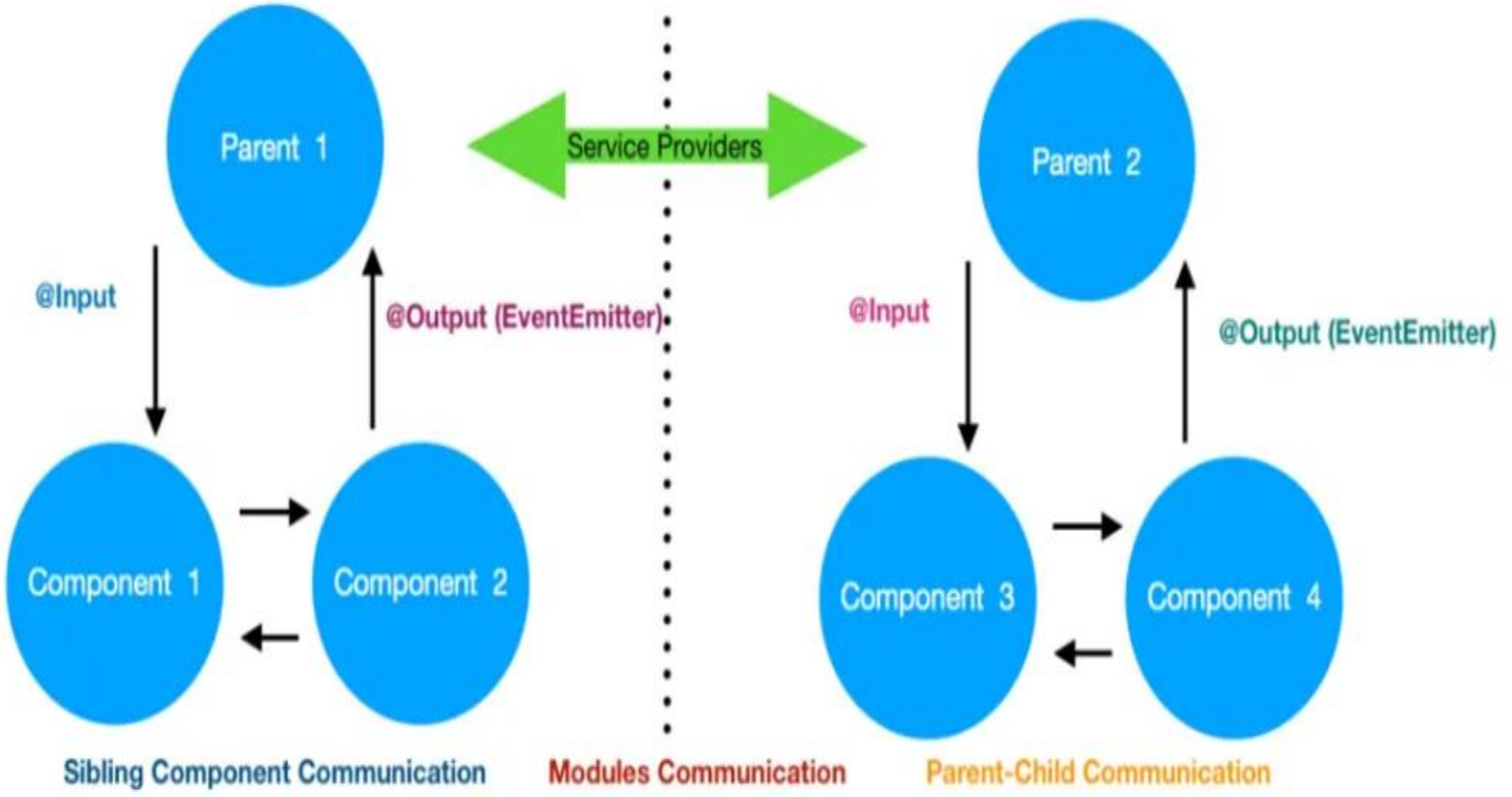
```
<h2>Using Custom Pipe</h2>
<ul>
  <li *ngFor="let value of values | filterByLength: 5">{{ value }}</li>
</ul>
```



```
Using Custom Pipe
apple
banana
carrot
```

The output shows the two strings from the values array that are longer than or equal to 5 characters.

Angular Components communication and sharing data



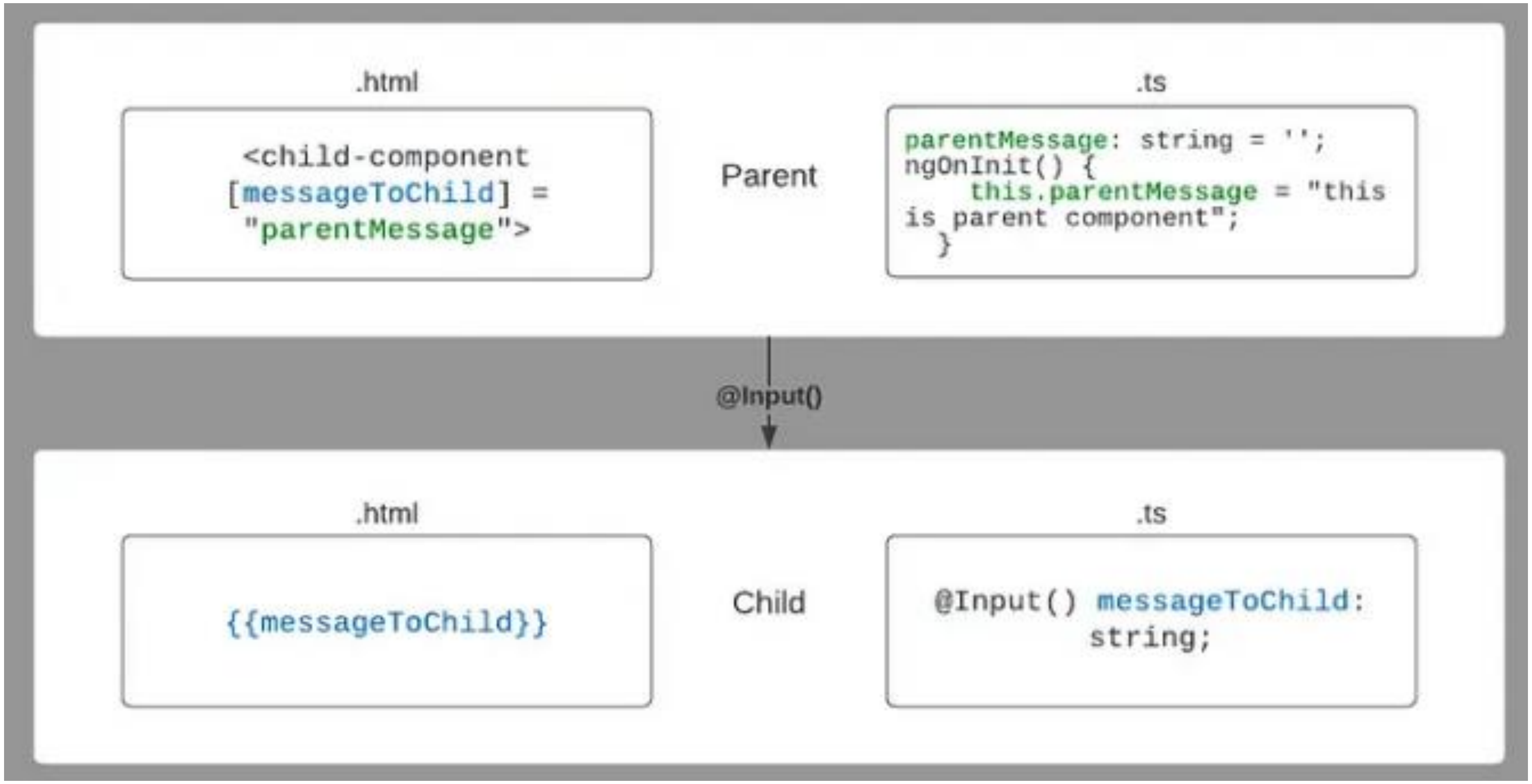
Angular Components communication and sharing data

There are few ways in which **components can communicate or share data between** them. And methods depend on **whether the components have a Parent-child relationship between them**.

1. Parent to Child Communication
2. Child to Parent Communication
3. Interaction when there is no parent-child relation.

Parent to Child Communication

Parent-to-child communication in Angular is the process of sending data or instructions from a **parent component** to a **child component**. This is typically done using **Input Properties and @Input() Decorator.**



```
<child-component
[messageToChild] =
"parentMessage">
```

```
parentMessage: string = '';
ngOnInit() {
      this.parentMessage = "this
is parent component";
  }
```

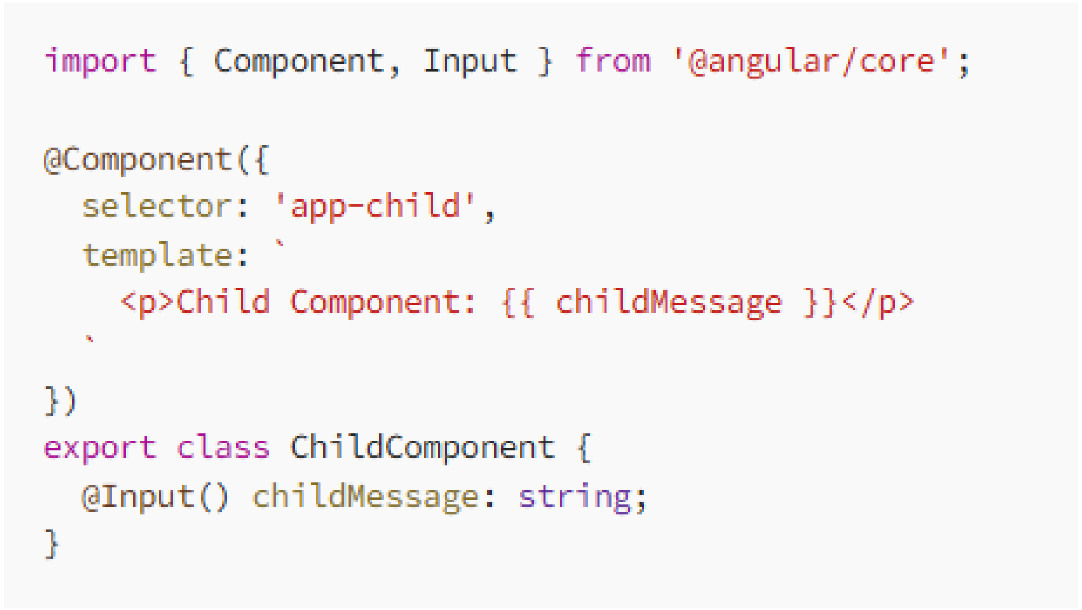
```
{{messageToChild}}
```

```
@Input() messageToChild:
string;
```

Parent to Child Communication

Child Component Definition

In the child component’s TypeScript file, define an input property using the **@Input()** decorator.



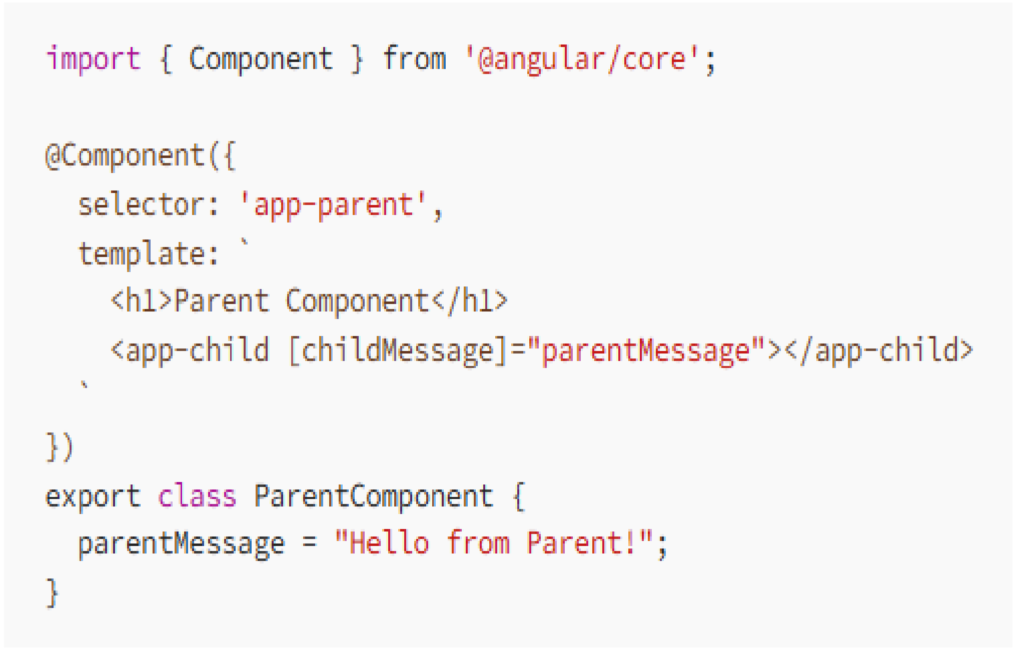
```
import { Component, Input } from '@angular/core';

@Component({
  selector: 'app-child',
  template: `
    <p>Child Component: {{ childMessage }}</p>
  `
})
export class ChildComponent {
  @Input() childMessage: string;
}
```

define an input property called **childMessage** of type string in the child component. The child component **will display the message passed from the parent.**

Parent to Child Communication

Parent Component Usage

In the parent component's template, you can use **the child component** and **bind the input property to a value in the parent component**.



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

@Component({
  selector: 'app-parent',
  template: `
    <h1>Parent Component</h1>
    <app-child [childMessage]="parentMessage"></app-child>
  `
})
export class ParentComponent {
  parentMessage = "Hello from Parent!";
}
```

we bind the **childMessage input property** in the **child component** to **parentMessage** property in the **parent component**. The child component will display the message passed from the parent.

Parent to Child Communication

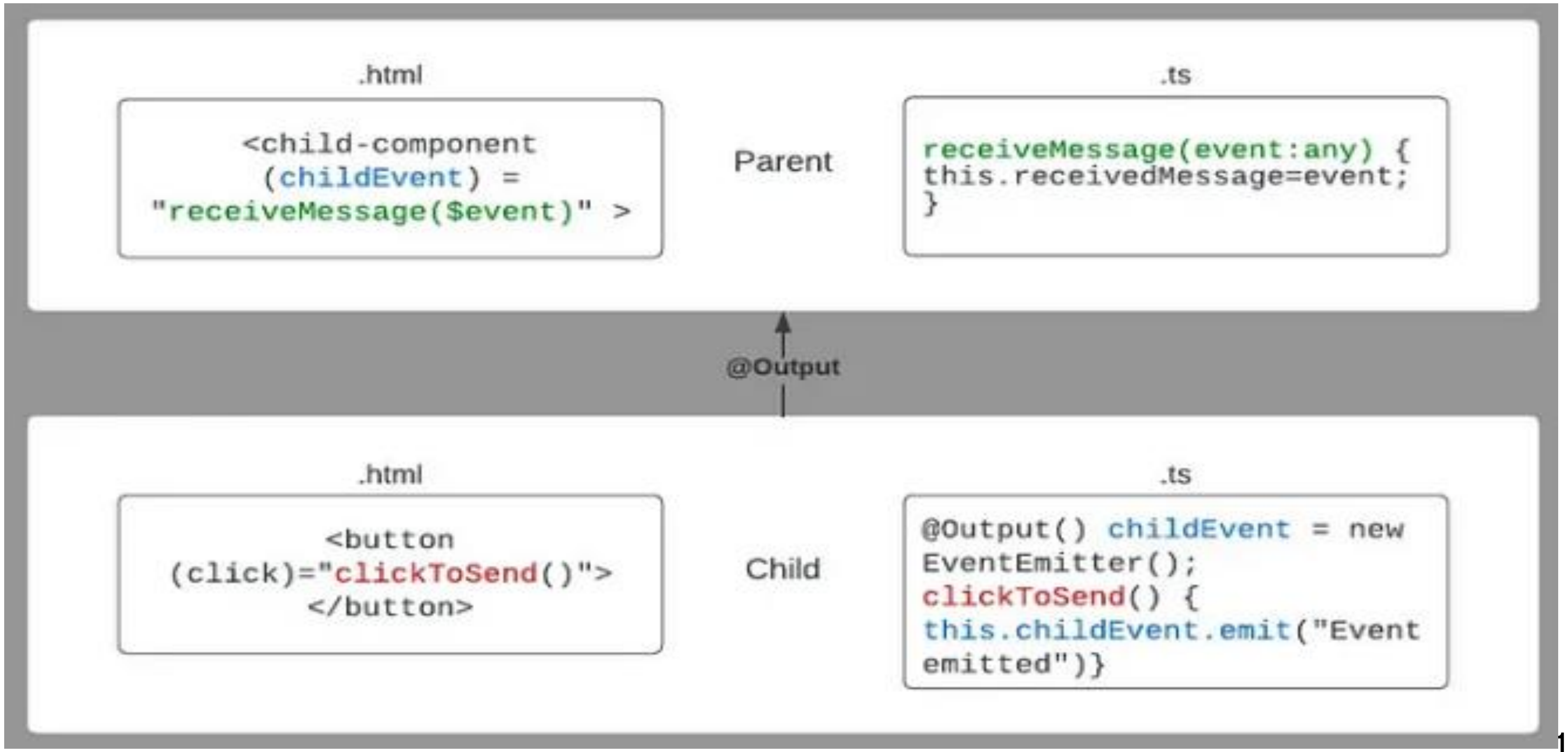
When to Use Parent-to-Child Communication

Parent-to-child communication is commonly used when you want to:

- •Pass data from a parent component to a child component.
- •Configure a child component’s behavior based on the parent’s input.

Child to Parent Communication

Child-to-parent communication in Angular is the process of sending data or events from a child component to a parent component. This is achieved using **Output Properties** and **@Output() Decorator** in conjunction with the **EventEmitter class**



```
<child-component
(childEvent) =
"receiveMessage($event)" >
```

```
receiveMessage(event:any) {
this.receivedMessage=event;
}
```

```
<button
(click)="clickToSend()">
</button>
```

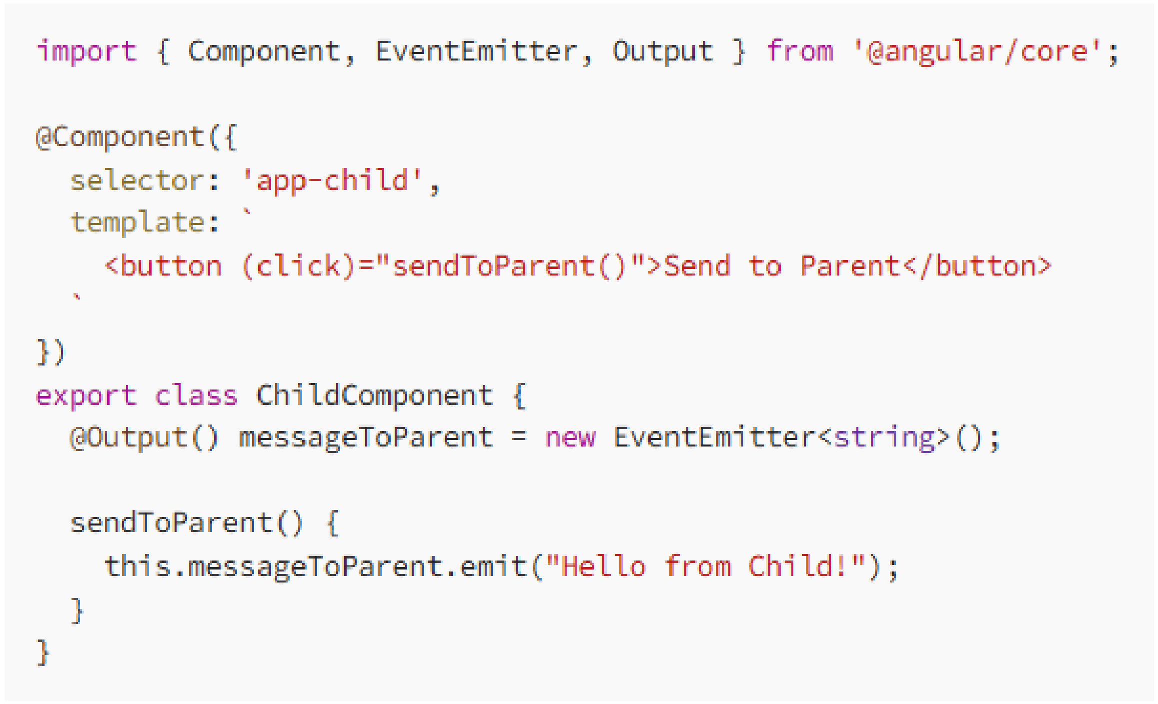
```
@Output() childEvent = new
EventEmitter();
clickToSend() {
this.childEvent.emit("Event
emitted")}
```

Child to Parent Communication

To enable child-to-parent communication, you define an **output property in the child component** using the **@Output() decorator** and the **EventEmitter** class.

Child Component Definition

In the child component’s TypeScript file, define an output property using the **@Output() decorator** and **EventEmitter**



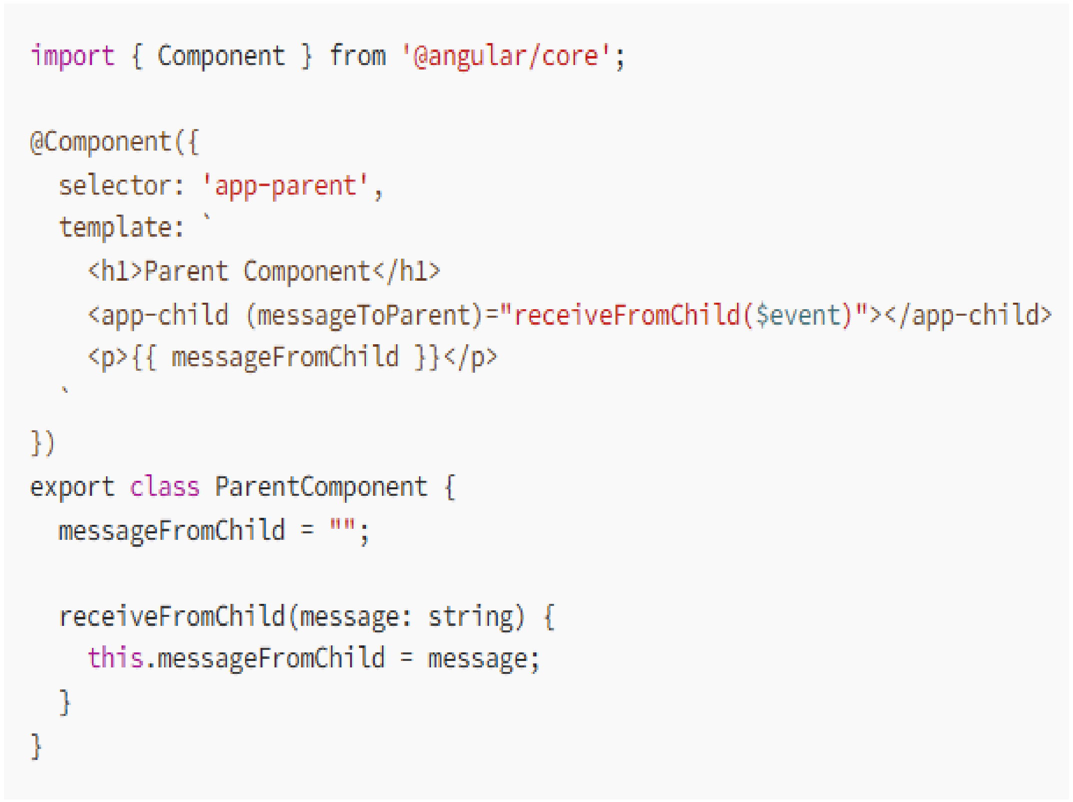
```
import { Component, EventEmitter, Output } from '@angular/core';

@Component({
  selector: 'app-child',
  template: `
    <button (click)="sendToParent()">Send to Parent</button>
  `
})
export class ChildComponent {
  @Output() messageToParent = new EventEmitter<string>();

  sendToParent() {
    this.messageToParent.emit("Hello from Child!");
  }
}
```

we define an **output property** named **messageToParent** and use the **EventEmitter** to **emit events** when a button is clicked.

Child to Parent Communication

Parent Component Usage

In the **parent component’s template**, you can **bind to the child component’s output property** and **listen for events**.



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

@Component({
  selector: 'app-parent',
  template: `
    <h1>Parent Component</h1>
    <app-child (messageToParent)="receiveFromChild($event)"></app-child>
    <p>{{ messageFromChild }}</p>
  `
})
export class ParentComponent {
  messageFromChild = "";

  receiveFromChild(message: string) {
    this.messageFromChild = message;
  }
}
```

we bind to **messageToParent output property** in the **child component** and **listen for events to update** the **messageFromChild** property in the **parent component**.

Child to Parent Communication

When to Use Child-to-Parent Communication

Child-to-parent communication is often used when you want to:

- Trigger actions in the parent component based on events in the child component.
- Pass data from a child component to a parent component in response to user interactions.

Communication when there is no relation

If the **Components do not share the Parent-child relationship**, then the only way they can share data is by using the **services** and **observable**.

The advantageous of using service is that

1. You can **share data between multiple components**.
2. Using **observable**, you can **notify each component, when the data changes**.

Create **the Service** and create the **Angular Observable in that service** using either **BehaviorSubject** or **Subject**.

Communication when there is no relation

```
export class TodoService {
private _todo = new BehaviorSubject<Todo[]>([]);
readonly todos$ = this._todo.asObservable();
…
}
```

```
this._todo.next(Object.assign([], this.todos));
```

The **_todo observable** will **emit data**, whenever it is available or changes using the **next method of the Subject**.

```
this.todoService.todos$.subscribe(val=> {
this.data=val;
//do whatever you want to with it.
})
```

In the component class, you **can listen to the changes** just by **subscribing to the observable**.