Feature of angular

03 September 2025 01:04

- Angular is a powerful web development framework created by Google that helps build fast, and scalable single-page applications (SPAs).
- Angular creates single page application(SPA).
  A Single-Page Application (SPA) is a web application that loads a single HTML page and then dynamically updates the content within that same page as the user navigates, instead of loading entirely new pages from the server.

Angular actually uses a different architecture called **Model-View-ViewModel (MVVM)**, which is an evolution of MVC.
Here is a breakdown of how the components you mentioned fit into this pattern:

- **View:** The **HTML files** are the **View** because they are responsible for what the user sees and interacts with.
- **Controller:** The **.component.ts** file does act like a controller by managing the view's behavior and user input, but it's more accurately called the **ViewModel**. It handles the logic for the view and exposes data and methods for the view to use.
- **Model:** The **service files** are the **Model**. They handle the application's data and business logic, like communicating with a server or manipulating data. Components then inject these services to access the data.

Why angular is used:

- **Two-way data binding** simplifies synchronization between the model and view.
- **Component-based architecture** promotes reusability and maintainability.
- **Dependency injection** makes components easy to manage and test.
- **TypeScript** provides static typing, reducing runtime errors.
- **CLI** streamlines project creation and development tasks.

Important files

15 September 2025 14:23

app.component.ts

This is the main component of your application. It contains the logic and configuration for the view.

- **@Component**: This is a **decorator** that marks the class as an Angular component. It provides metadata about the component.
- **selector**: The **identifier** for your component. It's the custom HTML tag you use to display this component in your templates (e.g., <app-root></app-root>).
- **templateUrl or template**: Links the component's **HTML template**. templateUrl points to a separate file, while template allows you to write the HTML inline.
- **styleUrls or styles**: Links or includes the component's **CSS styles**. The styles defined here are scoped only to this component.
- **export class**: The TypeScript class that holds the component's **logic**, properties, and methods.

app.module.ts

This is the main module that organizes your application.

- **@NgModule**: This **decorator** marks the class as an Angular module. It tells Angular how to compile and run your application.
- **declarations**: An array for listing all **components, directives, and pipes** that belong to this module.
- **imports**: An array for listing all **other modules** whose exported classes are needed by the components in this module. .
- **providers**: An array for registering **services**. This makes them available for dependency injection throughout the module.
- **bootstrap**: This property identifies the **root component** that Angular should load when the application starts. For the main module, this is usually AppComponent.

Other Important Files

- **main.ts**: The main **entry point** of your application. It bootstraps the AppModule to start the application.
- **index.html**: The single **host page** for your SPA. The <app-root> tag is placed here, and Angular loads all components into it.
- **styles.css**: The file for **global styles** that apply to your entire application.
- **angular.json**: The workspace and project configuration file. It holds build options, file locations, and other settings.
- **package.json**: Lists all **project dependencies** and scripts.
- **app-routing.module.ts**: A dedicated module for configuring your application's **routes** (the paths and components for navigation).

Commands

19 September 2025 01:31

Project & Server Commands

npx ng new my-app	Creates a new Angular project with the name my-app.	Project Start
npx ng serve --open	Builds and serves the application, allowing you to view it in a browser at localhost:4200.	Project Start
ng test	Runs the unit tests for your application using the Karma test runner.	Unit Testing
ng build	Compiles and bundles your application for deployment to a production server.	Build & Deployment
npm install @ngrx/store	Installs the core NgRx state management library to your project.	NgRx
ng add @ngrx/store	Installs NgRx and automatically configures the boilerplate code for you.	NgRx

Generation Commands

These are used to generate the building blocks of an Angular application.

Angular CLI Shorthand Commands

Shorthand Command	Purpose	Section
npx ng new my-app	Creates a new Angular project.	Project Start
ng serve --open or ng s	Builds and serves the application.	Project Start
ng test or ng t	Runs the unit tests for your application.	Unit Testing
ng build or ng b	Compiles your application for deployment.	Build & Deployment
npx ng g c my-component	Creates a new component. (g for generate, c for component)	Components
npx ng g s my-service	Creates a new service. (s for service)	Shared Services
npx ng g m my-module	Creates a new NgModule. (m for module)	NgModules, Lazy Loading
npx ng g m my-module --route my-route	Creates a module and sets up a route for lazy loading.	Lazy Loading
npx ng g g my-guard	Creates a route guard. (g for guard)	Route Guards
npx ng g i my-interceptor	Creates an HTTP interceptor. (i for interceptor)	Interceptors
Npx ng g p my-pipe	Creates a new pipe	
npx json-server --watch db.json	Used to run json server	

Life cycle hooks

24 September 2025 18:08

Life Cycle Hooks are specialized, built-in methods provided by a framework that allow us to **intervene at predictable moments** in a component's existence—specifically during its creation, update, and destruction phases. Their primary function is to give us reliable points to execute **critical setup logic** (like fetching data when a component loads) or **necessary cleanup code** (like closing connections) to maintain application stability.

1. Birth and Initialization

These are the first hooks that get called when your component comes to life.

- **ngOnChanges**: This is the very first hook. It's called when your component receives its initial **input data** from a parent component. It will also fire again every time that input data changes.
- **ngOnInit**: This is the most important hook. It's called **only once**, after all the initial setup is complete. This is the best place for your main logic, like fetching data from a server.

2. Growing and Updating

These hooks are all about change detection. They are called frequently as the component's data or view is checked and updated.

- **ngDoCheck**: This is a powerful but dangerous hook. It's called during every single change detection cycle. You can use it for your own custom change checks, but can easily reduce the performance of application if not used properly.

3. The Final Goodbye

This is the cleanup hook, and it's essential for preventing problems.

- **ngOnDestroy**: Called just before the component is completely removed from the page. This is your last chance to **clean up** everything. You must use this hook to unsubscribe from data streams and timers to prevent **memory leaks**.

File Structure

03 September 2025 01:18

Root Folder

- **node_modules/**: Stores all the project's dependencies from npm.
- **src/**: This is the main directory where all the application's source code resides.
- **.angular-cli.json**: Configuration file for the Angular CLI.
- **package.json**: Lists project dependencies and scripts.
- **tsconfig.json**: TypeScript compiler configuration.
- **angular.json**: Configuration file for the Angular workspace and individual projects.

src/ Folder

- **app/**: Contains all the application's components, modules, and services. This is where most of your work will be.
    - **app.component.ts**: The root component's TypeScript logic file.
    - **app.component.html**: The HTML template for the root component.
    - **app.component.css**: The component-specific CSS styles for the root component.
    - **app.component.spec.ts**: The unit test file for the root component.
    - **app.module.ts**: The root module that ties all the components together.
- **public/**: Stores static assets like images and fonts.
- **index.html**: The main HTML file that serves as the entry point for the application.
- **main.ts**: The main entry point of the application, which bootstraps the AppModule.
- **styles.css**: Global styles for the application.

One way(Interpolation) and two way data binding

03 September 2025 22:15

One-Way Data Binding (Interpolation)

One-way data binding is a method for displaying data from a **component**'s TypeScript class in its **HTML template**. The data flow is unidirectional, meaning it goes from the component to the view. The view is updated when the component's data changes, but not the other way around. Interpolation {{ }} is the most common form of one-way data binding.

Key points:

- **Syntax:** Use double curly braces {{ }}.
- **Purpose:** To display a component property's value directly in the template.
- **Data Flow: Component** --> **View**.

Example Code:

app.component.ts

TypeScript

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html'
})
export class AppComponent {
  // This property will be displayed in the view.
  productName = 'Angular T-Shirt';
  productPrice = 25;
}
```

app.component.html

HTML

```html
<div>
  <h1>Product: {{ productName }}</h1>
  <p>Price: ${{ productPrice }}</p>
</div>
```

Two-Way Data Binding

Two-way data binding creates a synchronized data flow between the component's **TypeScript class** and its **HTML template**. The flow of data is bidirectional. It ensures that any changes to the data in the component are instantly reflected in the view, and any changes in the view (from user input) are instantly reflected back in the component.
This is a two-part process that combines **property binding** (data flows from component to view) and **event binding** (data flows from view to component), which is conveniently

handled by the [(ngModel)] syntax.

Key points:

- **Syntax:** Use [(ngModel)]. This is often called the "banana in a box" syntax.
- **Purpose:** To keep the view and the component property synchronized. It is commonly used with form elements like <input>, <textarea>, and <select>.
- **Data Flow: Bidirectional Component** <--> **View**.
- **Import:** Requires the **FormsModule** to be imported in your app.module.ts file to use [(ngModel)].

Example Code:

app.module.ts

TypeScript

```typescript
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule } from '@angular/forms'; // 1. Import FormsModule
import { AppComponent } from './app.component';

@NgModule({
  declarations: [AppComponent],
  imports: [
    BrowserModule,
    FormsModule // 2. Add FormsModule to imports array
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

app.component.ts

TypeScript

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html'
})
export class AppComponent {
  // This property will be bound to the input field.
  // The initial value is 'Hello World'.
  userName = 'Hello World';
}
```

app.component.html

HTML

```
<div>
  <h1>Your Name: {{ userName }}</h1>

  <input type="text" [(ngModel)]="userName">
</div>
```

Property Binding

03 September 2025 22:41

What is Property Binding?

Property binding is a way to set the value of an **HTML property** using data from components typescript class. It is a type of **one-way data binding**, meaning data flows only from your component to the view (the HTML).

What it does	Sets a property of an HTML element ([src], [disabled], [alt], [placeholder]).
Syntax	Use square brackets [ ] around the property name. For example, [src].
Data Flow	**Component --> View**

Simple Example

Let's use a simple example of a button. We want the button to be clickable or not, based on a variable in our code.

1. The Component (`app.component.ts`)

This is where our logic and data live.

TypeScript

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html'
})
export class AppComponent {
  // A variable that will control if the button is enabled or disabled.
  // It starts as true, so the button will be disabled at first.
  isButtonDisabled = true;
}
```

2. The HTML (`app.component.html`)

This is where we use property binding to connect the HTML to our component's data.

HTML

```html
<div>
  <button [disabled]="isButtonDisabled">
    I am a button!
  </button>
</div>
```

The Flow of Data

This is how the property binding works in this example:

1. **Initial Load**: Angular reads the app.component.ts file and sees that isButtonDisabled is set to true.
2. **Binding**: Angular then looks at the app.component.html file and sees the [disabled] property binding.
3. **Connection**: It connects the button's disabled property to the isButtonDisabled variable.
4. **View Update**: Because isButtonDisabled is true, Angular sets the HTML disabled property to true. This makes the button unclickable and grayed out.

Event Binding

03 September 2025 23:34

What is Event Binding?

Event binding is a way to listen for events that happen in the HTML and run code in your Angular component when they occur. It's how you handle user actions like a button click, a key press, or a form submission. It is a type of **one-way data binding**, where data flows from the view (the HTML) to the component.

What it does	Runs a method in your component when an event happens ((click), (submit), (keypress)).
Syntax	Use parentheses () around the event name. For example, (click).
Data Flow	**View** --> **Component**

Simple Example

Let's use a simple example of a button. We want to show an alert message when the user clicks it.

1. The Component (`app.component.ts`)

This is where our logic and data live.

TypeScript

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html'
})
export class AppComponent {
  // A variable that we will change when the button is clicked.
  message = '';

  // A method that will be triggered by the button click event.
  // It changes the value of the 'message' variable.
  showAlert(): void {
    this.message = 'The button was clicked!';
  }
}
```

2. The HTML (`app.component.html`)

This is where we use event binding to listen for the click.

HTML

```
<div>
  <button (click)="showAlert()">
    Click Me
  </button>

  <p>{{ message }}</p>
</div>
```

The Flow of Data

This is how event binding works in this example:

1. **Initial Load**: The app loads, and the message variable in the component is an empty string. The <p> tag is empty.
2. **User Action**: You click the button labeled "Click Me."
3. **Event Trigger**: The (click) event binding on the button notices the click.
4. **Method Call**: Angular immediately calls the showAlert() method in the AppComponent.
5. **Component Logic**: The showAlert() method runs and changes the value of the message variable to 'The button was clicked!'.
6. **View Update**: Angular detects the change in the message variable. The interpolation {{ message }} automatically updates the <p> tag, and the text "The button was clicked!" appears on the screen.

Parent to Child Communication

03 September 2025 23:41

Parent to Child Communication

Parent-to-child communication is a **one-way data flow** where a parent component sends data to a child component.
This is used when a child component needs information from its parent to display or work with.
It's achieved by decorating a property in the child component with **@Input()** and then using **property binding** in the parent's template to pass the data.
The communication happens between parent html and child ts file.
Flow:

1. In the parent.component.ts, you have a variable with the data you want to send. Let's call it parentMessage.
2. In the child.component.ts, you create a variable to receive the data and decorate it with @Input(). Let's call it messageFromParent.
3. In the parent.component.html, you use property binding [ ] on the child component's selector to map the parent's variable to the child's @Input() variable: <app-child [messageFromParent]="parentMessage"></app-child>.
4. The data flows from the parentMessage variable down to the messageFromParent variable.

Example:

parent.component.ts

TypeScript

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

@Component({
  selector: 'app-parent',
  templateUrl: './parent.component.html'
})
export class ParentComponent {
  // 1. This is the variable in the parent component that holds the data.
  parentMessage: string = 'Hello from the parent!';
}
```

parent.component.html

HTML

```html
<div>
  <h2>Parent Component</h2>
  <app-child [messageFromParent]="parentMessage"></app-child>
</div>
```

child.component.ts

TypeScript

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

@Component({
  selector: 'app-child',
  template: `
    <h3>Child Component</h3>
    <p>Received message: {{ messageFromParent }}</p>
  `
})
export class ChildComponent {
  // 2. We use @Input() to create a property that can receive data from the parent
  // The parent component will send data to this property.
  @Input() messageFromParent: string = '';
}
```

// This is property binding, childs property Is binded with parents message

Child to Parent communication

12 September 2025 02:21

Child to Parent Communication

Child-to-parent communication is a way for a child component to send a message or data back up to its parent. It's used when a user action in the child needs to affect the parent. This is achieved by using the **@Output()** decorator and an **EventEmitter** in the child component. @Output() decorator creates a custom event. The child uses the **emit()** method to trigger the custom event. The parent then uses **event binding** in its template to listen for that event and react to it.

Data Flow and Example

Flow:

1. In the child.component.ts, you create an EventEmitter and decorate it with @Output(). Let's call it messageEvent.
2. A user action in the child.component.html (e.g., a button click) triggers a method in the child.
3. This method uses this.messageEvent.emit('your message here') to send the data.
4. In the parent.component.html, you use event binding ( ) to listen for the child's event: <app-child (messageEvent)="receiveMessage($event)"></app-child>.
5. The receiveMessage() method in the parent.component.ts is called, and the data sent by the child is passed as a parameter (represented by $event).

Example:

child.component.ts

TypeScript

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

@Component({
  selector: 'app-child',
  template: `
    <h3>Child Component</h3>
    <button (click)="sendMessage()">Send Message to Parent</button>
  `
})
export class ChildComponent {
  // 1. We use @Output and EventEmitter to create a custom event that can send data.
  // We're saying this event will emit a string.
  @Output() messageEvent = new EventEmitter<string>();

  // 3. This method is called by the button click. It emits the event with a message
  sendMessage(): void {
    this.messageEvent.emit('Hello from the child!');
  }
}
```

parent.component.html

HTML

```html
<div>
  <h2>Parent Component</h2>
  <app-child (messageEvent)="receiveMessage($event)"></app-child>

  <p>Message from child: {{ messageFromChild }}</p>
</div>
```

parent.component.ts

TypeScript

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

@Component({
  selector: 'app-parent',
  templateUrl: './parent.component.html'
})
export class ParentComponent {
  // 5. This variable will hold the message received from the child.
  messageFromChild = '';

  // 6. This method is called by the event binding in the parent's template.
  // The data emitted by the child is passed as the $event parameter.
  receiveMessage($event: string): void {
    this.messageFromChild = $event;
  }
}
```

Structural Directives

12 September 2025 11:27

A **Directive** is a specialized class that attaches to components to **programmatically modify their behavior, appearance, or structure**. They act as instructions, allowing developers to extend the capabilities of standard HTML.

What are Structural Directives?

Structural directives are a special type of Angular directive that changes the **structure** of the HTML layout (the DOM). They do this by adding, removing, or manipulating elements and their children. The key characteristic of a structural directive is the asterisk * prefix, which tells Angular to alter the HTML's structure.

The three main built-in structural directives in Angular are:

1. ***ngIf**: Used for conditional rendering.
2. ***ngFor**: Used to repeat elements for each item in a list.
3. ***ngSwitch**: Used to conditionally display one of many elements.

*ngIf

12 September 2025 11:33

What is *ngIf?

*ngIf is a **structural directive** that adds or removes an element from the page's structure based on a **boolean condition**.
If the condition is **true**, the element is added to the DOM. If it's **false**, the element is completely removed. This is used for **conditional rendering**, such as showing an error message only when an error exists.
The syntax is *ngIf="condition".

Data Flow and Example
Flow:

1. The AppComponent has a boolean variable, let's call it isUserLoggedIn.
2. In the app.component.html, the user's browser loads the HTML.
3. Angular sees the <p> element with the *ngIf="isUserLoggedIn" directive.
4. Angular checks the value of the isUserLoggedIn variable in the app.component.ts.
5. If isUserLoggedIn is false, Angular **removes** the entire <p> element from the DOM.
6. If the value of isUserLoggedIn later changes to true (e.g., after the user logs in), Angular will **add** the <p> element back into the DOM, and it becomes visible on the page.

Example:

app.component.ts

TypeScript

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html'
})
export class AppComponent {
  // A variable that will be our condition for *ngIf.
  // We'll set it to false initially.
  isUserLoggedIn: boolean = false;

  // This method will be triggered by a button click to change the condition.
  loginLogout(): void {
    // We are toggling the value of isUserLoggedIn to simulate logging in or out.
    this.isUserLoggedIn = !this.isUserLoggedIn;
  }
}
```

app.component.html

HTML

```
<div>
  <button (click)="loginLogout()">Toggle Login Status</button>

  <p *ngIf="isUserLoggedIn">Welcome back, you are now logged in!</p>
</div>
```

*ngFor

12 September 2025 11:33

What is *ngFor?

*ngFor is a **structural directive** that repeats an HTML element for each item in a list or array. It's used to dynamically create a list of elements based on a collection of data from your component. The asterisk * tells Angular to change the structure of the page.

What it does	Repeats an HTML element and its content for every item in an array or list.
Syntax	*ngFor="let item of items"
Why it's used	To display lists of data, such as a list of products, a list of users, or items in a to-do list. It saves you from manually writing HTML for every item.
Data Flow	The flow is one-way: Angular reads the array from your component and generates the HTML view based on its contents.

Simple Example

Let's use an example to display a list of fruits from an array.

1. The Component (`app.component.ts`)

This is where our data, the list of fruits, is stored.

TypeScript

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html'
})
export class AppComponent {
  // An array of strings that *ngFor will iterate over.
  fruits: string[] = ['Apple', 'Banana', 'Cherry', 'Mango'];
}
```

2. The HTML (`app.component.html`)

This is where we use the `*ngFor` directive to create the list.

HTML

```
<div>
  <h3>My Favorite Fruits</h3>

  <ul>
    <li *ngFor="let fruit of fruits">
      {{ fruit }}
    </li>
  </ul>
</div>
```

*ngSwitch

12 September 2025 19:18

*ngSwitch is a **structural directive** that works like a switch statement, conditionally displaying one element from a group based on expression's value.
It's used to show different parts of a template based on a variable's value, such as displaying a "home" or "about" page view.
You use a container with [ngSwitch]="expression", and then use *ngSwitchCase="value" on the elements inside to define which one to show.
ngSwitchDefault is also there to show some default html if none of the cases matches.

Data Flow and Example

Flow:

1. In the app.component.ts, we have a variable called currentPage with a string value.
2. In the app.component.html, Angular sees the [ngSwitch] directive on the container. It binds the currentPage variable to this directive.
3. Angular then looks at the *ngSwitchCase directives on the elements inside the container.
4. It checks the value of currentPage and compares it to the value of each *ngSwitchCase.
5. The element whose *ngSwitchCase value **matches** the currentPage variable is **added** to the DOM and displayed. All other elements are **removed**.
6. If currentPage doesn't match any *ngSwitchCase, the element with *ngSwitchDefault is added to the DOM.

Example:

App.component.ts

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
})
export class AppComponent {
  currentPage: string = 'home';

  // This method handles changing the page, keeping the logic in the component.
  goToPage(page: string): void {
    this.currentPage = page;
  }
}
```

App.component.html

```
<div>
  <button (click)="goToPage('home')">Home</button>
  <button (click)="goToPage('about')">About</button>
  <button (click)="goToPage('contact')">Contact</button>

  <hr>

  <div [ngSwitch]="currentPage">
    <p *ngSwitchCase="'home'">Welcome to the Home Page!</p>
    <p *ngSwitchCase="'about'">Learn more about us here.</p>
    <p *ngSwitchCase="'contact'">Contact us for more information.</p>
    <p *ngSwitchDefault>Select a page from the buttons above.</p>
  </div>
</div>
```

Attribute Directives

12 September 2025 19:28

What are Attribute Directives?

Attribute directives are a type of Angular directive that changes the **appearance** of a component or HTML element. Unlike structural directives (*ngIf, *ngFor) which add or remove elements, attribute directives simply modify an existing one. You apply them to an element as if they were a standard HTML attribute.
The two main built-in attribute directives are **ngStyle** and **ngClass**.

ngStyle

12 September 2025 19:32

What is ngStyle?

ngStyle is an **attribute directive** used to set multiple inline CSS styles on an HTML element dynamically.
You use it by binding an object to the [ngStyle] directive, where the object's keys are CSS properties and its values are expressions that determine the style. This is used for dynamic styling, such as changing a text's color based on a condition.
The syntax is [ngStyle]="{'style-property': expression}".

Data Flow and Example

Flow:

1. In the app.component.ts, we have a variable, let's say isUrgent, which is a boolean.
2. In the app.component.html, Angular sees the [ngStyle] directive on the <p> element. It will evaluate the object you've passed to it.
3. The directive checks the value of isUrgent from the app.component.ts file.
4. The value of the 'color' property in the object is set based on the isUrgent variable. If isUrgent is true, the color will be 'red'. If isUrgent is false, it will be 'black'.
5. Angular applies the resulting style directly to the <p> element. This is all handled in the background, updating the view whenever isUrgent changes.

Example:

app.component.ts

TypeScript

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html'
})
export class AppComponent {
  // A variable that will be our condition for the style.
  isUrgent: boolean = true;

  // A method to toggle the urgency.
  toggleUrgency(): void {
    this.isUrgent = !this.isUrgent;
  }
}
```

app.component.html

HTML

```
<div>
  <button (click)="toggleUrgency()">Toggle Urgency</button>

  <hr>

  <p [ngStyle]="{ 'color': isUrgent ? 'red' : 'black' }">
    This message's color will change.
  </p>
</div>
```

We are using property binding.
The square brackets [ ] around ngStyle indicate that it's a **property binding**. In this case, we're binding the ngStyle directive's input property to the JavaScript object { 'color': isUrgent ? 'red' : 'black' }. The ngStyle directive then takes that object and applies the styles to the HTML element.

ngClass

12 September 2025 19:34

What is ngClass?

ngClass is an **attribute directive** that dynamically adds or removes CSS classes from an HTML element based on a boolean condition.
It is used to change an element's appearance based on its state, such as highlighting an active menu item.
You use it by binding an object to the [ngClass] directive. The object's keys are the class names, and their values are the conditions (true/false) that determine if the class is applied.
It uses **property binding**.
The syntax is [ngClass]="{'class-name': condition}".

Data Flow and Example

Flow:

1. In the app.component.ts, you have a boolean variable, let's say isOnline. You also have a CSS file (app.component.css) with styles for a class like .status-online.
2. In the app.component.html, Angular sees the [ngClass] directive on the <p> element. It evaluates the object { 'status-online': isOnline }.
3. The directive checks the value of isOnline from the app.component.ts.
4. If isOnline is true, Angular adds the CSS class status-online to the <p> element.
5. If isOnline is false, Angular removes the status-online class from the <p> element.
6. The styling defined in your CSS file for the .status-online class will be applied or removed accordingly.

Example:

app.component.ts

TypeScript

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  // A variable that will control whether the 'status-online' class is applied.
  isOnline: boolean = true;

  // A method to toggle the status.
  toggleStatus(): void {
    this.isOnline = !this.isOnline;
  }
}
```

app.component.css

CSS

```css
/* This class will be added or removed by ngClass. */
.status-online {
  color: green;
  font-weight: bold;
}
```

app.component.html

HTML

```html
<div>
  <button (click)="toggleStatus()">Toggle Status</button>

  <hr>

  <p [ngClass]="{ 'status-online': isOnline }">
    You are {{ isOnline ? 'online' : 'offline' }}.
  </p>
</div>
```

ngClass or ngStyle

12 September 2025 19:54

Both ngClass and ngStyle change an element's style dynamically, but they do it in different ways.
Use **ngClass** when you want to apply a complete set of **pre-defined styles** from a CSS class. Think of it as putting on a new outfit. It's best for toggling complex styles, as you keep your styling logic in a separate CSS file, making it more organized and reusable.
Use **ngStyle** for **single, dynamic style properties** that are calculated or change frequently. Think of it as applying a specific color or a number. It's best when the style value isn't something you can easily put into a static class, like a progress bar's percentage or a font size based on a user's input.
In short, **ngClass for class-based styles**, and **ngStyle for inline, one-off styles**.

Custom Directives

13 September 2025 18:59

A **custom directive** is a reusable class that you create to add new behavior or appearance to any HTML element.
It's used to avoid repeating code. If you have a specific style or behavior (like highlighting text) that you want to use in many places, you can create a single directive for it instead of writing the code over and over again. Data can be passed into the directive using @Input().

1. Custom Attribute Directive Example

Example:
Create a directive first :
ng g d directive_name

```
C:\java\angular\MyProject3>ng g d HoverHighLight
Node.js version v23.11.0 detected.
Odd numbered Node.js versions will not enter LTS status and should not be used for production. For more information, ple
ase see https://nodejs.org/en/about/previous-releases/.
CREATE src/app/hover-high-light.directive.spec.ts (266 bytes)
CREATE src/app/hover-high-light.directive.ts (189 bytes)
UPDATE src/app/app.module.ts (1416 bytes)

C:\java\angular\MyProject3>
```

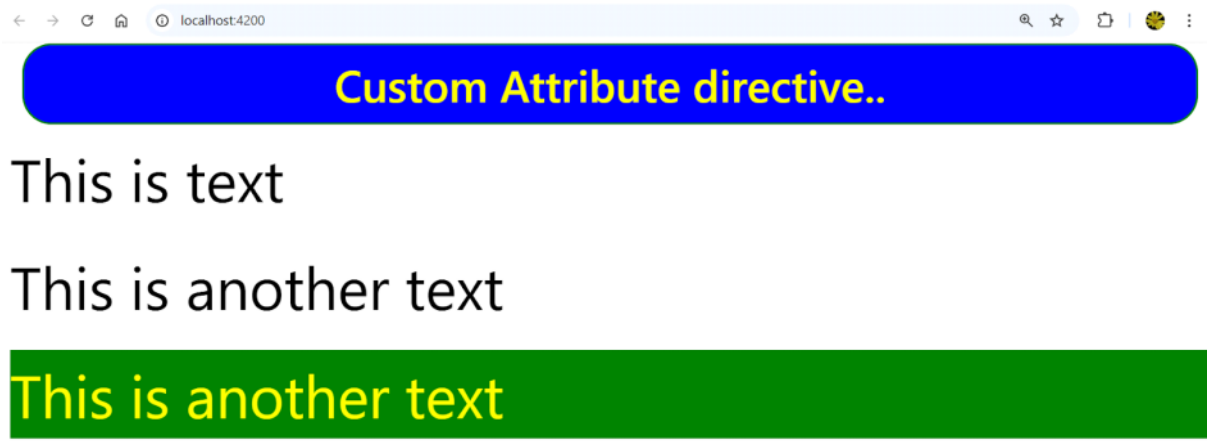
Step 2 ; In Hover.highlight.directive.ts

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

@Directive({
  selector: '[appHoverHighLight]',
  standalone: false
})
export class HoverHighLightDirective {

  @Input ('appHoverHighLight') strBackgroundColor : string = '';
  constructor(private e1 : ElementRef) { }

  @HostListener('mouseenter') onMouseEnter(){
    this.e1.nativeElement.style.backgroundColor = this.strBackgroundColor;
    this.e1.nativeElement.style.color = 'Yellow';
  }

  @HostListener ('mouseleave') onMouseLeave(){
    this.e1.nativeElement.style.backgroundColor = 'white';
    this.e1.nativeElement.style.color = 'black';
  }

}
```

Step 3 : In App.component.html

```
<p [appHoverHighLight]="'green'">This is text</p>
<p [appHoverHighLight]="'Green'">This is another text</p>
<p [appHoverHighLight]="'Green'">This is another text</p>
```

Step 4 : Verify the output.



2. Custom Structural Directive Example

What it is

A **custom structural directive** is a reusable class that gives you the power to manipulate the structure of the DOM (Document Object Model).

It determines **if**, **when**, and **how** a block of HTML is added to or removed from the page.

- You recognize it by the leading asterisk (*) in the template, like *ngIf or *ngFor.
- You use it by injecting **TemplateRef** (the content to potentially render) and **ViewContainerRef** (the location where the content should be placed) to build your custom logic.
- Template Ref - The TemplateRef represents the **template** itself—the section of HTML that is marked with a structural directive. Think of it as a **blueprint** or a **cookie cutter** of the element and its content.
- ViewContainerRef - The ViewContainerRef represents the **container** or the **insertion point** in the DOM where Angular will place one or more views. Think of it as the **workspace** or the **target location** on the page.

Example:

Create the directive using command :

npx ng g d directive-name

```
C:\java\angular\MyProject3>ng g d Delay1
Node.js version v23.11.0 detected.
Odd numbered Node.js versions will not enter LTS status and should not be used for production. For more information, ple
ase see https://nodejs.org/en/about/previous-releases/.
CREATE src/app/delay1.directive.spec.ts (232 bytes)
CREATE src/app/delay1.directive.ts (173 bytes)
UPDATE src/app/app.module.ts (1488 bytes)

C:\java\angular\MyProject3>
```

Step 2 : In App.component.ts

```
  nTimeInSeconds : number = 0;
  nTimeInMillis! : number;

  convertAndDisplay(){
    this.nTimeInMillis = this.nTimeInSeconds * 1000;

  }
```

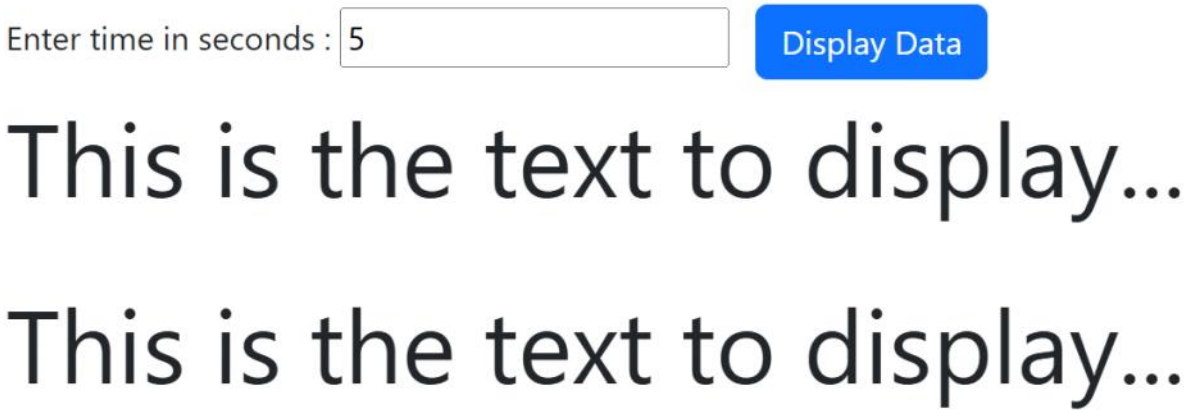
Step 3 : In delay1.directive.ts

```
//User gives the time to delay and then display the text.
export class Delay1Directive {

  constructor(private template : TemplateRef<any>, //HTML
              private container : ViewContainerRef //Container which includes the html.
  ) { }

//HTMl, CSS, BS, JS... Arrow function... hoc...

@Input() set  appDelay1 (delayTime : number){
  setTimeout( () => {
    this.container.createEmbeddedView(this.template);
  }, delayTime)
}
```

Step 4 : In App.component.html

```
 Enter time in seconds : <input type="number" [(ngModel)]="nDelayInSeconds"/>  
 <button class="btn btn-primary" (click)="convertToMills()">Display Data</button>

 <div *appDelay1="nDelayInMillis">
    <p> This is the text to display...</p>

 </div>
```

Step 5 : Check the output



Pipes

13 September 2025 18:59

What are Pipes?

Pipes are a way to transform data within your templates. They take a value as input and return a new, formatted value to display. Think of a pipe as a simple filter that formats your data without changing the original data itself.

The core of a pipe is the **pipe operator** |. You place this operator between the data you want to transform and the name of the pipe you want to use. The data flows from left to right through the pipe.

Syntax: {{ data | pipeName }}

The original variable in your component's TypeScript file remains untouched, ensuring your data integrity.

Common Built-in Pipes and Their Uses

Text and String Pipes

These pipes are used to change how strings appear.

- **uppercase**: Converts all characters to uppercase.
- **lowercase**: Converts all characters to lowercase.
- **slice**: Extracts a part of a string or array. You can specify a start and end point.

Numeric and Currency Pipes

These pipes are used to format numbers.

- **currency**: Formats a number as a currency string, complete with a currency symbol. You can customize the currency code, symbol, and decimal points.
- **decimal**: Formats a number with a specific number of decimal places.

Date and JSON Pipes

- **date**: Formats a date into a user-friendly format (e.g., 'shortDate', 'fullDate').
- **json**: Converts a JavaScript object or array into a JSON string, which is useful for debugging.

Example Code

Here's an example that uses all the pipes mentioned above in a single component.

pipes.component.ts

TypeScript

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

@Component({
  selector: 'app-pipes',
  template: `
    <h3>Angular Pipes</h3>
    <hr>

    <p>Uppercase: {{ title | uppercase }}</p>
    <p>Lowercase: {{ title | lowercase }}</p>
    <p>Slice: {{ longText | slice:0:15 }}...</p>

    <p>Currency: {{ amount | currency:'USD':'symbol':'1.2-2' }}</p>
    <p>Decimal: {{ amount | number:'1.2-2' }}</p>

    <p>Today's Date: {{ today | date:'mediumDate' }}</p>
    <p>User Data (JSON): {{ user | json }}</p>
  `
})
export class PipesComponent {
  title: string = 'Angular Pipes Demo';
  longText: string = 'Angular is a fantastic framework for building web applicatio
  amount: number = 12345.6789;
  today: Date = new Date();
  user = { name: 'John Doe', city: 'New York' };
}
```

Special Pipe: AsyncPipe

13 September 2025 19:33

What is AsyncPipe?

The async pipe is a powerful tool for working with **Observables** and **Promises** in your templates. It's a special pipe because it automatically handles the subscription and unsubscription for you. This means you don't have to manually subscribe to an Observable in your component's code, which helps prevent common memory leaks.

What it does	Automatically subscribes to an Observable or Promise and returns the latest value it emitted.
Why it's used	To simplify asynchronous data handling and ensure proper cleanup (unsubscribing) when your component is destroyed.
Data Flow	The pipe watches the Observable, and as soon as a new value is emitted, the pipe updates the view.

Example

Let's use a simple example where we display a number that changes every second. We'll use an Observable from RxJS to create this stream of numbers.

`app.module.ts`

This is the main module of your application. You need to import `BrowserModule` and declare your new component so Angular knows about it.

TypeScript

```typescript
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AsyncPipeComponent } from './async-pipe.component';

@NgModule({
  // The declarations array tells Angular which components belong to this module.
  declarations: [
    AsyncPipeComponent
  ],
  // The imports array contains other modules your app needs.
  imports: [
    BrowserModule
  ],
  providers: [],
  // The bootstrap array tells Angular which component to load first.
  bootstrap: [AsyncPipeComponent]
})
export class AppModule { }
```

async-pipe.component.ts

This is the complete component file. The template is included directly inside the `@Component` decorator, making it easy to copy and run.

TypeScript

```typescript
import { Component } from '@angular/core';
import { Observable, interval } from 'rxjs';
import { map } from 'rxjs/operators';

@Component({
  selector: 'app-async-pipe',
  // The HTML template is included directly here for simplicity.
  template: `
    <h3>Async Pipe Example</h3>
    <p>Value from Observable: {{ seconds$ | async }}</p>
  `
})
export class AsyncPipeComponent {
  // We create an Observable that emits a new number every second (1000 millisecond
  // The '$' at the end of the variable name is a common convention for Observables
  seconds$: Observable<number> = interval(1000).pipe(
    // We use the map operator to return the value as is.
    map(value => value)
  );
}
```

The Observable Part

1. **The Stream**: In async-pipe.component.ts, seconds$ = interval(1000) creates an **Observable**. Think of this Observable as a special data stream that will continuously emit numbers over time. The number 1000 tells it to emit a new number **every 1000 milliseconds** (1 second).
2. **The Data**: The numbers that this stream emits are sequential, starting from zero: 0, 1, 2, 3, and so on.
   The numbers are generated by the interval function itself. It is specifically designed to emit a sequence of numbers (0, 1, 2, 3, ...) at a set time interval. Each time the 1-second timer completes, interval automatically generates the next number in the sequence and sends it down the stream.

The async Pipe's Role

The async pipe's job is to connect the Observable stream from the ts file to the HTML template. It does this automatically so you don't have to write extra code.

The Flow of Data

1. **Component Initialization**: When the AsyncPipeComponent loads, Angular sees {{ seconds$ | async }} in the template. The async pipe immediately **subscribes** to the seconds$ Observable stream.
2. **First Emission**: One second after subscribing, the seconds$ Observable emits its first value: the number 0.
3. **View Update**: The async pipe receives this value and updates the <p> tag in the HTML. The page now displays **"Value from Observable: 0"**.
4. **Second Emission**: One second later, the Observable emits the number 1.
5. **Continuous Update**: The async pipe receives this new value and updates the HTML again, so the page now shows **"Value from Observable: 1"**.
6. **Repeat**: This cycle repeats indefinitely, with the number updating every second, until the component is destroyed. The async pipe then automatically **unsubscribes** from the stream to stop the flow.

Custom Pipes

13 September 2025 19:48

What are Custom Pipes?

A **custom pipe** is a class you create to format data in a unique way that isn't possible with built-in pipes.
It's used when you have a specific, repeated data formatting need. You create a pipe class with the **@Pipe** decorator and a **transform** method, and then use it in your template with the pipe operator (|).

Example:
Let's create a custom pipe that takes a number and calculates it to a given power (e.g., 2^3). This example also shows how to pass an argument to a pipe.

Create a pipe using

Npx ng g p PipeName

1. Creating the Pipe (`power-of.pipe.ts`)

This is the file where you define the custom pipe's logic.

TypeScript

```typescript
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
  // The 'name' is how you'll use the pipe in your template.
  name: 'powerOf'
})
export class PowerOfPipe implements PipeTransform {

  // The 'transform' method is where the pipe's logic goes.
  // It takes the value and any optional arguments (like 'exponent').
  transform(value: number, exponent: string): number {
    // The exponent is a string, so we convert it to a number.
    const exp = parseFloat(exponent);
    // We use Math.pow to calculate the result and return it.
    return Math.pow(value, exp);
  }
}
```

2. Using the Pipe (`app.component.ts` and `.html`)

This is how you use your new pipe in a component.

`app.component.ts`

TypeScript

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html'
})
export class AppComponent {
  // Our base number for the pipe.
  baseNumber: number = 2;
}
```

`app.component.html`

HTML

```html
<div>
  <h2>Custom Pipe Example</h2>
  <p>The number 2 to the power of 3 is: {{ baseNumber | powerOf:'3' }}</p>
  <p>The number 2 to the power of 5 is: {{ baseNumber | powerOf:'5' }}</p>
</div>
```

Note: You must also add the `PowerOfPipe` to the `declarations` array of your `AppModule` so that Angular knows about it.

The Flow of Data

1. **Component Data**: In app.component.ts, the baseNumber variable is set to 2.
2. **Pipe Call**: In app.component.html, Angular sees {{ baseNumber | powerOf:'3' }}. It takes the value 2 and passes it as the value argument to the powerOf pipe's transform method. It also passes the string '3' as the exponent argument.
3. **Transformation**: The transform method in power-of.pipe.ts receives 2 and '3'. It converts '3' to a number and calculates 23, which equals 8.
4. **Display**: The pipe returns the number 8. Angular takes this returned value and displays it in the p tag.

13 September 2025 19:55

What are Angular Forms?

Angular forms are a way to handle user input. Angular provides two main approaches for building forms: **Template-driven** and **Reactive**.

Template-driven

13 September 2025 19:57

Template-driven forms are a simple way to build forms almost entirely in your HTML template. They are best suited for basic forms with straightforward logic.
This majorly depends the **[(ngModel)]** directive for two-way data binding on each form control. When you use this, Angular automatically attaches the **NgForm** directive to the <form> tag, which manages the form's overall state and validation.
To use them, you must import the **FormsModule**.

Data Flow and Example

Flow:

1. In app.component.ts, you create an object to hold the form data.
2. In app.component.html, you wrap your form controls in a <form> tag.
3. For each form control (e.g., <input>), you use [(ngModel)] to bind it to a property of your data object.
4. Angular automatically attaches the ngForm directive to the form, which tracks the state of all the form controls.
5. When the user types, the [(ngModel)] binding instantly updates the data object in your app.component.ts.
6. When the form is submitted, you can access the data object to get all the user's inputs

Example:

app.module.ts

TypeScript

```typescript
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule } from '@angular/forms'; // 1. Import FormsModule
import { AppComponent } from './app.component';

@NgModule({
  declarations: [AppComponent],
  imports: [
    BrowserModule,
    FormsModule // 2. Add FormsModule to the imports array
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

app.component.ts

TypeScript

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html'
})
export class AppComponent {
  // 3. Create a data object to hold the form values.
  // The properties (name, feedback) must match the ngModel names in the template.
  feedback = {
    name: '',
    feedbackText: ''
  };

  onSubmit(): void {
    // 6. When the form is submitted, this method is called.
    // The 'feedback' object is already updated with the user's input.
    console.log('Form submitted!', this.feedback);
  }
}
```

app.component.html

HTML

```html
<div>
  <h2>Feedback Form</h2>
  <form (ngSubmit)="onSubmit()">

    <label for="name">Name:</label>
    <input type="text" id="name" name="name" [(ngModel)]="feedback.name" required>

    <br><br>

    <label for="feedbackText">Feedback:</label>
    <textarea id="feedbackText" name="feedbackText" [(ngModel)]="feedback.feedback

    <br><br>

    <button type="submit">Submit</button>
  </form>

  <p>Live Data: {{ feedback | json }}</p>
</div>
```

Reactive Forms

13 September 2025 20:20

What are Reactive Forms?

Reactive Forms is a framework for creating forms in a **model-driven** way. This means you build and manage the form's data model and its validation rules entirely in your component's TypeScript class. The HTML template is then simply a visual representation that binds to this model.

The key players are:

- **FormControl**: This class is used to manage a single input field, tracking its value and validation status.
- **FormGroup**: This class groups multiple FormControls together to represent the entire form.

Data Flow and Example

Flow:

1. **The Blueprint (.ts)**: In app.component.ts, you inject the FormBuilder service. You then use its .group() method to create the form's blueprint, a container that holds all your form controls.
2. **The Fields (.ts)**: Inside the .group() method, you define a FormControl for each input field (name, feedbackText). This is the form model, and it can also include validation rules directly.
3. **The HTML Connects**: In app.component.html, you use [formGroup]="feedbackForm" on the <form> tag. This tells Angular, "This HTML form should follow the blueprint named feedbackForm."
4. **The Data Path**: On each <input>, you use formControlName="name". This creates a direct link between that HTML input and the name field on your TypeScript blueprint.

Example:

Import the ReactiveFormsModule in app.module.ts

app.module.ts

TypeScript

```typescript
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { ReactiveFormsModule } from '@angular/forms'; // 1. Must import this modul
import { AppComponent } from './app.component';

@NgModule({
  declarations: [AppComponent],
  imports: [
    BrowserModule,
    ReactiveFormsModule // 2. Add it to the imports array
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

app.component.ts

TypeScript

```typescript
import { Component } from '@angular/core';
import { FormBuilder, FormGroup, Validators } from '@angular/forms'; // 3. Import Fo

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html'
})
export class AppComponent {
  // 4. This is the main form blueprint, a container for our form controls.
  feedbackForm: FormGroup;

  // 5. We inject FormBuilder into the constructor.
  constructor(private formBuilder: FormBuilder) {
    // 6. We use the FormBuilder service to create our form group.
    this.feedbackForm = this.formBuilder.group({
      // 7. Each field is created with its initial value and validation rules.
      // We add Validators.required to make this field mandatory.
      name: ['', Validators.required],
      feedbackText: ['']
    });
  }

  onSubmit(): void {
    // 8. When the form is submitted, we can easily get all the data from our bluepr
    console.log('Form submitted!', this.feedbackForm.value);
  }
}
```

app.component.html

HTML

```html
<div>
  <h2>Reactive Form</h2>
  <form [formGroup]="feedbackForm" (ngSubmit)="onSubmit()">

    <label for="name">Name:</label>
    <input type="text" id="name" formControlName="name">
    <div *ngIf="feedbackForm.get('name')?.invalid && feedbackForm.get('name')?.touch
      Name is required.
    </div>

    <br><br>

    <label for="feedbackText">Feedback:</label>
    <textarea id="feedbackText" formControlName="feedbackText"></textarea>

    <br><br>

    <button type="submit" [disabled]="feedbackForm.invalid">Submit</button>
  </form>

  <p>Live Data: {{ feedbackForm.value | json }}</p>
</div>
```

Basic Validators

13 September 2025 23:05

What are Basic Validations?

Basic validations are a set of built-in functions provided by Angular for checking common form requirements. They are the easiest way to ensure a user's input is correct without writing any extra code. These validators are part of the **Validators** class.

How to Apply Basic Validations

Applying basic validations is a two-step process: first, you define the rules in your TypeScript file, and then you display the error messages in your HTML template.

Step 1: Defining the Rules in TypeScript (.ts)

You add validators to each FormControl when you build your form using FormBuilder.

- **For a single validator:** Add the validator directly after the initial value. name: [" ", Validators.required]
- **For multiple validators:** Place all the validators in an array. email: ['', [Validators.required, Validators.email]]

Step 2: Displaying Errors in HTML (.html)

You use the state of the form controls to conditionally show error messages. The most important properties to check are:

- **.invalid**: A boolean property that's true if the control's value fails any of its validation rules.
- **.touched**: A boolean property that becomes true after the user has focused on and moved away from the input. This is used to prevent showing errors as soon as the page loads.
- **.hasError('validatorName')**: A method that checks if a specific validator has failed. The name is the same as the validator you used (e.g., 'required', 'email').

Example

```
import { Component } from '@angular/core';
import { FormBuilder, FormGroup, Validators } from '@angular/forms'; // 1. Import necessary classes for building reactive forms

@Component({
  selector: 'app-root',
  // 2. The component's template. The form is defined here.
  template: `
    <h2>Reactive Forms with Basic Validation</h2>
    <form [formGroup]="feedbackForm" (ngSubmit)="onSubmit()">

      <label for="name">Name:</label>
      <input type="text" id="name" formControlName="name">
      <div *ngIf="feedbackForm.get('name')?.invalid && feedbackForm.get('name')?.touched">
        <p *ngIf="feedbackForm.get('name')?.hasError('required')">Name is required.</p>
      </div>

      <br><br>

      <label for="email">Email:</label>
      <input type="text" id="email" formControlName="email">
      <div *ngIf="feedbackForm.get('email')?.invalid && feedbackForm.get('email')?.touched">
        <p *ngIf="feedbackForm.get('email')?.hasError('required')">Email is required.</p>
        <p *ngIf="feedbackForm.get('email')?.hasError('email')">Please enter a valid email.</p>
      </div>

      <br><br>

      <button type="submit" [disabled]="feedbackForm.invalid">Submit</button>
    </form>
  `
})
```

```
export class AppComponent {
  // 13. Declare a FormGroup variable to hold the form model.
  feedbackForm: FormGroup;

  // 14. Inject the FormBuilder service into the constructor.
  constructor(private formBuilder: FormBuilder) {
    // 15. Initialize the FormGroup using FormBuilder's group() method.
    this.feedbackForm = this.formBuilder.group({
      // 16. Define the 'name' FormControl. It starts with an empty string and has a 'required' validator.
      name: ['', Validators.required],
      // 17. Define the 'email' FormControl. It has a 'required' and an 'email' validator.
      email: ['', [Validators.required, Validators.email]]
    });
  }

  // 18. This method is called when the form is submitted.
  onSubmit(): void {
    // 19. Check if the entire form is valid before processing.
    if (this.feedbackForm.valid) {
      console.log('Form is valid!', this.feedbackForm.value);
    } else {
      console.log('Form is invalid.');
    }
  }
}
```

How to Use the Custom Validation in HTML

To use the custom validation in your HTML, you access the error from the FormGroup itself, not the individual controls. You then use *ngIf to display a message based on your custom error name.

- **form.hasError('yourCustomErrorName')**: You use this method on the FormGroup to check for your specific custom error.
- **form.get('controlName')?.touched**: You combine the error check with a touched check on one of the fields to ensure the error message only appears after the user has interacted with the form.

App.component.html

```
<div>
  <h2>Custom Validation Example</h2>
  <form [formGroup]="passwordForm" (ngSubmit)="onSubmit()">

    <label for="password">Password:</label>
    <input type="password" id="password" formControlName="password">
    <div *ngIf="passwordForm.get('password')?.invalid && passwordForm.get('password')?.touched">
      <p *ngIf="passwordForm.get('password')?.hasError('required')">Password is required.</p>
    </div>

    <br><br>

    <label for="confirmPassword">Confirm Password:</label>
    <input type="password" id="confirmPassword" formControlName="confirmPassword">
    <div *ngIf="passwordForm.get('confirmPassword')?.invalid && passwordForm.get('confirmPassword')?.touched">
      <p *ngIf="passwordForm.get('confirmPassword')?.hasError('required')">Confirm Password is required.</p>
    </div>

    <br><br>

    <div *ngIf="passwordForm.hasError('passwordMismatch') && passwordForm.get('confirmPassword')?.touched">
      <p>Passwords do not match.</p>
    </div>

    <br><br>

    <button type="submit" [disabled]="passwordForm.invalid">Submit</button>
  </form>
</div>
```

Custom Validation

13 September 2025 23:54

A custom validation is a function you create yourself to handle specific logic that Angular's built-in validators don't cover. A perfect example is checking if a password matches a confirmation password field.
Here's how to create a custom validator properly, covering all the key parts.

The Keywords and Code (.ts File)

A custom validator is a function that takes an AbstractControl and returns ValidationErrors or null.

- **AbstractControl**: This is the parent class for FormControl and FormGroup. By using AbstractControl as the function's parameter, your validator can be used on either a single form field or an entire form group.
- **ValidationErrors**: This is the type that the function must return if the validation fails. It's a simple object where the key is the name of your custom error (e.g., 'passwordMismatch').
- **null**: Your custom validator must return null if the validation passes successfully.
- **{ validators: ... }**: This is the key property you use to attach your custom validator to the FormGroup. You pass your function here, not on an individual FormControl.

App.component.ts

```ts
import { Component } from '@angular/core';
import { FormBuilder, FormGroup, Validators, AbstractControl, ValidationErrors } from '@angular/forms';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html'
})
export class AppComponent {
  passwordForm: FormGroup;

  constructor(private formBuilder: FormBuilder) {
    // 1. We use FormBuilder to create the form group for the password fields.
    this.passwordForm = this.formBuilder.group({
      password: ['', Validators.required],
      confirmPassword: ['', Validators.required]
    }, { validators: this.passwordMatchValidator }); // 2. We attach the custom validator to the whole form group.
  }

  // 3. This is our custom validator function. It takes an AbstractControl (which in this case is the entire form group).
  passwordMatchValidator(control: AbstractControl): ValidationErrors | null {
    const password = control.get('password')?.value;
    const confirmPassword = control.get('confirmPassword')?.value;

    // 4. We check if the passwords match. If they do, the validation passes, and we return null.
    if (password === confirmPassword) {
      return null;
    }

    // 5. If they do not match, we return a custom error object. The key 'passwordMismatch' is our custom error name.
    return { passwordMismatch: true };
  }

  onSubmit(): void {
    if (this.passwordForm.valid) {
      console.log('Form is valid! Passwords match.');
    } else {
      console.log('Form is invalid.');
    }
  }
}
```

How to Use the Custom Validation in HTML

To use the custom validation in your HTML, you access the error from the FormGroup itself, not the individual controls. You then use *ngIf to display a message based on your custom error name.

- **form.hasError('yourCustomErrorName')**: You use this method on the FormGroup to check for your specific custom error.
- **form.get('controlName')?.touched**: You combine the error check with a touched check on one of the fields to ensure the error message only appears after the user has interacted with the form.

App.component.html

```
<div>
  <h2>Custom Validation Example</h2>
  <form [formGroup]="passwordForm" (ngSubmit)="onSubmit()">

    <label for="password">Password:</label>
    <input type="password" id="password" formControlName="password">
    <div *ngIf="passwordForm.get('password')?.invalid && passwordForm.get('password')?.touched">
      <p *ngIf="passwordForm.get('password')?.hasError('required')">Password is required.</p>
    </div>

    <br><br>

    <label for="confirmPassword">Confirm Password:</label>
    <input type="password" id="confirmPassword" formControlName="confirmPassword">
    <div *ngIf="passwordForm.get('confirmPassword')?.invalid && passwordForm.get('confirmPassword')?.touched">
      <p *ngIf="passwordForm.get('confirmPassword')?.hasError('required')">Confirm Password is required.</p>
    </div>

    <br><br>

    <div *ngIf="passwordForm.hasError('passwordMismatch') && passwordForm.get('confirmPassword')?.touched">
      <p>Passwords do not match.</p>
    </div>

    <br><br>

    <button type="submit" [disabled]="passwordForm.invalid">Submit</button>
  </form>
</div>
```

Form Array

26 September 2025 21:58

A **FormArray** is a class in Angular's **Reactive Forms** module designed to manage a **dynamic list** (an array) of form controls or form groups.

What it Represents

- It's perfect for forms where users need to add or remove repeating elements, such as a list of phone numbers, addresses, or skills.
- Unlike a fixed FormGroup, the number of items in a FormArray is not determined in advance; you control its length programmatically in your TypeScript code.

Core Operations

In your component's class, you use methods on the FormArray instance to dynamically control the view:

1. **Adding Items:** Use the **.push()** method to add a new FormControl or FormGroup to the array.
2. **Removing Items:** Use the **.removeAt(index)** method to delete an item at a specific position.

```
import { FormBuilder, FormGroup, FormArray, Validators } from '@angular/forms';

// In your component class:
export class DynamicFormExample {
  userForm: FormGroup;

  // 1. Inject FormBuilder in the constructor
  constructor(private fb: FormBuilder) {
    this.userForm = this.fb.group({
      name: ['', Validators.required],
      // 2. Initialize the FormArray as an empty list
      skills: this.fb.array([])
    });
  }

  // 3. Convenience getter to easily access the FormArray instance
  get skills(): FormArray {
    return this.userForm.get('skills') as FormArray;
  }
```

```
  // 4. Method to define a new item (FormGroup) structure
  private createSkillGroup(): FormGroup {
    return this.fb.group({
      skillName: ['', Validators.required],
      level: ['Beginner']
    });
  }

  // 5. Method to add a new item to the FormArray
  addSkill(): void {
    this.skills.push(this.createSkillGroup());
  }

  // 6. Method to remove an item by its index
  removeSkill(index: number): void {
    this.skills.removeAt(index);
  }
}
```

Service

14 September 2025 10:30

An Angular service is a class used to **encapsulate reusable logic**, such as data fetching and manipulation. The main goal of a service is to separate the business logic from your components, making your application cleaner and more maintainable.
The most common use for a service is to handle **CRUD (Create, Read, Update, Delete)** operations. By centralizing these tasks in a service, your components become lightweight and focused on displaying data, while the service manages all communication with a data source like a backend API.

Here is a complete example of a service performing CRUD operations using a mock backend.

Create a service :

Npx ng g s Service_name

The Service (feedback.service.ts)

This service acts as the data layer, responsible for all communication with the backend. It uses Angular's HttpClient to send requests and receives data as Observables because these operations are asynchronous.

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

@Injectable({
  providedIn: 'root' // Makes this a singleton service available throughout the app.
})
export class FeedbackService {
  // The URL of our backend API endpoint (e.g., from a JSON Server).
  private apiUrl = 'http://localhost:3000/feedback';

  // We inject HttpClient to make API calls.
  constructor(private http: HttpClient) { }

  // READ: Fetches all feedback entries from the backend.
  getFeedback(): Observable<any> {
    return this.http.get(this.apiUrl);
  }

  // CREATE: Sends a new feedback entry to the backend.
  createFeedback(feedbackData: any): Observable<any> {
    return this.http.post(this.apiUrl, feedbackData);
  }

  // UPDATE: Sends an updated feedback entry to the backend for a specific ID.
  updateFeedback(id: number, feedbackData: any): Observable<any> {
    return this.http.put(`${this.apiUrl}/${id}`, feedbackData);
  }

  // DELETE: Sends a request to remove a feedback entry by its ID.
  deleteFeedback(id: number): Observable<any> {
    return this.http.delete(`${this.apiUrl}/${id}`);
  }
}
```

The Component (app.component.ts)

This component's job is to inject the FeedbackService, call its methods, and display the data.

It subscribes to the Observables to handle the asynchronous responses.

app.component.ts

This is the component's TypeScript file. The `template` property is replaced with `templateUrl` to link to a separate HTML file.

TypeScript

```typescript
import { Component, OnInit } from '@angular/core';
import { FeedbackService } from './feedback.service';

@Component({
  selector: 'app-root',
  // The templateUrl property links this component to its separate HTML file.
  templateUrl: './app.component.html'
})
export class AppComponent implements OnInit {
  feedbackEntries: any[] = [];
  newFeedbackName: string = '';
  newFeedbackMessage: string = '';

  // 1. We inject the FeedbackService into the component's constructor.
  constructor(private feedbackService: FeedbackService) { }

  // 2. We call the service to fetch data when the component initializes.
  ngOnInit(): void {
    this.getFeedbackEntries();
  }

  // READ Operation: This method calls the service's getFeedback() method.
  getFeedbackEntries(): void {
    this.feedbackService.getFeedback().subscribe(
      (data) => {
        // We subscribe to the Observable to receive the data.
        this.feedbackEntries = data;
      },
      (error) => {
        console.error('Error fetching feedback:', error);
      }
    );
  }

  // CREATE Operation: This method calls the service to create a new entry.
  onCreate(): void {
    const newEntry = { name: this.newFeedbackName, message: this.newFeedbackMessag
    this.feedbackService.createFeedback(newEntry).subscribe(
      () => { this.getFeedbackEntries(); } // Refresh the list after the operation
    );
  }

  // UPDATE Operation: This method calls the service to update an entry.
  onUpdate(id: number): void {
    const updatedEntry = { id: id, name: 'Updated Name', message: 'Updated Message
    this.feedbackService.updateFeedback(id, updatedEntry).subscribe(
      () => { this.getFeedbackEntries(); } // Refresh the list.
    );
  }

  // DELETE Operation: This method calls the service to delete an entry.
  onDelete(id: number): void {
    this.feedbackService.deleteFeedback(id).subscribe(
      () => { this.getFeedbackEntries(); } // Refresh the list.
    );
  }
}
```

app.component.html

This file contains only the HTML template for the component.

HTML

```html
<h2>Feedback Entries</h2>

<div *ngFor="let entry of feedbackEntries">
  <p><strong>{{ entry.name }}</strong>: {{ entry.message }}</p>
  <button (click)="onUpdate(entry.id)">Update</button>
  <button (click)="onDelete(entry.id)">Delete</button>
  <hr>
</div>

<h3>Add New Feedback</h3>
<input type="text" [(ngModel)]="newFeedbackName" placeholder="Name">
<input type="text" [(ngModel)]="newFeedbackMessage" placeholder="Message">
<button (click)="onCreate()">Create</button>
```

Imp concepts

21 September 2025 14:52

Observable

An **Observable** is a stream of data that can be used to handle asynchronous events and it does nothing until something **subscribes** to it.
When the Observable gets data (for example, after an HTTP request is complete), it starts **emitting** that data. Any component or element that has subscribed to the Observable is an **Observer**.
The Observable then automatically pushes the data to its Observers. This process calls the Observer's next() method, which automatically receives the data.
This creates a one-way, reactive flow where a component doesn't have to ask for data; it simply gets it when it becomes available.

- **Key Concepts**:
    - **Laziness**: An Observable won't run its code until something subscribes to it.
    - **Asynchronous**: It handles operations that don't happen immediately, like network requests or user clicks.

Observer & Subscription

An **Observer** is an object with three methods that define how to handle the data from an Observable:

1. **next()**: Called for every new value the Observable emits. This is where you get the actual data.
2. **error()**: Called if an error occurs. The Observable stream then stops.
3. **complete()**: Called when the Observable has finished its job and won't emit any more values. The stream then stops.

When you call an Observable's **subscribe()** method, you are creating a **Subscription**. The Subscription is a crucial link that connects the Observer to the Observable, causing the Observable to start emitting data.

Operators 🛠️

Operators are functions that help you process and transform the data flowing through an Observable stream. They are used with the **.pipe()** method. **Example Operators**:

- **map()**: Transforms each value in the stream into a new value.
- **filter()**: Filters out unwanted values.
- **tap()**: Lets you "peek" into the stream to run side effects (like logging) without changing the data.

The Flow: A Simple Example 🌊

1. A service returns an **Observable** from an HTTP call. The Observable is lazy and does nothing.
2. A component calls the service method and uses **subscribe()** to start the process.
3. The **Observable** performs the HTTP request.
4. When data arrives, the **Observable** emits it, and the **next()** method in the component's subscribe() is called, giving you the data.
5. If the request fails, the **error()** method is called.
6. Once the request is complete, the **complete()** method is called, and the **Subscription** is automatically ended.

Dependency Injection

26 September 2025 23:04

Dependency Injection (DI) is a core software design pattern where a class receives its required **dependencies** (other objects or services) from an external source, rather than creating those dependencies itself.

The pattern involves three roles:

1. **Consumer** (e.g., a Component): The class that needs the service.
2. **Provider** (e.g., a Service): The object that is created and supplied.
3. **Injector**: Angular's mechanism that creates the service instance and passes it to the Consumer's constructor.

Hierarchical Dependency Injection (DI)

Hierarchical DI is Angular's way of organizing all the application's services into a **nested structure of containers** (called **Injectors**) that mirrors the component tree.

How It Works (The Search Process)

When a component needs a service:

1. **Bottom-Up Search:** Angular starts searching for the service's provider in the component's **local Injector**.
2. **Ascending the Tree:** If the service is not found, the search automatically moves up to the parent component's Injector, continuing up the tree until it finds a match.

Routing

29 September 2025 08:08

Routing in Angular is the mechanism that allows navigation between different views or components in a single-page application (SPA) without reloading the whole page.

Router link

29 September 2025 08:23

📌 Small Description

In Angular, routing lets you move between components without reloading the page.
The routerLink directive is used in templates to define navigation paths.

📌 Small Example
In app-rouitng.module.ts you configure the routes

app-routing.module.ts
```
const routes: Routes = [
  { path: 'home', component: HomeComponent },
  { path: 'about', component: AboutComponent }
];
```

app.component.html

```
<nav>
  <a routerLink="/home">Home</a> |
  <a routerLink="/about">About</a>
</nav>

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

☑ Clicking the links swaps the content inside <router-outlet> without refreshing the page.

Navigation between Different Routes

14 September 2025 19:25

Programmatic navigation is a way to navigate between pages using your component's **TypeScript code** instead of clicking a link in the template.
You use it when navigation depends on **logic**, such as after a form is successfully submitted or an API call returns a successful response.

How to Use

1. **Inject the Router service**: You add the Router service to your component's constructor, and Angular automatically provides it.
2. **Use the .navigate() method**: You call router.navigate() and pass it the path you want to go to.

Complete Code Example

This example shows a LoginComponent that uses the Router service to navigate to a DashboardComponent after a "successful" login.

1. The Routing Module (`app-routing.module.ts`)

First, you must define the routes that you'll be navigating to.

TypeScript

```typescript
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { DashboardComponent } from './dashboard/dashboard.component';
import { LoginComponent } from './login/login.component';
import { UserProfileComponent } from './user-profile/user-profile.component';

const routes: Routes = [
  { path: '', redirectTo: '/login', pathMatch: 'full' },
  { path: 'login', component: LoginComponent },
  { path: 'dashboard', component: DashboardComponent },
  { path: 'user-profile/:id', component: UserProfileComponent } // A route with a pa
];

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

2. The Component with Navigation (`login.component.ts`)

This component contains the logic for programmatic navigation.

TypeScript

```typescript
import { Component } from '@angular/core';
import { Router } from '@angular/router'; // 1. Import the Router service

@Component({
  selector: 'app-login',
  template: `
    <h2>Login</h2>
    <p>Click the button to simulate a successful login and navigate to the dashboard
    <button (click)="onLoginSuccess()">Log In</button>

    <br><br>

    <p>Click the button below to navigate to a user profile with ID 101.</p>
    <button (click)="goToUserProfile()">Go to User Profile</button>
  `,
})
export class LoginComponent {

  // 2. Inject the Router service in the constructor.
  constructor(private router: Router) { }

  // 3. This method is called by the 'Log In' button.
  onLoginSuccess(): void {
    // 4. Perform a programmatic navigation to the '/dashboard' route.
    this.router.navigate(['/dashboard']);
  }

  // 5. This method shows how to navigate with a route parameter.
  goToUserProfile(): void {
    const userId = 101;
    // 6. The navigate method accepts an array. The first element is the path,
    //    and subsequent elements are the parameters.
    this.router.navigate(['/user-profile', userId]);
  }
}
```

Sharing data between Routes

14 September 2025 22:01

The Core Concepts

In Angular, you can pass data between routes using two main methods, each for a different purpose:

- **Route Parameters:** For data that is **required** to identify a specific resource. Think of it as a house number in a URL. Example: /product/123.
- **Query Parameters:** For data that is **optional** and used for things like filtering or sorting. Think of it as a note on a door. Example: /products?sort=price.

The **ActivatedRoute** service is your primary tool for accessing this data in the receiving component.

1. Route Parameters (For Required Data)

This method is for passing data that is crucial for identifying a specific resource, such as a user ID or a product ID. The data is embedded directly within the URL path.

Key Notes for an Interview

- **How to Define:** You use a **colon (:)** in the route configuration to define a placeholder, like path: 'product/:id'.
- **How to Send:** Use routerLink or router.navigate() with an array of route segments. Example: ['/product', 123].
- **How to Receive:** In the receiving component, you **inject** the **ActivatedRoute** service. You can then get the parameter's value using the **snapshot** or **paramMap**.
- **Keywords:** ActivatedRoute, snapshot, paramMap, routerLink, router.navigate().

Example: Viewing a Product's Details

Step 1: define the route in app-route.module.ts

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { ProductListComponent } from './product-list.component';
import { ProductDetailComponent } from './product-detail.component';

const routes: Routes = [
  { path: 'products', component: ProductListComponent },
  { path: 'products/:id', component: ProductDetailComponent } // ':id' is the route parameter
];

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

Step2: product-list.component.ts (Sender). We passed the data using the routeLink

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

@Component({
  selector: 'app-product-list',
  template: `
    <h2>Product List</h2>
    <p>Click a link to view a product's details:</p>
    <a [routerLink]="['/products', 101]">View Product 101</a>
    <br>
    <a [routerLink]="['/products', 102]">View Product 102</a>
  `
})
export class ProductListComponent { }
```

Step3: product-detail.component.ts

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

@Component({
  selector: 'app-product-detail',
  template: `
    <h2>Product Details</h2>
    <p>Product ID from URL: {{ productId }}</p>
  `
})
export class ProductDetailComponent implements OnInit {
  productId: string | null = null;

  // 1. Inject the ActivatedRoute service.
  constructor(private route: ActivatedRoute) { }

  ngOnInit(): void {
    // 2. Read the parameter from the URL's paramMap.
    this.productId = this.route.snapshot.paramMap.get('id');
  }
}
```

2. Query Parameters (For Optional Data)

This method is for passing optional data, such as filters, sorting options, or page numbers. The data is appended to the URL after a question mark and is not part of the route's path.

Key Notes for an Interview

- **How to Define:** You don't need to change the route configuration.
- **How to Send:** You use the **queryParams** property with routerLink. Example: [queryParams]="{ sort: 'price' }".
- **How to Receive:** You **inject** the **ActivatedRoute** service and use its **queryParamMap**.
- **Crucial Concept:** The **queryParamMap** is an **Observable**. This is a critical point. You must **subscribe()** to it to get the data because the parameters can change without the component being destroyed and reloaded.
- **Keywords:** queryParams, queryParamMap, Observable, subscribe.

Example: Sorting a Product List

Step 2: search.component.ts (Sender)

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

@Component({
  selector: 'app-search',
  template: `
    <h3>Search and Sort</h3>
    <a routerLink="/products" [queryParams]="{ sort: 'price' }">Sort by Price</a>
    <!-- using [queryParams], we are passin the values. This are optional. -->
    <br>
    <a routerLink="/products" [queryParams]="{ filter: 'new' }">Filter by New</a>
  `
})
export class SearchComponent { }
```

Step 3: product-list.component.ts (Receiver)

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

@Component({
  selector: 'app-product-list',
  template: `
    <h2>Product List</h2>
    <p>Current sort order: {{ sortOrder }}</p>
  `
})
export class ProductListComponent implements OnInit {
  sortOrder: string | null = null;

  constructor(private route: ActivatedRoute) { }

  ngOnInit(): void {
    // We subscribe to the queryParamMap to read the optional parameters.
    this.route.queryParamMap.subscribe(params => {
      this.sortOrder = params.get('sort');
      console.log('The sort parameter is:', this.sortOrder);
    });
  }
}
```

Comparison Summary

Feature	Route Parameters	Query Parameters
Purpose	Required data (e.g., resource ID)	Optional data (e.g., filters)
URL Format	Part of the path (/products/123)	Appended after a ? (/products?sort=price)
Route Config	Requires a placeholder (:id)	No changes needed
Data Access	**snapshot.paramMap**	**queryParamMap** (an Observable)

3) Shared Service

A shared service is the most reliable way to share data between any components in your application, especially those that aren't directly related (e.g., they don't have a parent-child relationship). It acts as a central hub for data.

Professional Notes

- **Purpose:** To create a single source of truth for shared data that any component can access.
- **Key Concept:** Services are singletons by default, meaning there's only one instance of the service in the application. This instance holds the data.
- **Keywords:** @Injectable(), BehaviorSubject (a type of Observable), subscribe().

Shared Service Keywords

- **@Injectable()**: A decorator that marks a class as a service. It tells Angular that this service can be **injected** into other components or services, making it available for use throughout the application. It also makes the service a **singleton**, meaning only one instance of it exists.
- **BehaviorSubject**: This is a type of **Observable** that holds a single, current value. When a component subscribes to it, it immediately receives the last value, and then continues to receive new values as they are sent. It's perfect for data that needs to be accessed and updated in real-time.
- **Observable**: A data stream that can emit multiple values over time. It represents data that arrives asynchronously. Components **subscribe** to an Observable to receive the data when it becomes available.
- **subscribe()**: A method used to listen for data from an Observable. When

you call subscribe(), you're telling Angular to give you the data as soon as the Observable sends it.

Proper Code Example with Comments This example shows a DataService that holds a message. A SenderComponent updates the message, and a ReceiverComponent gets the message in real-time.

Example :

Proper Code Example with Comments

This example shows a `DataService` that holds a message. A `SenderComponent` updates the message, and a `ReceiverComponent` gets the message in real-time.

1. The Service (`data.service.ts`)

TypeScript

```typescript
import { Injectable } from '@angular/core';
import { BehaviorSubject } from 'rxjs'; // 1. Import BehaviorSubject

@Injectable({
  providedIn: 'root' // This makes the service a singleton.
})
export class DataService {
  // 2. Use a BehaviorSubject to hold the data. It's an Observable that holds a value
  // We initialize it with an empty string.
  private messageSource = new BehaviorSubject<string>('');

  // 3. We create a public property that other components can subscribe to.
  currentMessage = this.messageSource.asObservable();

  // 4. This method allows other components to change the data.
  changeMessage(message: string): void {
    this.messageSource.next(message); // .next() sends the new data to all subscriber
  }
}
```

2. The Sender Component (`sender.component.ts`)

TypeScript

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

@Component({
  selector: 'app-sender',
  template: `
    <h2>Sender Component</h2>
    <input type="text" #inputMessage>
    <button (click)="sendMessage(inputMessage.value)">Send Message</button>
  `
})
export class SenderComponent {
  // 5. We inject the DataService into the component.
  constructor(private dataService: DataService) { }

  sendMessage(message: string): void {
    // 6. We call the service's method to update the data.
    this.dataService.changeMessage(message);
  }
}
```

3. The Receiver Component (`receiver.component.ts`)

TypeScript

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

@Component({
  selector: 'app-receiver',
  template: `
    <h2>Receiver Component</h2>
    <p>Received Message: {{ receivedMessage }}</p>
  `
})
export class ReceiverComponent implements OnInit {
  receivedMessage: string = '';

  constructor(private dataService: DataService) { }

  ngOnInit(): void {
    // 7. We subscribe to the service's Observable to get real-time updates.
    this.dataService.currentMessage.subscribe(message => {
      this.receivedMessage = message; // We assign the new message to our local prope
    });
  }
}
```

4) Router State (history.state)

This method is for passing temporary data during a programmatic navigation without putting it in the URL. It's useful for passing simple information that you don't need to persist on a page reload.

Professional Notes

- **Purpose:** To pass transient (non-persistent) data like a success message or an

object.

- **Key Concept:** The data is attached to the navigation event itself and is stored in the browser's history.state property.
- **Important Limitation:** The data is **lost if the page is reloaded** or a new URL is typed.
- **Keywords:** Router, router.navigate(), state, history.state.

Router State Keywords

- **history.state**: A standard browser API property that allows you to access data that was passed during a navigation event. It's a key feature used by the Angular Router to pass data between routes without putting it in the URL.
- **router.navigate()**: A method on the **Router service** used for **programmatic navigation**. It allows you to trigger navigation from within your component's TypeScript code, giving you full control over when and where the user goes.

Proper Code Example with Comments

This example shows a `SenderComponent` that navigates to a `ReceiverComponent` and passes a message using the router's state.

1. The Router Module (`app-routing.module.ts`)

TypeScript

```typescript
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { SenderComponent } from './sender.component';
import { ReceiverComponent } from './receiver.component';

const routes: Routes = [
  { path: 'sender', component: SenderComponent },
  { path: 'receiver', component: ReceiverComponent },
];

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

2. The Sender Component (`sender.component.ts`)

TypeScript

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

@Component({
  selector: 'app-sender',
  template: `
    <h2>Sender Component</h2>
    <p>Click the button to navigate and pass a message in the router state.</p>
    <button (click)="goToReceiver()">Go to Receiver</button>
  `
})
export class SenderComponent {
  constructor(private router: Router) { }

  goToReceiver(): void {
    const message = "Hello from the previous page!";
    // 1. We use router.navigate() and pass a 'state' object.
    this.router.navigate(['/receiver'], { state: { data: message } });
  }
}
```

3. The Receiver Component (`receiver.component.ts`)

TypeScript

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

@Component({
  selector: 'app-receiver',
  template: `
    <h2>Receiver Component</h2>
    <p>Message from previous page: {{ receivedMessage }}</p>
  `
})
export class ReceiverComponent implements OnInit {
  receivedMessage: string = 'No message received.';

  ngOnInit(): void {
    // 2. We access the data from the browser's history.state property.
    // We check if history.state has the 'data' property.
    if (history.state && history.state.data) {
      this.receivedMessage = history.state.data;
    }
  }
}
```

Data Flow

1. The user clicks the button in the SenderComponent.
2. The router.navigate() method is called, which begins the navigation process.
3. The **state** object containing the message is temporarily stored in the browser's history.
4. The ReceiverComponent is loaded.
5. In ngOnInit, the ReceiverComponent accesses history.state to get the message. The message is only available during this initial load; it will be gone if the page is reloaded.

Nested Routes

18 September 2025 16:13

Nested routes allow you to have a component that contains its own set of routes and <router-outlet>. This is how you build a complex layout where a part of the page changes based on the URL, while the rest of the page remains the same.
Think of it like a main dashboard. When you're on the dashboard, the main navigation doesn't change, but a section inside the dashboard changes depending on whether you're viewing your profile, settings, or messages.

What they are	Routes defined inside another route, creating a parent-child relationship.
How they work	The parent component loads into the main <router-outlet>, and the child components load into a second <router-outlet> located inside the parent's template.

Nested Routes Keywords

- **children**: An array property used in a route configuration. It's how you define a **parent-child relationship** between routes. The routes listed inside the children array are loaded as "sub-routes" of the main parent path.
- **Nested <router-outlet>**: A second <router-outlet> placed inside a parent component's template. Angular uses this placeholder to render the components of the child routes, while the parent component stays visible.

Example Code

This example shows how to set up a Dashboard component with two nested child routes: profile and settings.

1. The Routing Module (`app-routing.module.ts`)

This is where you define the parent-child relationship using the `children` property.

TypeScript

```typescript
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { HomeComponent } from './home/home.component';
import { DashboardComponent } from './dashboard/dashboard.component';
import { ProfileComponent } from './profile/profile.component';
import { SettingsComponent } from './settings/settings.component';

const routes: Routes = [
  { path: '', component: HomeComponent },
  // 1. This is the parent route.
  {
    path: 'dashboard',
    component: DashboardComponent,
    // 2. The 'children' array defines the nested routes.
    children: [
      { path: 'profile', component: ProfileComponent }, // e.g., /dashboard/profile
      { path: 'settings', component: SettingsComponent }  // e.g., /dashboard/settin
    ]
  }
];

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

2. The Parent Component (`dashboard.component.ts` and `.html`)

This component acts as the parent container. Its template contains a second `<router-outlet>`.

TypeScript

```typescript
// dashboard.component.ts
import { Component } from '@angular/core';

@Component({
  selector: 'app-dashboard',
  template: `
    <h2>Dashboard</h2>
    <nav>
      <a routerLink="profile">Profile</a> |
      <a routerLink="settings">Settings</a>
    </nav>
    <hr>
    <router-outlet></router-outlet>
  `
})
export class DashboardComponent { }
```

3. The Child Components

These components will be rendered inside the `DashboardComponent`'s `<router-outlet>`.

TypeScript

```typescript
// profile.component.ts
import { Component } from '@angular/core';

@Component({
  selector: 'app-profile',
  template: `<h3>User Profile Page</h3>`
})
export class ProfileComponent { }
```

TypeScript

```typescript
// settings.component.ts
import { Component } from '@angular/core';

@Component({
  selector: 'app-settings',
  template: `<h3>User Settings Page</h3>`
})
export class SettingsComponent { }
```

Data Flow

1. The user navigates to /dashboard.
2. The main <router-outlet> in app.component.html loads the **DashboardComponent**.
3. The user then clicks a link to /dashboard/profile.
4. Angular recognizes this is a nested route and loads the **ProfileComponent** into the **second <router-outlet>** located inside the DashboardComponent's template.
5. If the user navigates to /dashboard/settings, the ProfileComponent is replaced by the SettingsComponent in the second <router-outlet>

AuthGuards

18 September 2025 16:41

Angular Route Guards

Route guards in Angular are security checks that control access to routes based on conditions like user authentication or permissions. They help protect pages by allowing or blocking navigation depending on whether the condition is met.

The Four Main Types

1. **CanActivate**: This is the most common guard. Its primary job is to check if a user is **authorized to enter** a specific route. If the check fails (e.g., the user is not logged in), the guard stops navigation.
2. **CanDeactivate**: This guard determines if a user is allowed to **exit** a route. It's often used to prevent accidental data loss by checking if a form has unsaved changes and prompting the user before navigation proceeds.
3. **CanLoad**: The **CanLoad** guard runs when a user tries to navigate to a **lazy-loaded module**.Its sole job is to check permissions and determine if the **module's data** should be downloaded from the server. If the guard returns false, the code is never fetched, saving bandwidth and maintaining high-level security.

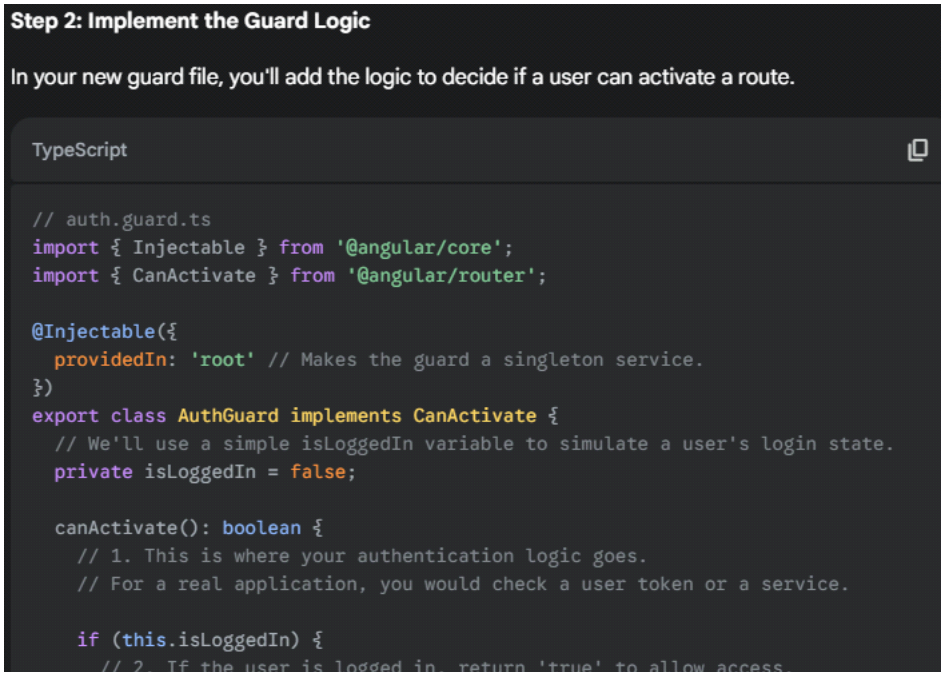
4. **Resolve**: This guard is used for **data pre-fetching**. It ensures that all necessary data for a route is **loaded and available** *before* the component is even initialized. This prevents the user from seeing an empty page or a flickering loading spinner.

Step 1: Generate the Guard

The first step is to generate the guard file using the Angular CLI.

Npx ng g g authGuardName

Step 2: Implement the Guard Logic



```
    // For a real application, you would check a user token or a service.

    if (this.isLoggedIn) {
      // 2. If the user is logged in, return 'true' to allow access.
      console.log('Access granted!');
      return true;
    } else {
      // 3. If the user is NOT logged in, return 'false' to block access.
      console.log('Access denied!');
      // Optional: You would typically redirect the user to a login page here.
      // this.router.navigate(['/login']);
      return false;
    }
  }
}
```

Step 3: Apply the Guard to a Route

Step 3: Apply the Guard to a Route

Finally, you add the guard to any route you want to protect using the `canActivate` property.

TypeScript

```
// app-routing.module.ts
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { DashboardComponent } from './dashboard/dashboard.component';
import { AuthGuard } from './auth.guard'; // 1. Import the AuthGuard

const routes: Routes = [
  // 2. We protect the dashboard route with our AuthGuard.
  {
    path: 'dashboard',
    component: DashboardComponent,
    canActivate: [AuthGuard] // Use the canActivate property
  },
  { path: 'login', component: LoginComponent }
];

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

Data Flow

1. A user attempts to navigate to the /dashboard route.
2. Angular's **Router** sees the canActivate: [AuthGuard] property.
3. The Router calls the canActivate() method in the **AuthGuard** to get permission.
4. The AuthGuard runs its logic (e.g., checks if a user is logged in).
5. If canActivate() returns **true**, the Router allows navigation, and the DashboardComponent is displayed.
6. If canActivate() returns **false**, the Router immediately blocks the navigation, and the user stays on the current page.

Resolve Auth Guard

18 September 2025 18:50

Resolve is a type of route guard that's all about **pre-fetching data before a component is loaded**. Instead of loading the component and then fetching the data, Resolve ensures the data is available *before* the component even initializes. This prevents the user from seeing an empty page or a page that flashes with content as data arrives.

Resolve Guard Notes

- **Purpose:** To fetch data for a route before the component is loaded. This ensures a component has all its required data from the start, avoiding a "blank screen" effect.
- **How it Works:** You create a service that implements the **Resolve** interface. This service has a single method, **resolve()**, which typically returns an **Observable** (like from an HTTP call). The Angular Router waits for this Observable to complete, then passes the fetched data to the component.
- **Data Flow:** The data is retrieved by the Resolve guard and then passed to the component via the **ActivatedRoute** service. You can access the data using the route.data property.
- **Keywords:** Resolve, resolve(), Observable, ActivatedRoute, route.data.

Keywords definitions:

- **Resolve**: This is an **interface** that a service implements. It tells Angular that this service is designed to pre-fetch data for a route.
- **resolve()**: This is the required **method** of the Resolve interface. You write your data-fetching logic inside this method, and it must return an **Observable**.
- **Observable**: A powerful concept from the RxJS library. It represents a **data stream** that can emit values over time. The Angular Router waits for this data stream to complete before activating the component.
- **ActivatedRoute**: A crucial **service** that provides information about the active route. You inject it into a component to access data from the URL, like parameters and, in this case, data from a resolver.
- **route.data**: This is a **property** on the ActivatedRoute that holds the data returned by the resolve() method. You access the pre-fetched data here, using the key you defined in your routing configuration.

Resolve Guard Example

This example shows a ProductResolver that fetches product details before the ProductDetailComponent is displayed.

1. Create the Resolver (`product-resolver.service.ts`)

TypeScript

```typescript
import { Injectable } from '@angular/core';
import { Resolve, ActivatedRouteSnapshot } from '@angular/router';
import { Observable, of } from 'rxjs';
import { ProductService } from './product.service'; // A service to get product data

@Injectable({
  providedIn: 'root'
})
export class ProductResolver implements Resolve<any> {
  constructor(private productService: ProductService) {}

  // The resolve() method must return an Observable, Promise, or a value.
  resolve(route: ActivatedRouteSnapshot): Observable<any> {
    const productId = route.paramMap.get('id');

    // We call our service to get the product details.
    // The router will wait for this Observable to complete before activating the ro
    return this.productService.getProductDetails(productId);
  }
}
```

2. Update the Route Configuration (`app-routing.module.ts`)

TypeScript

```typescript
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { ProductDetailComponent } from './product-detail/product-detail.component';
import { ProductResolver } from './product-resolver.service';

const routes: Routes = [
  {
    path: 'product/:id',
    component: ProductDetailComponent,
    // Add the 'resolve' property, which points to our resolver service.
    resolve: { product: ProductResolver }
    // 'product' is the key we'll use to access the data later.
  }
];

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

3. Access the Data in the Component (`product-detail.component.ts`)

TypeScript

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

@Component({
  selector: 'app-product-detail',
  template: `
    <h3>Product Details</h3>
    <p>Product ID: {{ product.id }}</p>
    <p>Product Name: {{ product.name }}</p>
  `
})
export class ProductDetailComponent implements OnInit {
  product: any;

  constructor(private route: ActivatedRoute) { }

  ngOnInit(): void {
    // The data is available in the 'data' property of the ActivatedRoute.
    // The key 'product' matches the key we used in the route configuration.
    this.product = this.route.snapshot.data['product'];
  }
}
```

Router events and Navigation Cycle.

18 September 2025 17:15

Angular's router navigation cycle includes a series of events that occur during route changes. Each event represents a specific phase in the navigation process.
Here's a brief explanation of each key router event in the order they occur:

🔄 Angular Router Events (Navigation Cycle)

- **NavigationStart**
  Triggered when navigation begins. It contains the target URL and signals the start of routing.
- **RoutesRecognized**
  Fired after the router matches the URL to a route. It includes route configuration and parameters.
- **GuardsCheckStart**
  Begins the evaluation of route guards like CanActivate and CanDeactivate.
- **GuardsCheckEnd**
  Indicates the completion of guard checks. It shows whether navigation is allowed or blocked.
- **ResolveStart**
  Starts the data resolution phase. Resolvers begin fetching required data for the route.
- **ResolveEnd**
  Marks the end of data resolution. All necessary data is now available for the route.
- **NavigationEnd**
  Final event when navigation completes successfully. The URL is updated and the new view is rendered.
- **NavigationCancel**
  Occurs if navigation is canceled—often due to a guard returning false.
- **NavigationError**
  Triggered when navigation fails due to errors like invalid routes or resolver issues.
- **NavigationSkipped**
  Happens when navigation is skipped, such as when navigating to the same URL again.

Component Selector VS Routing Directive

18 September 2025 18:46

An <app-feedback> tag is a **component selector**, while a <router-outlet> is a **routing directive**.
The main difference is that one is **static** and the other is **dynamic**.

- A <app-feedback> selector is an **identifier** that you place in your HTML to load the FeedbackComponent directly and permanently. It always displays the same component.
- A <router-outlet> is a **dynamic placeholder**. It doesn't display any specific component on its own. Instead, it waits for the **Angular Router** to tell it which component to load based on the current URL. It can display different components at different times.

Navigation Routes and events

18 September 2025 19:03

Navigation Routes

A **route** is a specific URL path that maps to a component. It tells Angular which component to display for a given URL.

- **Example:** In a URL like mywebsite.com/products/123, the route is /products/123. Angular sees this and knows to display the ProductDetailsComponent.

Navigation Events

Navigation events are a series of actions that happen as a user moves from one route to another. The Angular **Router** emits these events in a specific order, which you can listen for to perform actions.

- **Example:** When a user clicks a link, the router emits an NavigationStart event, then a RoutesRecognized event, and finally a NavigationEnd event once the new component is loaded.

Lazy Loading

27 September 2025 00:29

Lazy Loading is a design pattern used in Angular routing to load parts of your application **on demand**, rather than all at once when the application starts.

Core Concept

Instead of loading all the code for every route, Angular only downloads the JavaScript code for a specific feature **just before the user navigates to that route**.

How It Works in Routing

1. **Code Splitting:** You separate your application into distinct, self-contained feature modules (e.g., an AdminModule or a DashboardModule).
2. **loadChildren:** In your main routing configuration, you use the loadChildren property instead of the component property. This tells Angular: "Don't load this module yet; wait until the user hits this path."

```
const routes: Routes = [
  {
    path: 'admin',
    loadChildren: () => import('./admin/admin.module').then(m => m.AdminModule)
  }
];
```

RxJs - Reactive Extension for Java Script

18 September 2025 19:04

RxJS, or Reactive Extensions for JavaScript, is a library for **reactive programming** using **Observables**. It provides a powerful way to handle asynchronous data and events, such as API calls, user input, and real-time data streams. The core idea is to treat everything as a stream of data that can be observed.
Here are the four key concepts that make it work:

Observable

An **Observable** is a stream of data that can be used to handle asynchronous events and it does nothing until something **subscribes** to it.
When the Observable gets data (for example, after an HTTP request is complete), it starts **emitting** that data. Any component or element that has subscribed to the Observable is an **Observer**.
The Observable then automatically pushes the data to its Observers. This process calls the Observer's next() method, which automatically receives the data.
This creates a one-way, reactive flow where a component doesn't have to ask for data; it simply gets it when it becomes available.

- **Key Concepts**:
  - **Laziness**: An Observable won't run its code until something subscribes to it.
  - **Asynchronous**: It handles operations that don't happen immediately, like network requests or user clicks.

Observer & Subscription

An **Observer** is an object with three methods that define how to handle the data from an Observable:

1. **next()**: Called for every new value the Observable emits. This is where you get the actual data.
2. **error()**: Called if an error occurs. The Observable stream then stops.
3. **complete()**: Called when the Observable has finished its job and won't emit any more values. The stream then stops.

When you call an Observable's **subscribe()** method, you are creating a **Subscription**. The Subscription is a crucial link that connects the Observer to the Observable, causing the Observable to start emitting data.

- **Operators**: Pure functions that enable you to transform, filter, and combine data streams. They allow you to manipulate the data without altering the original source. A common example is the map operator, which transforms each item in the stream.
- Reactive Programming : Reactive programming is a programming paradigm that works with asynchronous data streams. It's about reacting to changes and events that occur over time, rather than just processing data sequentially.

How to Describe the Data Flow

You can explain the data flow like this:
"A component first **subscribes** to an Observable. This creates a Subscription and signals to the Observable to begin emitting data. The Observable then **emits** values as they become available. The Observer receives these values through its next method. Finally, when the data stream is complete, the complete method is called. This entire process is controlled by **operators** which can manipulate the data at any point in the stream."

CRUD Operations

18 September 2025 19:33

CRUD operations (Create, Read, Update, Delete) in Angular are handled by the **HttpClient** service, which is built on **RxJS**. Each HTTP method you call (GET, POST, PUT, DELETE) returns an **Observable**. You then subscribe to this Observable to handle the backend response.

Step 1: The Service

First, you'll create a service to handle all your API calls. This is the professional way to separate your data-fetching logic from your components.

```
// data.service.ts
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';

@Injectable({
  providedIn: 'root',
})
export class DataService {
  private apiUrl = 'https://jsonplaceholder.typicode.com/posts'; // A public API

  constructor(private http: HttpClient) {}

  // **R**ead: Fetch all items
  getItems(): Observable<any> {
    // The .get() method returns an Observable.
    return this.http.get(this.apiUrl);
  }

  // **C**reate: Add a new item
  createItem(item: any): Observable<any> {
    // The .post() method returns an Observable. You pass the URL and the data.
    return this.http.post(this.apiUrl, item);
  }

  // **U**pdate: Change an existing item
  updateItem(id: number, item: any): Observable<any> {
    // The .put() method returns an Observable. You pass the ID in the URL.
    return this.http.put(`${this.apiUrl}/${id}`, item);
  }

  // **D**elete: Remove an item
  deleteItem(id: number): Observable<any> {
    // The .delete() method returns an Observable. You pass the ID in the URL.
    return this.http.delete(`${this.apiUrl}/${id}`);
  }
}
```

Step 2: The Component (Using the Service)

This is where you'll call the service's methods. Remember to always **subscribe()** to the returned Observable to trigger the HTTP request and handle the response.

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

@Component({
  selector: 'app-crud',
  template: `
    <h3>CRUD with RxJS</h3>
    <p>Check the console to see the results of each operation.</p>
  `,
})
export class CrudComponent implements OnInit {
  constructor(private dataService: DataService) {}

  ngOnInit(): void {
    // We'll run each operation in order.
    this.readItems();
    this.createItem();
    this.updateItem();
    this.deleteItem();
  }

// Example of a READ operation
readItems(): void {
  this.dataService.getItems().subscribe({
    next: (data) => {
      // The 'data' variable contains the response from the API.
      console.log('Read all items:', data);
    },
    error: (err) => console.error('Error reading items:', err),
    complete: () => console.log('Read operation completed.')
  });
}

// Example of a CREATE operation
createItem(): void {
  const newItem = { title: 'New Angular Post', body: 'This is a test.', userId: 1 };
  this.dataService.createItem(newItem).subscribe({
    next: (response) => {
      console.log('Created new item:', response);
    },
    error: (err) => console.error('Error creating item:', err),
    complete: () => console.log('Create operation completed.')
  });
}
```

```
  // Example of an UPDATE operation
  updateItem(): void {
    const updatedItem = { title: 'Updated Post' };
    this.dataService.updateItem(1, updatedItem).subscribe({
      next: (response) => {
        console.log('Updated item:', response);
      },
      error: (err) => console.error('Error updating item:', err),
      complete: () => console.log('Update operation completed.')
    });
  }

  // Example of a DELETE operation
  deleteItem(): void {
    this.dataService.deleteItem(1).subscribe({
      next: (response) => {
        console.log('Deleted item:', response);
      },
      error: (err) => console.error('Error deleting item:', err),
      complete: () => console.log('Delete operation completed.')
    });
  }
}
```

Data Flow for a READ Operationr

1. The CrudComponent calls the **getItems()** method on the **DataService**.
2. The DataService uses the HttpClient to send a **GET request** to the backend API.
3. The HttpClient returns an **Observable**. The request hasn't been sent yet.
4. Back in the component, calling **subscribe()** on the Observable **triggers the HTTP request**.
5. The backend processes the request and sends a response.
6. The Observable receives the data and passes it to your subscribe block, where you can then use the data.

Observable and Promises

27 September 2025 00:33

Feature	**Promise**	**Observable (RxJS)**
Execution	**Eager (Immediate)** **Means** Executes immediately when created.	**Lazy (On Demand)** **Means** Only executes when a observer **subscribes** to it.
Return	**Single Value** Delivers one resolved value or one error, then completes.	**Multiple Values (Stream)** Can emit zero, one, or many values over time (a stream).
Cancellation	**Cannot be Cancelled** Once started, execution runs to completion.	**Cancellable** Execution can be stopped at any time by calling **.unsubscribe()**.
API	Uses standard methods: **.then()** and **.catch()**.	Uses the rich RxJS library with operators like **map, filter, debounce**, etc.
Consumption	**One-time** with .then(success, failure).	**Continuous** with .subscribe(next, error, complete).
Use Case	Asynchronous operations with a single result (e.g., waiting for an HTTP request response).	Handling a sequence of events over time (e.g., user input, timers, shared data streams).

Error handeling and retry stratergy

27 September 2025 00:43

1. Error Handling (Catching and Reacting to Errors)

Error handling defines what happens **after** an Observable stream fails.

The catchError Operator

The primary RxJS operator for handling errors is catchError. When an error occurs in the Observable stream, catchError intercepts the error and allows you to return a new Observable to keep the stream alive or perform some side effect.

Action	Description	RxJS Operator
Log and Continue	Log the error and return an Observable that completes gracefully, often with a default value (e.g., an empty array).	`catchError(() => of([]))`
Throw New Error	Catch the lower-level error and throw a more domain-specific, informative error (e.g., for user feedback).	`catchError(err => throwError(() => new Error('Custom message')))`

TypeScript

```typescript
import { catchError, of } from 'rxjs';

this.dataService.fetchData().pipe(
  catchError(error => {
    // 1. Log the error for debugging
    console.error('API call failed:', error);

    // 2. Return a new Observable with a safe default value
    //    to prevent the entire stream from terminating.
    return of([]);
  })
).subscribe(data => {
  // Process data or the safe default value ([]); the app continues working.
});
```

2. Retry Strategy (Resilience)

The retry strategy defines what happens **before** an Observable stream is considered failed. It allows you to attempt the operation again if it fails due to a temporary issue (e.g., a brief network glitch).

The `retry` and `retryWhen` Operators

Operator	Usage
`retry(N)`	Retries the source Observable a fixed number of times (`N`) immediately upon failure.
`retryWhen()`	Provides more sophisticated control. It allows you to retry based on a condition, or after a specific delay, which is critical for implementing **Exponential Backoff**.

The simplest approach is a fixed number of retries:

TypeScript

```typescript
import { retry } from 'rxjs';

this.dataService.fetchData().pipe(
  // If the request fails, retry the request up to 3 more times.
  retry(3),
  catchError(error => {
    // This catchError only executes if all 3 retries also fail.
    console.error('Request failed permanently after retries.');
    return of(null);
  })
).subscribe();
```

HTTP interceptor

27 September 2025 00:58

An **HTTP Interceptor** is a feature in Angular that provides a mechanism to **intercept** and handle outgoing HTTP requests and incoming HTTP responses globally.

Simple Definition

It acts as a **middleware layer** between your Angular application and your backend server. Whenever your application sends a request, the interceptor steps in, and whenever the server sends a response, the interceptor steps in again.

Primary Purpose

The main goal is to perform common tasks that apply to *all* or *many* HTTP interactions in a single, centralized place.
Common uses include:

- **Adding Headers:** Automatically attaching authorization tokens (JWTs) to every outgoing request.
- **Error Handling:** Centralizing the logic for handling HTTP errors (like a 401 Unauthorized redirect).
- **Logging:** Logging the timing and details of all network activity.
- **Loading Spinners:** Managing a global "loading..." indicator.

1. Asynchronous Programming Basics

Asynchronous programming is a way of writing code where tasks run independently and don't block the main program flow. Instead of waiting for one task to finish before starting the next, asynchronous code allows multiple operations to happen in the background, improving performance and responsiveness

Why Node.js Uses It:
Node.js is **single-threaded** (it uses one main thread called the **Event Loop**). All time-consuming operations (I/O) must be non-blocking to prevent that single thread from freezing, ensuring the server can handle thousands of simultaneous users efficiently.

2. Callbacks

Callbacks were the original way to handle asynchronous operations in JavaScript. A callback is simply a function passed as an argument to another function, intended to be executed *after* the operation finishes.

- **Pro:** Simple and effective for basic async tasks.
- **Con:** Leads to **"Callback Hell"** when handling sequential async operations, making the code deeply nested and hard to read/maintain.

Code Snippet: Using a Callback:

```
// 1. The main function accepts three arguments, where the third is the callback
function calculate(a, b, callback) {
  // Perform the primary action (e.g., addition)
  let result = a + b;

  // 2. Call the function that was passed in, and give it the result
  callback(result);
}

// 3. The function that we want to run LATER
function displayResult(result) {
  console.log("The result is: " + result);
}

// Call calculate and pass the *reference* to the displayResult function
calculate(5, 10, displayResult);
// Output: The result is: 15
```

3. Promises

Promises were introduced to solve the complexity of callback hell, offering a cleaner, more readable way to handle async results. A Promise represents a value that may be available *now*, *later*, or *never*.
A Promise can be in one of three states:

1. **Pending:** The initial state; the operation is still running.
2. **Resolved:** The operation completed successfully.
3. **Rejected:** The operation failed (an error occurred).

Syntax: .then() and .catch()

Instead of nesting callbacks, Promises allow chaining operations using the .then() method for success and the .catch() method for errors.

Code Snippet: Using Promises

```
// A simple function that returns a Promise object
function checkStatus(isGood) {
  // A Promise requires a function with two parameters: resolve and reject
  return new Promise((resolve, reject) => {

    // Simulate a check that takes time (e.g., an API call)
    setTimeout(() => {

      if (isGood) {
        // 1. Success path: Call resolve() with the result data
        resolve("Status Check OK: Data received successfully.");
      } else {
        // 2. Failure path: Call reject() with an Error object
        reject(new Error("Status Check Failed: Connection timed out."));
      }

    }, 500); // 0.5 second delay
  });
}


// --- CONSUMPTION ---

// Scenario 1: Success (isGood = true)
checkStatus(true)
  .then(successMessage => {
    // This function executes when resolve() is called
    console.log(`✅ SUCCESS: ${successMessage}`);
  })
  .catch(error => {
    // This is skipped
    console.log("This will not run.");
  });

// Scenario 2: Failure (isGood = false)
checkStatus(false)
  .then(successMessage => {
    // This is skipped
    console.log("This will not run.");
  })
  .catch(error => {
    // This function executes when reject() is called
    console.error(`❌ ERROR: ${error.message}`);
  });
```

4. Async/Await Basics

async/await is built on top of Promises. It allows you to write asynchronous code that looks and behaves like synchronous code.

- **async Keyword:** Must be placed before any function definition that will use the await keyword. An async function implicitly returns a Promise, and also makes the function

asynchronous.

- **await Keyword:** Can only be used inside an async function. It pauses the execution of the async function until the Promise it precedes is resolved (or rejected).

Code Snippet: Using Async/Await

Error handling is done using a standard **try...catch** block, just like synchronous code.

```
// Function that returns a Promise (Success/Fail)
function checkStatus(isGood) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (isGood) {
        resolve("Status Check OK: Data received successfully.");
      } else {
        reject(new Error("Status Check Failed: Connection timed out."));
      }
    }, 500);
  });
}

// ---------------------------------------------------
// The ASYNC/AWAIT CONSUMER
// ---------------------------------------------------

// 1. Mark the function as 'async'
async function requestData(isAllowed) {
  console.log("1. Starting request...");

  try {
    // 2. Use 'await' to PAUSE execution until the Promise resolves
    //    If the Promise resolves, the value is assigned to the variable.
    const message = await checkStatus(isAllowed);

    console.log(`2. ✅ Success Message: ${message}`);
    console.log("3. Process complete.");

  } catch (error) {
    // 3. If the Promise REJECTS, execution immediately jumps to the 'catch' block
    console.error(`2. ❌ Error Caught: ${error.message}`);
    console.log("3. Process terminated.");
  }
}

// --- EXECUTION ---

// Run the success path
console.log("--- Test 1 (Success) ---");
requestData(true);

// Run the failure path (If you uncomment this, it will run after the first test fin
/*
console.log("\n--- Test 2 (Failure) ---");
requestData(false);
*/
```

Benefits Over .then().catch()

The core benefit of async/await is that it makes asynchronous code **easy to read and debug**.

1. Looks Synchronous (Better Readability)

- **.then():** The logic is spread out across multiple nested or chained functions, making it

harder to track local variables and the flow.

- **async/await:** The code runs top-to-bottom, just like standard, synchronous JavaScript. You assign the result directly to a variable (let result1 = await ...), which is much clearer than waiting for the value to arrive in a .then() callback.

2. Simple Error Handling (try...catch)

- **.then()/.catch():** Error handling is separate—the errors bubble to the nearest .catch() block at the end of the chain.
- **async/await:** You use the familiar **try...catch** block. If any await call rejects its Promise, execution immediately jumps to the catch block, behaving exactly how you expect synchronous code to handle errors.

State Management

27 September 2025 01:00

Defination :
State management is the process of handling all the dynamic data in an application in a centralized way. It ensures that different parts of the app can access and update shared data consistently, making the app more predictable and easier to maintain.

It is done by two methods -

Technique	Description
Shared Services (Simple)	A simple pattern where a root-level service acts as a centralized data store *(*singleton). It typically uses the RxJS **BehaviorSubject** to hold the current state and broadcast updates reactively to subscribing components.
NgRx (Advanced)	A robust, complex pattern (based on Redux) that enforces a strict, predictable data flow using concepts like **Actions**, **Reducers**, and **Selectors** to manage large-scale application state.

Data sharing by Service

18 September 2025 20:12

A shared service is used for communication between components that don't have a direct parent-child relationship. The service is a **singleton**, meaning there is only one instance of it for the entire application, which makes it a perfect central location for sharing data.
The service uses **Dependency Injection** to be made available to any component that needs it. Inside the service, we use **RxJS** to create a data stream with a **BehaviorSubject.**

BehaviorSubject: This is a special type of **Observable** that has a unique property: it stores the **last emitted value**. When a new component subscribes, it immediately receives this most recent value.This most recent value is given to all the subscribers. This BehaviorSubject acts as a data hub, broadcasting any changes to all components that are subscribed to it.
A component that wants to send data calls a method on the service to update the BehaviorSubject. Any component that wants to receive that data simply **subscribes** to the service's Observable. This creates an efficient, real-time communication channel.

Detailed Code Example

1. The Service (data.service.ts)

This is the core of the data-sharing mechanism. It uses a **BehaviorSubject** to hold and manage the data stream.

```
// data.service.ts
import { Injectable } from '@angular/core';
import { BehaviorSubject, Observable } from 'rxjs';

@Injectable({
  // 'providedIn: root' makes this service a singleton.
  // This means there will be only one instance of the service
  // for the entire application, making it perfect for sharing data.
  providedIn: 'root'
})
export class DataService {

  // A private BehaviorSubject is used to manage the data.
  // It holds the current value and sends new values to all subscribers.
  // It's private so that components cannot directly modify the stream.
  private messageSource = new BehaviorSubject<string>('Default Message');

  // A public Observable that components will subscribe to.
  // We expose it as an Observable, which means components can only listen,
  // they cannot use the .next() method to push data.
  currentMessage$ = this.messageSource.asObservable();

  // A public method for other components to send new data to the service.
  // This is the only way for components to update the shared data.
  changeMessage(message: string): void {
    // The .next() method sends the new 'message' to all subscribers
    // who are currently listening to this data stream.
    this.messageSource.next(message);
  }
}
```

.next() method has a dual role:

1. It **updates the value** held by the BehaviorSubject (e.g., messageSource).
2. It **sends that new value** to all active subscribers who are listening to the Observable stream.

2. The Sender Component (sender.component.ts)

This component is responsible for triggering a change in the shared data.

```
// sender.component.ts
import { Component } from '@angular/core';
import { DataService } from './data.service';

@Component({
  selector: 'app-sender',
  template: `
    <h2>Sender Component</h2>
    <input type="text" #inputMessage>
    <button (click)="sendMessage(inputMessage.value)">Send Message</button>
  `
})
export class SenderComponent {
  // We inject the DataService into the constructor.
  // Angular's Dependency Injection system provides the singleton instance here.
  constructor(private dataService: DataService) { }

  sendMessage(message: string): void {
    // We call the public 'changeMessage' method on the service.
    // This action broadcasts the new data to the entire application.
    this.dataService.changeMessage(message);
  }
}
```

3. The Receiver Component (receiver.component.ts

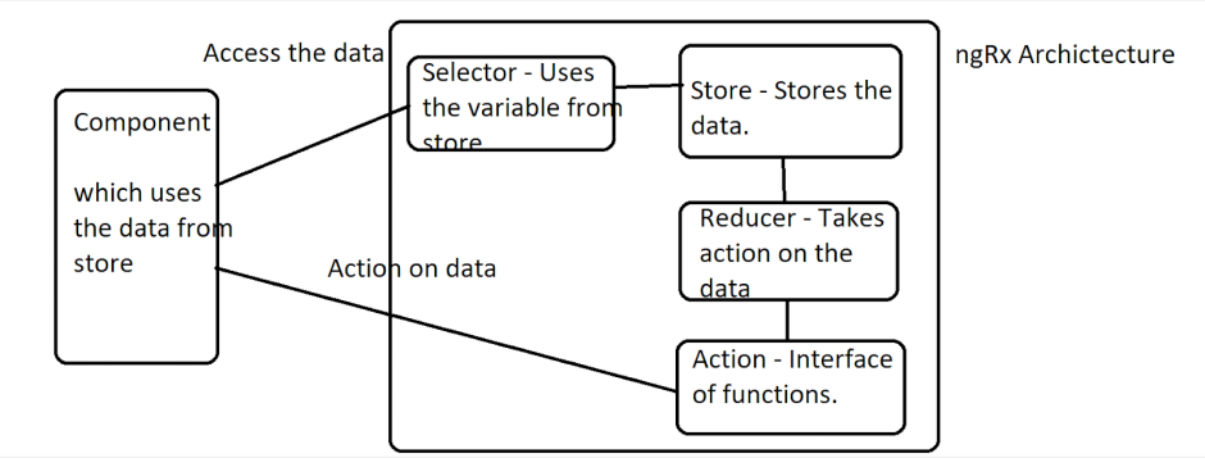
This component listens for and receives the data from the shared service.

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

@Component({
  selector: 'app-receiver',
  template: `
    <h2>Receiver Component</h2>
    <p>Received Message: {{ receivedMessage }}</p>
  `
})
export class ReceiverComponent implements OnInit {
  receivedMessage: string = '';

  constructor(private dataService: DataService) { }

  ngOnInit(): void {
    // This is the most crucial part. We subscribe to the service's Observable.
    // This means our component will listen for and automatically receive
    // any new data that is emitted from the messageSource in the service.
    this.dataService.currentMessage$.subscribe(message => {
      this.receivedMessage = message;
    });
  }
}
```

Data Sharing by ngRx

18 September 2025 23:31

NgRx is a state management library that is used to share the data between different components.
The architecture of NgRx includes:



Store

The **Store** is a single, central container where all the data for your application is stored. The data inside the store is **immutable**, meaning it cannot be changed directly. To update it, you must create a new version of the data.

Action

Actions are **plain objects** that describe *what happened* in the application'. We create actions using create action method from NgRx library.

Reducer

The **Reducer** is a special function that listens for actions. When it hears an action it knows about, it takes the current data from the store and the action, and then returns a **brand new copy** of the data with the changes applied. The reducer is the only thing that is allowed to update the store's data.

Selector

A **Selector** is a function that extracts (or selects) a specific piece of data from the store.

Component

A **Component** is where the work happens. A component's job is to:

- **Dispatch** (send) an **Action** to the store when something happens (like a button click).
- Use a **Selector** to get the data it needs from the store to display on the screen.

The Flow

Here's how it all works together in a simple flow:

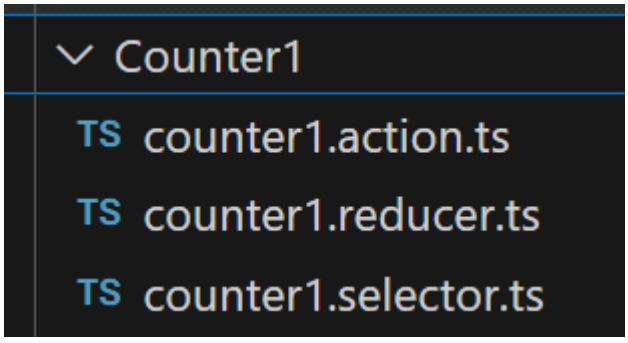
1. You click a button on a **Component**.
2. The component **dispatches** an **Action** (a message like increment_counter).
3. The **Reducer** sees the increment_counter action, takes the current count from the store, adds one to it, and returns a new count.

4. The **Store** replaces its old data with this new data from the reducer.
5. The **Selector** notices the count has changed.
6. The **Component**, which is listening to the selector, gets the new count and updates the screen.

Example :

Install NgRx

```
C:\java\angular\MyProject3>ng add @ngrx/store
Node.js version v23.11.0 detected.
Odd numbered Node.js versions will not enter LTS status and should not be used for production. For more information, ple
ase see https://nodejs.org/en/about/previous-releases/.
Skipping installation: Package already installed
UPDATE src/app/app.module.ts (3056 bytes)
UPDATE package.json (1098 bytes)
√ Packages installed successfully.
```



Action :

src/app/counter/counter.actions.ts

This file defines the **actions**, which are unique messages that get sent to the store.

TypeScript

```typescript
import { createAction } from '@ngrx/store';

// createAction() is a function that creates a new Action creator.
// We give it a descriptive name in brackets, which is its unique type.
// This unique name is a good practice as it helps with debugging.
export const increment = createAction('[Counter Component] Increment');

export const decrement = createAction('[Counter Component] Decrement');

export const reset = createAction('[Counter Component] Reset');
```

Reducer :

src/app/counter/counter.reducer.ts

This file contains the **reducer function**, which is responsible for updating the state based on the actions it receives.

TypeScript

```typescript
import { createReducer, on } from '@ngrx/store';
import { increment, decrement, reset } from './counter.actions';

// This is the starting value of our state. The reducer will use this
// value the very first time an action is dispatched.
export const initialState = 0;

// The createReducer function sets up the rules for how our state can be changed.
// It takes two arguments: the initial state and a set of 'on' functions.
export const counterReducer = createReducer(
  initialState,

  // The 'on' function listens for a specific action.
  // When the 'increment' action is dispatched, this function is called.
  // It takes the current 'state' and returns a brand new state (current state + 1).
  on(increment, (state) => state + 1),

  // This 'on' function listens for the 'decrement' action.
  // It returns a new state that is one less than the current state.
  on(decrement, (state) => state - 1),

  // This 'on' function listens for the 'reset' action.
  // It returns a new state that is 0, regardless of the current state.
  on(reset, (state) => 0)
);
```

In app.module.ts you are defining the key in the store on which reducer will take actions.

```typescript
// counter.module.ts (or feature-level module)

import { NgModule } from '@angular/core';
import { StoreModule } from '@ngrx/store';

// Assume you have imported your counter's reducer function
import { countReducer } from './+state/count.reducer';

// 1. You define the string constant (the KEY)
export const COUNT_FEATURE_KEY = 'counter'; // <-- The unique key for this slice of state

@NgModule({
  imports: [
    // 2. You register the feature using the KEY and the reducer
    StoreModule.forFeature(counter, counterReducer)
  ],
  // ...
})
export class CounterModule {}
```

Selector:

Selector File

A dedicated file for selectors, like `counter.selectors.ts`, is a best practice. It centralizes all the data access logic for a feature.

Here is what that file would look like:

`src/app/counter/counter.selectors.ts`

TypeScript

```typescript
import { createFeatureSelector, createSelector } from '@ngrx/store';
// Define the shape of your state.
import { CounterState } from './counter.reducer';

// createFeatureSelector gets the slice of state called 'counter'.
// This makes your selector type-safe.
export const selectCounterState = createFeatureSelector<CounterState>('counter');

// createSelector is used to get a specific piece of data from the state slice.
// This is more powerful for complex states.
export const selectCount = createSelector(
  selectCounterState,
  (state) => state.count // We can now access properties safely.
);
```

Component:

`src/app/counter/counter.component.ts`

This component is the user interface. It sends messages (**actions**) to the store and reads data from the store via an **Observable**.

TypeScript

```typescript
import { Component } from '@angular/core';
import { Store } from '@ngrx/store';
import { Observable } from 'rxjs';
import { increment, decrement, reset } from './counter.actions';
import { selectCount } from './counter.selectors';

@Component({
  selector: 'app-counter',
  template: `
    <h2>The Count: {{ count$ | async }}</h2>

    <button (click)="increment()">Increment</button>
    <button (click)="decrement()">Decrement</button>
    <button (click)="reset()">Reset</button>
  `
})
export class CounterComponent {
  // We declare an Observable here that will hold the latest count from the store.
  count$: Observable<number>;

  // We inject the Store service into the constructor. Angular's Dependency Injectio
  // provides us with the single, shared instance of the store.
  constructor(private store: Store<{ counter: number }>) {
    // The 'select' method creates a new Observable that listens for changes
    // to a specific part of the store's state. It's how we get data.
    this.count$ = this.store.select(selectCount);
  }
```

```
  // This method is called when the button is clicked.
  increment(): void {
    // The dispatch() method is how we send an Action to the Store.
    // This is the beginning of the NgRx data flow.
    this.store.dispatch(increment());
  }

  decrement(): void {
    this.store.dispatch(decrement());
  }

  reset(): void {
    this.store.dispatch(reset());
  }
}
```

createFeatureSelector vs. store.select

Your instructor is using the recommended approach.
I used store.select('key') in my simple examples because it's a quicker way to demonstrate the concept for basic states (like just a number).

- **store.select('key')**: Simple and works for basic states. It's a quick shortcut.
- **createFeatureSelector**: The **official best practice**. It is more robust because it is **type-safe** and can be used to build more complex selectors for objects with multiple properties. It's the professional standard for real-world projects.

Example in single file

19 September 2025 00:20

```
// src/app/counter.component.ts

import { Component } from '@angular/core';
import { Store, createAction, createReducer, on } from '@ngrx/store';
import { Observable } from 'rxjs';

// **1. Actions**: These are unique messages that describe "what happened."
// They don't have any logic, just a name.
const increment = createAction('[Counter Component] Increment');
const decrement = createAction('[Counter Component] Decrement');
const reset = createAction('[Counter Component] Reset');


// **2. Reducer**: This is a pure function that listens for actions.
// It takes the current state and an action, and returns a *new* state.
const counterReducer = createReducer(
  0, // This is the initial state (the starting value).
  on(increment, (state) => state + 1), // When it sees 'increment', it returns a new state that is +1.
  on(decrement, (state) => state - 1), // When it sees 'decrement', it returns a new state that is -1.
  on(reset, (state) => 0)             // When it sees 'reset', it returns a new state of 0.
);


// **3. Component**: This is where you dispatch actions and read data.
@Component({
  selector: 'app-counter',
  template: `
    <h2>The Store's Data: {{ count$ | async }}</h2>

    <button (click)="increment()">Increment</button>
    <button (click)="decrement()">Decrement</button>
    <button (click)="reset()">Reset</button>
  `,
  // This tells Angular to use the 'counterReducer' for this component's local state.
  // This is for demonstration purposes. In a real app, it would be in the AppModule.
  providers: [{ provide: 'localCounterReducer', useValue: counterReducer }]
})
```

```
export class CounterComponent {

  // **4. Selector**: This is a tool to get data from the Store.
  // We declare an Observable here that will always have the latest 'count' data.
  count$: Observable<number>;

  // **5. Store**: This is injected here. It is the central container for the data.
  // We initialize the selector in the constructor.
  constructor(private store: Store<{ count: number }>) {
    // We use the 'select' operator to pick out the 'count' data from the Store.
    this.count$ = this.store.select('count');
  }

  // **6. Dispatching Actions**: When a button is clicked, we dispatch an Action.
  // This sends a message to the Store.
  increment(): void {
    this.store.dispatch(increment());
  }

  decrement(): void {
    this.store.dispatch(decrement());
  }

  reset(): void {
    this.store.dispatch(reset());
  }
}
```

Reactive Pattern

27 September 2025 01:11

The **Reactive Pattern** in Angular is a software architectural approach centered around the concept of **data streams** and **change propagation**.
It fundamentally shifts the way an application handles data, moving from traditional imperative coding (where you explicitly tell the code *when* to check for changes) to a declarative, **data-driven** approach.

Core Principles

1. **Data as Streams (Observables):** Instead of thinking of data as static variables, the reactive pattern treats data and events (like user clicks, HTTP responses, or form inputs) as continuous, asynchronous streams represented by **Observables**.
2. **Immutability and Purity:** Data is typically kept immutable, and functions (operators) are used to transform the streams without modifying the original data source.
3. **Automatic Change Propagation:** When the source data stream emits a new value, the changes automatically propagate through the application to all components that are **subscribed** to that stream. This is why services use **BehaviorSubject** to alert components to updates.

Where You See It

The reactive pattern is heavily implemented in key Angular features:

- **RxJS:** The library that powers all reactive functionality.
- **Reactive Forms:** Forms are modeled as Observable streams of value changes.
- **HTTP:** HTTP requests return Observables.
- **State Management:** Tools like NgRx and the Shared Service pattern rely on Observables to broadcast state changes.

RxJS Operator

19 September 2025 00:41

RxJS **Operators** are powerful functions that allow you to manage and transform data streams. They act like tools in a workshop, helping you manipulate the values flowing through an **Observable** pipe.
There are two main types of operators:

- **Creation Operators**: Functions that create a new Observable from scratch.
- **Pipeable Operators**: Functions that modify an existing Observable stream, typically used with the .pipe() method.

Key Operators and Their Use Cases

- **map()**: Transforms each value in the stream into a new value.
  - **Use Case:** Converting data from an API into a format your component needs.
- **filter()**: Selects values from the stream that meet a specific condition.
  - **Use Case:** Displaying only products that are in stock.
- **tap()**: Performs a side effect for each value, like logging to the console, without changing the stream.
  - **Use Case:** Debugging a stream to see the values at a certain point.

Example:

```
export class App implements OnInit {

  ngOnInit(): void {

    // Create an Observable that emits a sequence of numbers.
    const source$ = of(1, 2, 3, 4, 5, 6, 7, 8, 9);

    // Use a pipeline to apply a series of operators.
    source$.pipe(

      // The tap() operator lets you perform side effects (like logging)
      // without modifying the data in the stream.
      tap(value => console.log('Tap 1 (initial value):', value)),
      // Output: 1, 2, 3, 4, 5, 6, 7, 8, 9

      // The filter() operator checks a condition for each value.
      // It only lets values that return 'true' pass through to the next operator.
      filter(value => value % 2 === 0),
      // Output: 2, 4, 6, 8

      // The map() operator transforms each value into a new value.
      // Here, we multiply each even number by 10.
      map(value => value * 10),
      // Output: 20, 40, 60, 80
```

```
      // The tap() operator again to see the transformed values.
      tap(value => console.log('Tap 2 (after map and filter):', value)),
      // Output: 20, 40, 60, 80

      // The take() operator takes a specified number of values from the stream
      // and then completes, preventing further values from passing through.
      take(2),
      // Output: 20, 40

    ).subscribe({
      next: value => console.log('Final Result (in subscribe):', value),
      // Final Output:
      // Final Result (in subscribe): 20
      // Final Result (in subscribe): 40

      complete: () => console.log('The Observable stream has completed.')
    });
  }
}
```

Unit Testing (Jasmine & Karma)

19 September 2025 00:55

Unit Testing (Jasmine & Karma) 🧪

Unit testing is the practice of testing small, isolated pieces of code which are called "unit"—to ensure they work as expected. In Angular, a unit is typically a single component, service, or pipe.

Jasmine is the testing framework that provides the syntax for writing tests.

- describe(): Defines a test suite, which is a group of related tests.
- it(): Defines a single test case.
- expect(): The assertion, which checks if a value meets a certain condition.

Karma is the test runner. It opens a browser, runs your tests, and reports the results back to you.

Example: A Simple Service Test

Let's test a simple service that calculates a sum.

in App.compoenent.spec.ts

```typescript
// service-to-test.service.ts
import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'root'
})
export class CalculatorService {
  add(a: number, b: number): number {
    return a + b;
  }
}

// service-to-test.service.spec.ts (The Test File)
import { TestBed } from '@angular/core/testing';
import { CalculatorService } from './service-to-test.service';

describe('CalculatorService', () => {
  let service: CalculatorService;

  beforeEach(() => {
    // TestBed configures a testing module for our test.
    TestBed.configureTestingModule({});
    // We get a new instance of our service before each test.
    service = TestBed.inject(CalculatorService);
  });

  // A test case for the 'add' method.
  it('should be able to add two numbers', () => {
    const sum = service.add(5, 5);
    // The 'expect' statement checks if the result is correct.
    expect(sum).toBe(10);
  });
});
```

When you generate a component with the Angular CLI, the boilerplate code for the test file

(.spec.ts) is automatically created for you. This includes the describe, beforeEach, and TestBed setup.
Your main job as a developer is to write the individual test cases within the **it()** blocks. This is where you write the code to test a specific piece of your component's functionality.

Lazy Loading

19 September 2025 01:18

Lazy Loading 🚀

Lazy loading is a design pattern that loads parts of your application only when a user navigates to them. This dramatically reduces the initial load time of your application, as the browser only downloads the necessary code.

- **How it Works**: You create a separate **feature module** for a section of your app (e.g., a dashboard or an admin panel). In your routing configuration, you use loadChildren instead of component to point to this module.
- **Benefits**: Faster initial load times and improved performance, especially for large applications with many routes.

Example: Lazy Loading a Dashboard Module

1. The Feature Module's Routing (dashboard-routing.module.ts)

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

const routes: Routes = [
  // This is the default route for the lazy-loaded module.
  { path: '', component: DashboardComponent }
];

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

2. The App Module's Routing (app-routing.module.ts)

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

const routes: Routes = [
  // This is the key part: we use loadChildren.
  // The module is only loaded when the user navigates to /dashboard.
  {
    path: 'dashboard',
    loadChildren: () => import('./dashboard/dashboard.module').then(m => m.DashboardModule)
  },
  // Other routes...
];

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

Eager Vs Lazy loading

19 September 2025 01:22

The main difference between normal (eager) loading and lazy loading is **when** the code for a specific route is downloaded by the browser.

- **Normal (Eager) Loading**: The browser downloads all the code for all the routes when the application first starts.
- **Lazy Loading**: The browser only downloads the code for a specific route when the user actually navigates to it.

1. Normal Loading (Eager)

In normal loading, your router directly imports the component and loads it immediately. This is best for small applications.

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { HomeComponent } from './home/home.component';
import { AboutComponent } from './about/about.component';

const routes: Routes = [
  // Both HomeComponent and AboutComponent are loaded as soon as the app starts.
  { path: '', component: HomeComponent },
  { path: 'about', component: AboutComponent }
];

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

2. Lazy Loading

With lazy loading, you point the router to a **module**, and the code for that module is only loaded when the user goes to that specific route. This is ideal for large applications.

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

const routes: Routes = [
  { path: '', redirectTo: '/dashboard', pathMatch: 'full' },
  // This is the key part for lazy loading.
  // The code for the DashboardModule is only downloaded and bundled when the user
  // navigates to the '/dashboard' path.
  {
    path: 'dashboard',
    loadChildren: () => import('./dashboard/dashboard.module').then(m => m.DashboardModule)
  }
];

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

Comparison

Feature	Normal Loading (Eager)	Lazy Loading
When It Loads	All at once, on app startup.	Only when the user navigates to the route.
Configuration	Uses component: MyComponent.	Uses loadChildren: () => import('./module').then(...).
Initial Bundle	Larger, contains all the code.	Smaller, contains only the necessary code for the first page.
Best For	Small applications with few routes.	Large applications with many routes, especially those with admin panels.

Imp topic

19 September 2025 00:51

NgModules help you **organize** your app into logical groups. Think of them as a box for related components and services. For a large app, you'd use many of them to keep things tidy.

- forRoot(): You use this once in your main AppModule to set up something for the **whole application**, like your main routing or a service.
- forChild(): You use this in a **feature module** to set up things that are only for that specific part of your app, like routes for a lazy-loaded section.

State Management without NgRx

This is a simple way to **share data** between any components. You create a special **shared service** that acts as a central data hub.

- This service uses an RxJS **BehaviorSubject** to hold the data. It's a type of **Observable** that broadcasts new data to anyone listening.
- Components that need the data simply **subscribe** to the service.
- Components that change the data call a method on the service, which then uses .next() to update the BehaviorSubject.

Angular Universal

This is a performance tool for your Angular app. It lets you run your app on a server first, creating a static HTML version of the page.

- The user's browser gets this **static page instantly**, so they see something right away.
- Then, the full Angular app loads in the background and takes over.
- This is called **Server-Side Rendering (SSR)**.
- It improves your app's load time and helps with **SEO** (Search Engine Optimization), so search engines like Google can find and index your content.