**Angular 19 Features**

**Basic Concepts**

**1.** [Angular CLI](#angularcli)

**2.** [Component-Based Architecture](#componentbasedarchitecture)

**3.** [String Interpolation](#stringinterpolation)

**4.** [Event Binding](#eventhandling)

**5.** [Two-Way Data Binding](#twodatabinding)

**6.** [Directives](#directives)

**7.** [Pipes](#pipes)

**8.** [Forms Handling](#formshandling)

**9.** [Routing and Navigation](#routingandnavigation)

**10.** [HttpClient for API Integration](#httpclient)

**11.** [Lifecycle Hooks](#lifecyclehooks)

**12.** [Dependency Injection (DI)](#dependencyinjection)

**13.** [Custom Directives](#customdirectives)

**14.** [Custom Pipes](#custompipes)

**15.** [Modular Architecture](#modulararchitecture)

**Intermediate Concepts**

**16.** [Angular Material](#angularmaterial)

**17.** [Component Communication](#componentcomunication)

**18.** [Host Binding and Host Listener](#hostbindingandlistener)

**19.** [Preloading Modules](#preloadingmodules)

**20.** [Lazy Loading Modules](#lazyloadingmodules)

**21.** [Content Projection](#contentprojection)

**22.** [Dynamic Forms](#dynamicforms)

**23.** [Resolver in Routing](#resolverrouting)

**24.** [Environment Configuration](#environmentconfiguration)

**25.** [Testing](#testing)

**26.** [Unit Testing with Jasmine and Karma](#unittesting)

**27.** [Error Handling with Interceptors](#errorhandlingwithinterceptors)

**28.** [Advanced Dependency Injection](#advancedependencyinjection)

**29.** [Custom Validators](#customvalidator)

**30.** [Change Detection](#changedetection)

**31.** [State Management with Services](#statemanagementwithservices)

**32.** [Animation Support](#animation)

**33.** [Internationalization (i18n)](#internationalization)

**Advanced Concepts**

**34.** [Angular Universal (Server-Side Rendering)](#serversiderendering)

**35.** [State Management with NgRx](#statemanagementngrx)

**36.** [Progressive Web Apps (PWA)](#progressivewebapps)

**37.** [Service Worker Integration](#serviceworkerintegration)

**38.** [Web Worker Integration](#webworker)

**39.** [Signals](#signals)

**40.** [Dynamic Component Loading](#dynamiccomponentloading)

**41.** [Internationalization (i18n) with Runtime Translations](#internationalizationwithruntimetranslati)

**42.** Angular Material Theming and Customization

**43.** [Platform Agnostic Framework](#platformagnostic)

**44.** [Custom Elements (Web Components)](#webcomponents)

**45.** [Reactive Programming with RxJS](#reactiveprogrammingwithrxjs)

**46.** [Standalone Components](#standalonecomponent)

**1. Angular CLI Schematics**

[Go to Index](#Index)

Angular CLI (Command Line Interface) Schematics are a set of tools and templates that help automate repetitive coding tasks in Angular projects. They simplify and standardize the creation and configuration of components, services, modules, and even the integration of third-party libraries.

**Why Use Angular CLI Schematics?**

* Save time by automating code generation.
* Ensure consistency in the project's structure and code style.
* Reduce errors by using predefined templates and configurations.
* Simplify the integration of libraries and frameworks.

**Common Commands and Use Cases**

1. **Generate a Component**  
   Creates a new Angular component with the necessary files (HTML, CSS, TS, and spec files) and updates the appropriate module.

**Command**:

**ng generate component my-component**

**Options**:

* + --inline-style: Use inline CSS instead of a separate .css file.
  + --inline-template: Use an inline template instead of a separate .html file.
  + --skip-tests: Skip generating the spec file for tests.

**Example**:

**ng generate component login --inline-style --skip-tests**

1. **Generate a Service**  
   Creates a new Angular service and its spec file for testing. Services are used to handle business logic and data sharing.

**Command**:

**ng generate service my-service**

**Options**:

* + --skip-tests: Avoid generating a test file for the service.

**Example**:

**ng generate service auth/auth-guard --skip-tests**

1. **Add a Third-Party Library**  
   Easily integrate third-party libraries, such as Angular Material, using schematics provided by the library.

**Command**:

**ng add @angular/material**

This command:

* + Installs the @angular/material package.
  + Updates the angular.json file with necessary styles and configurations.
  + Provides an interactive setup for configuring themes and animations.

1. **Generate a Module**  
   Creates a new Angular module, allowing you to organize your app into logical units.

**Command**:

**ng generate module my-module**

**Options**:

* + --route: Create a lazy-loaded module and configure the router.
  + --module: Specify the parent module where the new module should be imported.

**Example**:

**ng generate module features/user --route user --module**

**app.module**

1. **Generate a Directive or Pipe**  
   Simplifies creating reusable directives and pipes.

**Command**:

**ng generate directive my-directive**

**ng generate pipe my-pipe**

**Example**:

**ng generate directive highlight --skip-tests**

**ng generate pipe currency-format --skip-tests**

**How It Works**

Schematics use a JSON-based configuration to define:

* Templates for generating files.
* Rules for updating existing files (e.g., adding imports to a module).
* Commands for installing dependencies or running scripts.

**Custom Schematics**

In addition to the default schematics, you can create your own custom schematics tailored to your project's needs.

1. Install the Schematics CLI:

**npm install -g @angular-devkit/schematics-cli**

1. Create a new schematic:

**schematics blank --name=my-schematic**

1. Customize the schematic logic and templates.

**Best Practices**

* Use meaningful names for components, services, and modules to reflect their purpose.
* Take advantage of options to minimize unnecessary boilerplate (e.g., --skip-tests for quick prototyping).
* Regularly update Angular CLI to leverage the latest schematics features.

By mastering Angular CLI Schematics, you can streamline your development pocess, ensure consistency across your project, and focus on solving business problems rather than managing boilerplate code.

**2. Component-Based Architecture**

[Go to Index](#Index)

Angular apps are built using **components**. A component controls a part of the UI.

**Code Example:**

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

**@Component**({

selector: 'app-hello',

template: `<h1>{{ title }}</h1>`,

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

})

export class HelloComponent {

title = 'Hello, Angular!';

}

**Explanation:**

* **@Component** decorator defines metadata for the component.
* **selector** specifies the custom HTML tag <app-hello> to render this component.
* **template** defines the HTML structure.
* **title** is a **property** bound to the template.

**3. String Interpolation**

[Go to Index](#Index)

**String Interpolation** in Angular allows you to embed **dynamic data** (from the component’s class properties) directly into the HTML template using **double curly braces** {{ }}. It's a one-way data-binding mechanism where data flows from the **component** to the **view**.

**Syntax**

{{ expression }}

Here:

* **expression**: Any valid TypeScript expression that Angular evaluates and converts to a string.

**Sample**

**Component:**

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

@Component({

selector: 'app-string-interpolation',

template: `

<h1>Welcome, {{ username }}!</h1>

<p>Today is {{ today }}</p>

`

})

export class StringInterpolationComponent {

username: string = 'John Doe';

today: string = new Date().toDateString();

}

**Output:**

Welcome, John Doe!

Today is Wed Jan 15 2025

**Features of String Interpolation**

1. **Property Binding** You can bind a property of the component to the view.

<p>Your name is: {{ name }}</p>

1. **Expression Evaluation** Angular evaluates expressions and renders the result.

<p>Result of 10 + 20 = {{ 10 + 20 }}</p>

**Output:**

Result of 10 + 20 = 30

1. **Method Invocation** You can call methods from the component in the template.

greet() {

return 'Hello, Angular!';

}

<p>{{ greet() }}</p>

1. **Using JavaScript Operators** Use operators like +, \*, ?, and more within {{ }}.

<p>Discount: {{ price > 100 ? '10%' : '5%' }}</p>

1. **Accessing Object Properties** You can directly access object properties.

user = { firstName: 'Jane', lastName: 'Doe' };

<p>{{ user.firstName }} {{ user.lastName }}</p>

**Best Practices**

1. Keep expressions simple and avoid complex logic. **Bad:**

{{ items.reduce((acc, item) => acc + item.price, 0) }}

**Good:** Compute in the component:

totalPrice = this.items.reduce((acc, item) => acc + item.price, 0);

Bind in the template:

{{ totalPrice }}

1. Use **string interpolation** only for displaying data. Use **property binding** for assigning values to element properties.

**Example of Property Binding:**

<input [value]="username">

**Example of Interpolation:**

<p>{{ username }}</p>

**Difference Between Interpolation and Property Binding**

| **Feature** | **String Interpolation** | **Property Binding** |
| --- | --- | --- |
| **Syntax** | {{ expression }} | [property]="expression" |
| **Use Case** | Binding data as text content | Binding properties or attributes directly |
| **Example** | <p>{{ username }}</p> | <input [value]="username"> |

**4. Event Binding**

[Go to Index](#Index)

Bind events like clicks, keypresses, etc., directly to methods.

**Code Example:**

<button **(click)**="onClick()">Click Me</button>

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

@Component({

selector: 'app-event-binding',

templateUrl: './event-binding.component.html'

})

export class EventBindingComponent {

message = 'Hello!';

onClick() {

this.message = 'Button Clicked!';

}

}

**Explanation:**

* Use **(event)** syntax to bind methods to events like click, mouseover, etc.

Event binding in Angular allows you to listen to and respond to user actions in the DOM. The syntax for event binding in Angular is:

<element (event)="expression"></element>

Here's a detailed list of event bindings for various HTML elements:

**General Event Binding**

**Common HTML events supported in Angular:**

| **Event** | **Description** |
| --- | --- |
| **click** | Triggered when the element is clicked. |
| **dblclick** | Triggered when the element is double-clicked. |
| **mousedown** | Triggered when a mouse button is pressed. |
| **mouseup** | Triggered when a mouse button is released. |
| **mousemove** | Triggered when the mouse is moved over the element. |
| **mouseenter** | Triggered when the mouse pointer enters the element. |
| **mouseleave** | Triggered when the mouse pointer leaves the element. |
| **contextmenu** | Triggered when the right mouse button is clicked. |
| **keydown** | Triggered when a key is pressed. |
| **keyup** | Triggered when a key is released. |
| **keypress** | Triggered when a key is pressed and released. |
| **input** | Triggered when the value of an <input> or <textarea> changes. |
| **focus** | Triggered when the element gains focus. |
| **blur** | Triggered when the element loses focus. |
| **change** | Triggered when the value of an input element changes and loses focus. |
| **submit** | Triggered when a form is submitted. |

**Form Element Events**

**Input, Textarea, and Select Elements:**

| **Event** | **Description** |
| --- | --- |
| **input** | Triggered when the user types or modifies input. |
| **focus** | Triggered when the element receives focus. |
| **blur** | Triggered when the element loses focus. |
| **change** | Triggered when the value of the element changes. |
| **select** | Triggered when text is selected inside the element. |

**Mouse Events**

**Buttons, Divs, and Other Clickable Elements:**

| **Event** | **Description** |
| --- | --- |
| click | Triggered on a mouse click. |
| dblclick | Triggered on a double mouse click. |
| mousedown | Triggered when the mouse button is pressed. |
| mouseup | Triggered when the mouse button is released. |
| mouseenter | Triggered when the pointer enters the element. |
| mouseleave | Triggered when the pointer leaves the element. |
| mouseover | Triggered when the pointer is over the element. |
| mouseout | Triggered when the pointer moves out of the element. |
| mousemove | Triggered when the pointer moves inside the element. |

**Keyboard Events**

**Applied to Input Elements or General Elements:**

| **Event** | **Description** |
| --- | --- |
| keydown | Triggered when a key is pressed down. |
| keypress | Triggered when a key is pressed and held. |
| keyup | Triggered when a key is released. |

**Drag and Drop Events**

**Drag-Enabled Elements:**

| **Event** | **Description** |
| --- | --- |
| drag | Triggered when an element is being dragged. |
| dragstart | Triggered when dragging starts. |
| dragend | Triggered when dragging ends. |
| dragover | Triggered when a dragged item is over a drop target. |
| dragenter | Triggered when the dragged item enters a target. |
| dragleave | Triggered when the dragged item leaves a target. |
| drop | Triggered when the dragged item is dropped. |

**Touch Events**

**For Mobile and Touchscreen Devices:**

| **Event** | **Description** |
| --- | --- |
| touchstart | Triggered when a touch starts. |
| touchmove | Triggered when a touch moves. |
| touchend | Triggered when a touch ends. |
| touchcancel | Triggered when a touch is canceled. |

**Media Events**

**Audio, Video, and Media Elements:**

| **Event** | **Description** |
| --- | --- |
| play | Triggered when playback starts. |
| pause | Triggered when playback is paused. |
| ended | Triggered when playback ends. |
| volumechange | Triggered when the volume changes. |
| timeupdate | Triggered when the playback position changes. |
| seeking | Triggered when seeking starts. |
| seeked | Triggered when seeking ends. |
| loadedmetadata | Triggered when metadata is loaded. |

**Window and Document Events**

**Global Scope Events:**

| **Event** | **Description** |
| --- | --- |
| resize | Triggered when the window is resized. |
| scroll | Triggered when the window or element is scrolled. |
| load | Triggered when the window or element is fully loaded. |
| unload | Triggered when the user leaves the page. |
| error | Triggered on error loading a resource. |
| beforeunload | Triggered before the user leaves the page. |

**Example Usage in Angular:**

<!-- Button Click -->

<**button** **(click)**="onButtonClick()">Click Me</button>

<!-- Input Change -->

<**input** **(input)**="onInputChange($event)" placeholder="Type here" />

<!-- Keydown Event -->

<**input** (keydown)="onKeydown($event)" placeholder="Press a key" />

<!-- Mouse Enter Event -->

<**div** **(mouseenter)**="onMouseEnter()">Hover over me</div>

**5. Two-Way Data Binding**

[Go to Index](#Index)

Angular supports two-way binding, syncing the data between the model and the view.

**Code Example:**

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

<p>Hello, {{ name }}!</p>

**Explanation:**

* The **[(ngModel)]** directive binds the name property to the input field, reflecting changes both ways.

Two-way data binding is a powerful feature of Angular that synchronizes the data between the model (component) and the view (template). This means any changes in the view are immediately reflected in the model, and vice versa. It ensures consistency between the user interface and the application's data.

**How Two-Way Binding Works**

Angular's two-way binding is achieved using the [(ngModel)] directive. This directive combines:

1. **Property Binding ([value]="expression"):** Sets the value of an element from the component's property.
2. **Event Binding ((input)="expression = $event.target.value"):** Updates the component's property when the user interacts with the element.

The [(ngModel)] directive simplifies this process into a single syntax.

**Code Example**

Here’s a simple example of two-way data binding:

<!-- Template -->

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

<p>Hello, {{ name }}!</p>

**Explanation:**

1. **[(ngModel)]:**
   * Binds the name property in the component to the input field.
   * Any change to the input field updates the name property in the component.
   * Any change to the name property in the component updates the value in the input field.
2. **Dynamic Update:**
   * As the user types in the input field, the value of name is updated dynamically.
   * The paragraph (<p>) immediately reflects the updated value using interpolation ({{ name }}).

**Component Setup**

For the above code to work, the Angular component should include the following:

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

@Component({

selector: 'app-root',

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

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

})

export class AppComponent {

name: string = ''; // Model property bound to the view

}

**Key Points to Remember**

1. **FormsModule Requirement:** To use ngModel, you need to import the FormsModule in your Angular module:

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

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

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

@NgModule({

declarations: [AppComponent],

imports: [

BrowserModule,

**FormsModule** // Include FormsModule

],

bootstrap: [AppComponent]

})

export class AppModule {}

1. **Bidirectional Flow:**
   * The data flows in two directions:
     + From the component to the view (**property** **binding**).
     + From the view to the component (**event** **binding**).
2. **Use Cases:**
   * **Form Inputs:** Sync form controls (e.g., text fields, checkboxes) with the model.
   * **Real-Time Feedback:** Provide dynamic updates in the UI based on user input.
3. **Scoped to Template-Driven Forms:**
   * **[(ngModel)]** is mainly used in template-driven forms. For reactive forms, Angular uses a different approach.

**Advantages of Two-Way Data Binding**

* **Simplified Code:** Reduces the need for manually wiring data between the view and model.
* **Real-Time Updates:** Creates dynamic user interfaces with minimal effort.
* **Enhanced Productivity:** Speeds up development for interactive applications.

**6. Directives**

[Go to Index](#Index)

Directives add behavior to elements.

* **Structural Directive (e.g., \*ngIf, \*ngFor)**:

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

<ul>

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

</ul>

* **Attribute Directive (e.g., ngClass)**:

<div **[ngClass]**="{ 'active': isActive }">Styled Div</div>

Directives are a core feature in Angular that allow you to add behavior, manipulate the DOM, and dynamically control the appearance or structure of elements in your application. Angular provides three types of directives:

**Structural Directives**

**Attribute Directives**

**Custom Directives**

**Structural Directives**

Structural directives are used to change or manipulate the DOM structure by adding, removing, or altering elements dynamically.

**Common Structural Directives:**

* **\*ngIf:** Conditionally includes or excludes elements in the DOM.
* **\*ngFor:** Iterates over a collection and creates a template for each item.
* **\*ngSwitch:** Displays one element from a set of elements based on a condition.

**Examples:**

**\*ngIf Example:**

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

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

* The \*ngIf directive evaluates the isVisible property and includes or removes the <div> from the DOM based on its truthy or falsy value.

**\*ngFor Example:**

<ul>

<li \*ngFor="let item of items; let i = index">Item {{ i + 1 }}: {{ item }}</li>

</ul>

* The \*ngFor directive iterates over the items array, creating a <li> element for each item.
* Additional properties like index, first, last, and even can be used to get contextual information about the iteration.

**\*ngSwitch Example:**

<div [ngSwitch]="viewMode">

<p \*ngSwitchCase="'list'">List View</p>

<p \*ngSwitchCase="'grid'">Grid View</p>

<p \*ngSwitchDefault>Default View</p>

</div>

* Displays different content based on the value of the viewMode property.

**Attribute Directives**

Attribute directives are used to change the appearance, behavior, or style of an element. Unlike structural directives, they do not change the DOM structure but modify the element they are applied to.

**Common Attribute Directives:**

* **ngClass:** Dynamically adds or removes CSS classes.
* **ngStyle:** Dynamically sets inline styles on an element.
* **[disabled]:** Enables or disables an element dynamically.

**ngClass Example:**

<div [ngClass]="{ 'active': isActive, 'inactive': !isActive }">

Styled Div

</div>

* Adds the active class if isActive is true and the inactive class if isActive is false.

**ngStyle Example:**

<p [ngStyle]="{ 'color': isError ? 'red' : 'green', 'font-size': fontSize + 'px' }">

Dynamic Styles

</p>

* Dynamically sets the color and font-size of the paragraph based on component properties.

**Dynamic Disabling with [disabled]:**

<button [disabled]="isDisabled">Submit</button>

* Disables the button if isDisabled is true.

**Custom Directives**

You can create your own directives to encapsulate reusable behaviors.

**Creating a Custom Directive:**

import { **Directive**, **ElementRef**, **Renderer2**, **HostListener** } from '@angular/core';

**@Directive**({

selector: '[appHighlight]'

})

export class HighlightDirective {

constructor(private el: **ElementRef**, private renderer: **Renderer2**) {}

**@HostListener**('mouseenter') onMouseEnter() {

this.renderer.setStyle(this.el.nativeElement, 'background-color', 'yellow');

}

**@HostListener**('mouseleave') onMouseLeave() {

this.renderer.removeStyle(this.el.nativeElement, 'background-color');

}

}

**Usage:**

<p appHighlight>Hover over me to see the highlight effect!</p>

**Key Points to Remember**

1. **Structural Directives:**
   * Prefixed with \* (e.g., \*ngIf, \*ngFor).
   * Modify the DOM structure by adding or removing elements.
2. **Attribute Directives:**
   * Modify the appearance or behavior of elements.
   * Do not change the DOM structure.
3. **Custom Directives:**
   * Enable reusable logic and behavior.
   * Use the @Directive decorator to define custom directives.

**Advantages of Using Directives**

* **Reusability:** Encapsulate behaviors for repeated use across components.
* **Dynamic Behavior:** Enable dynamic updates to UI based on application state.
* **Simplified Development:** Abstract complex DOM manipulations into reusable directives.

Directives are a foundational concept in Angular, empowering developers to create dynamic, reusable, and maintainable UI components.

**7. Pipes**

[Go to Index](#Index)

Transform output in templates.

**Code Example:**

<p>{{ date **|** **date**:'longDate' }}</p>

**Explanation:**

* Angular pipes (e.g., date, uppercase, currency) transform data for display.

**Summary of Built-in Pipes**

| **Pipe** | **Description** |
| --- | --- |
| **uppercase** | Converts a string to uppercase. |
| **lowercase** | Converts a string to lowercase. |
| **titlecase** | Converts the first letter of each word to uppercase. |
| **number** | Formats a number with customizable precision. |
| **currency** | Formats a number as a currency value. |
| **percent** | Formats a number as a percentage. |
| **date** | Formats a date or time value. |
| **slice** | Extracts a portion of a string or array. |
| **keyvalue** | Converts an object or map into key-value pairs. |
| **json** | Converts an object into a JSON string. |
| **async** | Subscribes to and displays values from an observable/promise. |

**Best Practices with Pipes**

1. **Use Built-in Pipes:** Leverage Angular's built-in pipes whenever possible for standard transformations.
2. **Avoid Complex Logic in Pipes:** Keep pipes simple and lightweight to avoid performance bottlenecks.
3. **Combine Pipes:** Chain multiple pipes to achieve complex transformations.
4. **Custom Pipes for Specific Needs:** Create reusable custom pipes for transformations not covered by the built-in options.

Pipes are a key feature in Angular that enhance templates by providing elegant and efficient data transformations.

Angular comes with a rich set of built-in pipes that allow you to transform data easily in your templates. These pipes can handle common transformations for strings, numbers, dates, arrays, objects, and more

**Text Transformation Pipes**

**uppercase**

Converts a string to uppercase.

**Example:**

html

CopyEdit

<p>{{ 'angular' | **uppercase** }}</p> <!-- Output: ANGULAR -->

**lowercase**

Converts a string to lowercase.

**Example:**

<p>{{ 'ANGULAR' | **lowercase** }}</p> <!-- Output: angular -->

**titlecase**

Converts the first letter of each word in a string to uppercase.

**Example:**

<p>{{ 'hello angular world' | **titlecase** }}</p> <!-- Output: Hello Angular World -->

**Number and Currency Pipes**

**number**

Formats a number with specified digit options.

**Example:**

<p>{{ 1234.5678 | **number**:'1.2-2' }}</p> <!-- Output: 1,234.57 -->

* **1.2-2**: At least 1 digit before the decimal, 2 minimum and 2 maximum digits after the decimal.

**currency**

Formats a number as currency.

**Example:**

<p>{{ 1234.56 | **currency**:'USD':'symbol' }}</p> <!-- Output: $1,234.56 -->

* Parameters:
  + **Currency Code:** (e.g., 'USD', 'EUR').
  + **Display Type:** 'symbol' (default), 'code', or 'symbol-narrow'.
  + **Digit Options:** '1.2-2'.

**percent**

Formats a number as a percentage.

**Example:**

<p>{{ 0.1234 | **percent**:'1.1-2' }}</p> <!-- Output: 12.34% -->

* **1.1-2**: At least 1 digit before the decimal, 1 minimum, and 2 maximum digits after the decimal.

**Date Pipe**

**date**

Formats a date according to the specified format and locale.

**Example:**

<p>{{ today | **date**:'fullDate' }}</p> <!-- Output: Sunday, January 19, 2025 -->

**Common Formats:**

* **short**: e.g., 1/19/25, 3:15 PM.
* **medium**: e.g., Jan 19, 2025, 3:15:30 PM.
* **longDate**: e.g., January 19, 2025.
* **fullDate**: e.g., Sunday, January 19, 2025.

**Custom Format Tokens:**

* yyyy: Year (e.g., 2025).
* MM: Month (e.g., 01).
* dd: Day (e.g., 19).
* HH: Hour (24-hour format).
* mm: Minute.
* ss: Second.

**Array and Collection Pipes**

**slice**

Extracts a subset of an array or string.

**Example:**

<p>{{ [1, 2, 3, 4, 5] | **slice**:1:4 }}</p> <!-- Output: [2, 3, 4] -->

<p>{{ 'Angular' | **slice**:0:3 }}</p> <!-- Output: Ang -->

* **Parameters:**
  + **Start Index:** The starting position (inclusive).
  + **End Index:** The ending position (exclusive).

**keyvalue**

Converts an object or map into an array of key-value pairs.

**Example:**

<div \*ngFor="let pair of object | **keyvalue**">

<p>{{ pair.key }}: {{ pair.value }}</p>

</div>

* **Input:**

object = { name: 'Angular', version: 16 };

* **Output:**

name: Angular

version: 16

**Utility Pipes**

**json**

Converts an object into a JSON string for display.

**Example:**

<p>{{ { name: 'Angular', version: 16 } | **json** }}</p>

* **Output:**

{"name":"Angular","version":16}

**async**

Handles asynchronous operations like Promise or Observable and automatically subscribes to them, displaying the emitted values.

**Example:**

<p>{{ data$ | **async** }}</p>

* **Input:** data$ is an observable.
* **Behavior:** Automatically subscribes and unsubscribes to the observable.

**8. Forms Handling**

[Go to Index](#Index)

Angular provides two approaches for building and managing forms:

1. **Template-Driven Forms**: Easier to set up, uses directives in the HTML template.
2. **Reactive Forms**: More robust, programmatically managed, and ideal for complex forms.

Both approaches leverage Angular's forms module to handle user input, validation, and submission.

**Template-Driven Forms**

Template-driven forms rely heavily on the template (HTML) to define and bind form elements. Angular directives such as **ngModel** are used for two-way binding.

**Example:**

**HTML Template:**

<**form** #form="ngForm" (ngSubmit)="onSubmit(form)">

<**input**

type="text"

name="name"

**[(ngModel)]**="user.name"

required

placeholder="Enter your name"

>

<**input**

type="email"

name="email"

**[(ngModel)]**="user.email"

required

placeholder="Enter your email"

>

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

</**form**>

**Component:**

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

@Component({

selector: 'app-template-form',

templateUrl: './template-form.component.html'

})

export class TemplateFormComponent {

user = { name: '', email: '' };

onSubmit(form: any) {

console.log('Form Data:', form.value);

}

}

**Key Features:**

* **ngModel:** Binds form inputs to the component's model (user).
* **#form="ngForm"**: References the form in the template.
* **Validation:** Automatically adds classes like **ng-valid** or **ng-invalid** to form controls.

**Reactive Forms**

Reactive forms provide more control and flexibility by managing the form structure and logic programmatically. This approach is ideal for complex and dynamic forms.

**Key Classes:**

* **FormGroup**: Represents the entire form as a group.
* **FormControl**: Represents individual form controls.
* **FormBuilder**: Simplifies the creation of forms.

**Example:**

**Component:**

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

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

@Component({

selector: 'app-reactive-form',

template: `

<**form** [**formGroup**]="form" (ngSubmit)="onSubmit()">

<label>

Name:

<input formControlName="name" placeholder="Enter your name">

</label>

<label>

Email:

<input formControlName="email" placeholder="Enter your email">

</label>

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

</**form**>

`

})

export class ReactiveFormComponent {

form: **FormGroup**;

constructor(private fb: **FormBuilder**) {

this.form = this.fb.group({

name: ['', **Validators**.required],

email: ['', [**Validators**.required, Validators.email]]

});

}

onSubmit() {

console.log('Form Data:', this.form.value);

}

}

**Explanation:**

* **FormBuilder**: Simplifies form creation by combining FormGroup and FormControl.
* **Validators**: Adds validation rules to the form controls (e.g., required, email).

**Key Differences Between Template-Driven and Reactive Forms**

| **Feature** | **Template-Driven Forms** | **Reactive Forms** |
| --- | --- | --- |
| **Setup** | Defined in the template using directives like ngModel. | Defined in the component using FormGroup and FormControl. |
| **Form Structure** | Implicitly created by Angular. | Explicitly created by the developer. |
| **Validation** | Defined in the template using attributes like required. | Defined programmatically using Validators. |
| **Complexity** | Simple and easier to set up for basic forms. | Ideal for complex and dynamic forms. |
| **Scalability** | Limited for large or highly interactive forms. | Highly scalable and manageable for advanced use cases. |

**9. Routing and Navigation**

[Go to Index](#Index)

Angular provides a router for navigation between views.

**Code Example:**

// **app-routing.module.ts**

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** = [

{ path: '', component: HomeComponent },

{ path: 'about', component: AboutComponent }

];

@NgModule({

imports: [**RouterModule.forRoot**(routes)],

exports: [**RouterModule**]

})

export class AppRoutingModule { }

**Explanation:**

* Define routes using the **Routes** array.
* Use **RouterModule.forRoot()** for app-wide routing.
* Use **<router-outlet>** in templates to specify where routed components should load.

**10. HttpClient for API Integration**

[Go to Index](#Index)

Angular provides **HttpClient** for communication with REST APIs.

**Code Example:**

import { **HttpClient** } from '@angular/common/http';

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

@Component({

selector: 'app-data',

template: `<ul><li \*ngFor="let user of users">{{ user.name }}</li></ul>`

})

export class DataComponent {

users: any[] = [];

constructor(private http: **HttpClient**) {

**this.http.get**<any[]>('https://jsonplaceholder.typicode.com/users')

.**subscribe**(data => this.users = data);

}

}

**Explanation:**

* Use **HttpClient** to make HTTP requests.
* **subscribe()** handles the **asynchronous** **response**.

**11. Lifecycle Hooks**

[Go to Index](#Index)

Angular provides lifecycle hooks to tap into key events during a component’s lifecycle.

**Code Example:**

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

@Component({

selector: 'app-lifecycle',

template: `<p>Check the console for lifecycle logs.</p>`

})

export class LifecycleComponent **implements** **OnInit**, **OnDestroy** {

**ngOnInit**() {

console.log('Component Initialized');

}

**ngOnDestroy**() {

console.log('Component Destroyed');

}

}

**Explanation:**

* **ngOnInit**: Invoked once the component is initialized.
* **ngOnDestroy**: Invoked just before the component is destroyed.
* Other hooks include **ngOnChanges, ngAfterViewInit, ngAfterContentInit**, etc.

**12. Dependency Injection (DI)**

[Go to Index](#Index)

Angular has a built-in DI system to manage services and dependencies.

**Code Example:**

**Service Creation**:

* You created a service (GreetingService) using the @Injectable decorator with the providedIn: 'root' configuration, which ensures the service is a singleton and available throughout the application.
* The getGreeting method in the service returns the greeting message.

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

**@Injectable**({ providedIn: 'root' })

export class GreetingService {

getGreeting() {

return 'Hello from Service!';

}

}

**Service Consumption in the Component**:

* The GreetingComponent class declares a greeting property and initializes it in the constructor using the GreetingService.
* The greeting property is then displayed in the component's template.

@Component({

selector: 'app-greeting',

template: `<p>{{ greeting }}</p>`

})

export class GreetingComponent {

greeting: string;

constructor(private greetingService: GreetingService) {

this.greeting = this.greetingService.getGreeting();

}

}

**Explanation:**

* **@Injectable** marks a class as a service that can be injected.
* The service is injected into the component's constructor.

**13. Custom Directives**

[Go to Index](#Index)

You can create your own structural or attribute directives.

**Code Example:**

import { **Directive**, **ElementRef**, **Renderer2**, **HostListener** } from '@angular/core';

**@Directive**({

selector: '**[appHighlight]**'

})

export class HighlightDirective {

constructor(private el: **ElementRef**, private renderer: **Renderer2**) {}

**@HostListener**('mouseenter') **onMouseEnter**() {

**this.renderer**.setStyle(**this.el.nativeElement**, 'background-color', 'yellow');

}

**@HostListener**('mouseleave') **onMouseLeave**() {

**this.renderer**.removeStyle(**this.el.nativeElement**, 'background-color');

}

}

**Usage:**

<p **appHighlight**>Hover over me!</p>

**Explanation:**

* Use **ElementRef** to access the DOM element and **Renderer2** for safer DOM manipulation.
* **HostListener** listens to events on the host element.

**14. Custom Pipes**

[Go to Index](#Index)

Create custom pipes for transforming data in templates.

**Code Example:**

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

@Pipe({ name: **'reverse'** })

export class ReversePipe **implements** **PipeTransform** {

transform(value: string): string {

return value.split('').reverse().join('');

}

}

**Usage:**

<p>{{ 'Angular' | **reverse** }}</p> <!-- Outputs: ralugnA -->

**Explanation:**

* Custom pipes extend the built-in ones for specific transformation needs.

**15. Modular Architecture**

[Go to Index](#Index)

Angular apps are divided into modules for better organization.

**Code Example:**

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

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

**@NgModule**({

declarations: [AppComponent],

imports: [BrowserModule],

providers: [],

bootstrap: [AppComponent]

})

export class AppModule { }

**Explanation:**

* **Modules** group related components, services, and other code.

**16. Angular Material**

[Go to Index](#Index)

Angular Material provides pre-built UI components.

**Code Example:**

**ng add @angular/material**

**Using a Material Button:**

<**button mat-button**>Click Me</**button**>

**Module Import:**

import { **MatButtonModule** } from '@angular/material/button';

@NgModule({

imports: [**MatButtonModule**]

})

export class AppModule {}

**Explanation:**

* Use Material components like **MatButton**, **MatToolbar**, etc., for a consistent UI.

**17. Component Communication**

[Go to Index](#Index)

Components can communicate using **Input/Output decorators** or services.

**Parent to Child:**

@Component({

selector: 'app-child',

template: `<p>{{ message }}</p>`

})

export class ChildComponent {

**@Input()** message!: string;

}

<app-child [message]="'Hello from Parent'"></app-child>

**Child to Parent:**

@Component({

selector: 'app-child',

template: `<button (click)="sendMessage()">Send</button>`

})

export class ChildComponent {

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

sendMessage() {

this.notify.**emit**('Hello Parent!');

}

}

<app-child **(**notify**)**="onNotify($event)"></app-child>

**Explanation:**

* Use **@Input** and **@Output** for direct communication.
* Use services for complex interactions.

**18. Host Binding and Host Listener**

[Go to Index](#Index)

Interact with the host element directly in a directive or component.

**Code Example:**

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

@Directive({

selector: '[appHighlight]'

})

export class HighlightDirective {

**@HostBinding**('style.backgroundColor') bgColor!: string;

**@HostListener**('mouseenter') onMouseEnter() {

this.bgColor = 'yellow';

}

**@HostListener**('mouseleave') onMouseLeave() {

this.bgColor = 'transparent';

}

}

**Explanation:**

* **HostBinding** binds a property to the host element.
* **HostListener** listens to events on the host element.

**19.** **Preloading Modules**

[Go to Index](#Index)

Improve performance by preloading lazy-loaded modules.

**Code Example:**

const routes: Routes = [

{ path: 'feature', **loadChildren**: () => **import**('./feature/feature.module').**then**(m => m.FeatureModule) }

];

**@NgModule**({

imports: [RouterModule.forRoot(routes, { **preloadingStrategy**: **PreloadAllModules** })],

exports: [RouterModule]

})

export class **AppRoutingModule** {}

**Explanation:**

* Use **PreloadAllModules** to preload modules in the background after the app loads.

**20. Lazy Loading Modules**

[Go to Index](#Index)

Angular supports lazy loading to optimize app performance by loading modules on demand.

**Code Example:**

// **app-routing.module.ts**

const routes: **Routes** = [

{ path: 'feature', **loadChildren**: () => import('./feature/feature.module').then(m => m.FeatureModule) }

];

**Explanation:**

* Use the **loadChildren** property to dynamically load a module only when the route is accessed.

**21. Content Projection**

[Go to Index](#Index)

Content projection allows you to insert external content into a component's template.

**Code Example:**

@Component({

selector: 'app-card',

template: `

<div class="card">

<**ng-content**></**ng-content**>

</div>

`

})

export class CardComponent {}

**Usage:**

<app-card>

<p>This is projected content!</p>

</app-card>

**Explanation:**

* **<ng-content>** acts as a placeholder for external content.

**22. Dynamic Forms**

[Go to Index](#Index)

Angular's **Reactive Forms** are highly flexible for creating dynamic forms.

**Code Example:**

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

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

@Component({

selector: 'app-dynamic-form',

template: `

<form [**formGroup**]="form">

<div **formArrayName**="fields">

<div \*ngFor="let field of fields.controls; let i = index">

<input [formControlName]="i" placeholder="Field {{ i + 1 }}">

</div>

</div>

<button (click)="addField()">Add Field</button>

</form>

`

})

export class DynamicFormComponent {

form: FormGroup;

constructor(private fb: **FormBuilder**) {

this.form = **this.fb.group**({

fields: **this.fb.array**([this.fb.control('')])

});

}

get fields() {

return this.form.get('fields') as **FormArray**;

}

addField() {

this.fields.push(this.fb.control(''));

}

}

**Explanation:**

* Use **FormArray** to create dynamic form controls at runtime.

**23. Resolver in Routing**

[Go to Index](#Index)

**Resolvers** fetch data before navigating to a route.

**Code Example:**

// **resolver.service.ts**

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

import { **Resolve** } from '@angular/router';

import { Observable, of } from 'rxjs';

@Injectable({ providedIn: 'root' })

export class DataResolver implements **Resolve**<string> {

resolve(): Observable<string> {

return of('Resolved Data');

}

}

**Routing Configuration:**

const routes: Routes = [

{ path: 'example', component: ExampleComponent, resolve: { data: DataResolver } }

];

**Usage in Component:**

@Component({

selector: 'app-example',

template: `<p>{{ data }}</p>`

})

export class ExampleComponent {

data: string;

constructor(route: ActivatedRoute) {

this.data = route.snapshot.data['data'];

}

}

**Explanation:**

* Resolvers pre-load data for a route before navigation.

**24. Environment Configuration**

[Go to Index](#Index)

Angular supports environment-specific configurations.

**Code Example:**

// **environment.ts**

export const **environment** = {

production: false,

apiUrl: 'http://localhost:3000'

};

// **environment.prod.ts**

export const **environment** = {

production: true,

apiUrl: 'https://api.example.com'

};

**Usage:**

import { **environment** } from '../environments/environment';

console.log(environment.apiUrl);

**Explanation:**

* Configure multiple environments like dev, prod, and staging.

**25. Testing**

[Go to Index](#Index)

Angular provides tools for unit testing (**Karma**, **Jasmine**).

**Component Testing Example:**

import { TestBed } from '@angular/core/testing';

import { HelloComponent } from './hello.component';

**describe**('HelloComponent', () => {

**beforeEach**(() => {

TestBed.configureTestingModule({

declarations: [HelloComponent]

});

});

**it**('should create', () => {

const fixture = TestBed.createComponent(HelloComponent);

const component = fixture.componentInstance;

expect(component).toBeTruthy();

});

});

**Explanation:**

* **TestBed** is Angular's primary API for testing components and services.

**26. Unit Testing with Jasmine and Karma**

[Go to Index](#Index)

Angular CLI sets up a testing framework by default.

**Component Test:**

import { ComponentFixture, **TestBed** } from '@angular/core/testing';

import { MyComponent } from './my.component';

**describe**('MyComponent', () => {

let component: MyComponent;

let fixture: ComponentFixture<MyComponent>;

**beforeEach**(() => {

TestBed.configureTestingModule({

declarations: [MyComponent]

});

fixture = **TestBed**.createComponent(MyComponent);

component = fixture.componentInstance;

fixture.detectChanges();

});

**it**('should create', () => {

expect(component).toBeTruthy();

});

});

**Explanation:**

* Use **TestBed** to configure the testing environment.
* Test components, services, and directives.

**27. Error Handling with Interceptors**

[Go to Index](#Index)

Use **HttpInterceptors** to handle API errors globally.

**Code Example:**

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

import { **HttpInterceptor**, HttpRequest, HttpHandler, HttpErrorResponse } from '@angular/common/http';

import { catchError } from 'rxjs/operators';

import { throwError } from 'rxjs';

@Injectable()

export class ErrorInterceptor **implements** **HttpInterceptor** {

**intercept**(req: HttpRequest<any>, next: HttpHandler) {

return next.handle(req).pipe(

catchError((error: HttpErrorResponse) => {

console.error('Error occurred:', error.message);

return throwError(error);

})

);

}

}

**Explanation:**

* Register the interceptor in the **providers** array in your module to handle errors or modify requests globally.

**28.** **Advanced Dependency Injection**

[Go to Index](#Index)

Angular’s DI system supports providers at various levels.

**Hierarchical DI Example:**

**@Injectable**({ **providedIn: 'root'** })

export class **GlobalService** {

getData() {

return 'Global Data';

}

}

**@Injectable**()

export class **LocalService** {

getData() {

return 'Local Data';

}

}

@Component({

selector: 'app-local-di',

template: `<p>{{ service.getData() }}</p>`,

**providers**: **[LocalService]**

})

export class LocalDiComponent {

constructor(public service: **LocalService**) {}

}

**Explanation:**

* Use **providedIn: 'root'** for app-wide services.
* Use the **providers** array for component-specific services.

**29. Custom Validators**

[Go to Index](#Index)

Angular forms support custom validators.

**Code Example:**

import { **AbstractControl**, **ValidationErrors** } from '@angular/forms';

export function forbiddenNameValidator(control: **AbstractControl**): **ValidationErrors** | null {

const forbidden = /admin/.test(control.value);

return forbidden ? { forbiddenName: { value: control.value } } : null;

}

**Usage:**

this.form = this.fb.group({

username: ['', [forbiddenNameValidator]]

});

**Explanation:**

* Custom validators enforce specific validation logic.

**30. Change Detection**

[Go to Index](#Index)

Angular's **Change Detection** ensures the UI stays in sync with data.

**Key Strategies:**

* **Default**: Runs change detection for all components.
* **OnPush**: Runs detection only when an input changes.

**Code Example:**

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

@Component({

selector: 'app-immutable',

template: `<p>{{ data }}</p>`,

changeDetection: **ChangeDetectionStrategy**.**OnPush**

})

export class ImmutableComponent {

@Input() data: string;

}

**Explanation:**

* Use **OnPush** for performance optimization in components with immutable inputs.

**31. State Management with Services**

[Go to Index](#Index)

For simpler apps, **services handle state** without external libraries like NgRx.

**Code Example:**

@Injectable({ providedIn: 'root' })

export class **CounterService** {

private count = 0;

**getCount**() {

return this.count;

}

increment() {

this.count++;

}

}

@Component({

selector: 'app-counter',

template: `

<p>Count: {{ counterService.**getCount**() }}</p>

<button (click)="counterService.increment()">Increment</button>

`

})

export class CounterComponent {

constructor(public counterService: **CounterService**) {}

}

**Explanation:**

* Services can manage and share state across components without needing complex libraries.

**32. Animation Support**

[Go to Index](#Index)

Angular includes **@angular/animations** for adding dynamic animations.

**Code Example:**

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

import { **trigger**, **state**, **style**, **transition**, **animate** } from '@angular/animations';

@Component({

selector: 'app-animate',

template: `

<div [@toggle]="isOpen ? 'open' : 'closed'" (click)="toggle()">

Click to Animate

</div>

`,

animations: [

trigger('toggle', [

state('open', style({ height: '200px', backgroundColor: 'green' })),

state('closed', style({ height: '100px', backgroundColor: 'red' })),

transition('open <=> closed', animate('0.5s ease-in-out'))

])

]

})

export class AnimateComponent {

isOpen = false;

toggle() {

this.isOpen = !this.isOpen;

}

}

**Explanation:**

* Define animations using **trigger**, **state**, **style**, and **transition**.

**33. Internationalization (i18n)**

[Go to Index](#Index)

Angular supports built-in tools for localization.

**Code Example:**

<p i18n="@@welcome">Welcome to Angular!</p>

**Explanation:**

* Use the **i18n** attribute to mark translatable content.
* Angular CLI provides tools for extracting and managing translations.

**34.** **Angular Universal (Server-Side Rendering)**

[Go to Index](#Index)

Angular Universal enables server-side rendering (SSR) for better SEO and performance.

**Steps:**

1. Add Universal: **ng add** @nguniversal/express-engine
2. Build the server: **npm run build:ssr**
3. Serve the app: **npm run serve:ssr**

**Explanation:**

* With SSR, the app is rendered on the server before being sent to the browser.

**35.** **State Management with NgRx**

[Go to Index](#Index)

Angular integrates with **NgRx**, a library for **managing application state**.

**Code Example:**

// **actions.ts**

import { **createAction** } from '@ngrx/store';

export const increment = **createAction**('[Counter Component] Increment');

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

// **reducer.ts**

import { **createReducer**, on } from '@ngrx/store';

import \* as **CounterActions** from './actions';

export const initialState = 0;

export const counterReducer = **createReducer**(

initialState,

on(**CounterActions**.increment, state => state + 1),

on(**CounterActions**.decrement, state => state - 1)

);

**Explanation:**

* Use **NgRx** for centralized state management and predictability in large applications.

**36. Progressive Web Apps (PWA)**

[Go to Index](#Index)

Angular has built-in support for PWAs.

**Steps:**

1. Install PWA features: **ng add @angular/pwa**
2. Angular generates a service worker for caching resources, enabling offline support.

**37. Service Worker Integration**

[Go to Index](#Index)

Angular makes apps work offline with service workers.

**Steps:**

1. Add service worker: **ng add @angular/pwa**
2. Configure caching in **ngsw-config.json**

**Code Example:**

{

"index": "/index.html",

"assetGroups": [{

"name": "app",

"resources": {

"files": ["/\*.html", "/\*.css", "/\*.js"]

}

}]

}

**Explanation:**

* Service workers cache files and API responses for offline access.

**38.** **Web Worker Integration**

[Go to Index](#Index)

Web Workers handle heavy computations in the background to improve performance.

**Steps:**

1. Generate a worker: **ng generate web-worker** my-worker
2. Use the worker in your component:

if (typeof Worker !== 'undefined') {

const worker = new **Worker**(new URL('./my-worker.worker', import.meta.url));

**worker**.onmessage = ({ data }) => {

console.log(`Worker said: ${data}`);

};

**worker**.postMessage('Hello Worker');

}

**Explanation:**

* Web **Workers** offload resource-intensive tasks from the main thread.

**39. Signals**

[Go to Index](#Index)

Angular introduces **signals** for reactive state management, eliminating change detection's overhead.

**Code Example:**

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

@Component({

selector: 'app-signals',

template: `

<p>{{ count() }}</p>

<button (click)="increment()">Increment</button>

`

})

export class SignalsComponent {

count = **signal**(0);

increment() {

this.count.**update**(value => value + 1);

}

}

**Explanation:**

* **Signals** provide reactive state management where changes automatically update the UI.

**40. Dynamic Component Loading**

[Go to Index](#Index)

Angular supports creating and injecting components dynamically.

**Code Example:**

import { Component, **ComponentFactoryResolver**, ViewChild, **ViewContainerRef** } from '@angular/core';

@Component({

selector: 'app-dynamic',

template: `<**ng-template** #container></**ng-template**><button (click)="loadComponent()">Load Component</button>`

})

export class DynamicComponent {

**@ViewChild**('container', { read: ViewContainerRef, static: true }) container: **ViewContainerRef**;

constructor(private resolver: **ComponentFactoryResolver**) {}

loadComponent() {

const factory = **this.resolver.resolveComponentFactory**(SomeOtherComponent);

**this.container.createComponent**(factory);

}

}

**Explanation:**

* Use **ComponentFactoryResolver** and **ViewContainerRef** to create components dynamically.

**41. Internationalization (i18n) with Runtime Translations**

[Go to Index](#Index)

Angular supports runtime translation files.

**Steps:**

1. Extract translations: **ng extract-i18n**
2. Generate locale files (messages.en.xlf, etc.)
3. Use translation libraries like **ngx-translate** for runtime translations

**Code Example with ngx-translate:**

import { **TranslateService** } from '@ngx-translate/core';

@Component({

selector: 'app-translate',

template: `<p>{{ 'HELLO' | translate }}</p>`

})

export class TranslateComponent {

constructor(private translate: **TranslateService**) {

translate.setDefaultLang('en');

}

}

**43. Platform Agnostic Framework**

[Go to Index](#Index)

Angular apps can run in different environments, such as web, mobile, or desktop.

**Web Example:** Default Angular apps.

**Mobile Example:** Use **Ionic Framework** with Angular:

**npm install @ionic/angular**

**Desktop Example:** Use **Electron** with Angular:

**npm install electron --save-dev**

**44. Custom Elements (Web Components)**

[Go to Index](#Index)

Convert Angular components into **reusable web components**.

**Code Example:**

import { NgModule, **DoBootstrap** } from '@angular/core';

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

import { **createCustomElement** } from '@angular/elements';

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

**@NgModule**({

declarations: [AppComponent],

imports: [BrowserModule],

entryComponents: [AppComponent]

})

export class AppModule implements **DoBootstrap** {

constructor(private injector: Injector) {

const customElement = **createCustomElement**(AppComponent, { injector });

customElements.**define**('app-element', customElement);

}

ngDoBootstrap() {}

}

**Usage:**

<app-element></app-element>

**Explanation:**

* Use **createCustomElement** to turn Angular components into web components.

**45. Reactive Programming with RxJS**

[Go to Index](#Index)

Angular heavily relies on **RxJS** (Reactive Extensions for JavaScript) for reactive programming, enabling developers to **handle asynchronous data streams** efficiently.

**Code Example:**

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

import { **of**, **interval** } from 'rxjs';

import { **map**, **take** } from 'rxjs/operators';

@Component({

selector: 'app-rxjs-example',

template: `

<p>Numbers: {{ numbers | async }}</p>

<button (click)="emitValues()">Start</button>

`

})

export class RxjsExampleComponent {

numbers: any;

emitValues() {

this.numbers = **interval**(1000).pipe(

**map**(value => value \* 2),

**take**(5) // Emit only 5 values

);

}

}

**Explanation:**

* **RxJS Observables** represent data streams that emit values over time.
* Operators like **map**, **take**, and **filter** transform these streams.

**46.** **Standalone Components**

[Go to Index](#Index)

Angular introduced **standalone components** to reduce module complexity.

**Code Example:**

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

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

@Component({

**standalone: true**,

imports: [**CommonModule**],

selector: 'app-standalone',

template: `<h2>Standalone Component</h2>`

})

export class StandaloneComponent {}

**Explanation:**

* Add **standalone: true** in the component decorator.
* Use the **imports** property to include required Angular modules like **CommonModule**.

### 47. ****Attribute Binding****

[Go to Index](#Index)

Bind element attributes dynamically.

**Example:**

@Component({

selector: 'app-root',

template: `<img **[attr.src]**="**imageSrc**" alt="Dynamic Image">`,

})

export class AppComponent {

**imageSrc** = 'assets/logo.png';

}

### 48. ****TrackBy in ngFor****

[Go to Index](#Index)

Optimize rendering by using trackBy to reduce DOM manipulation.

**Example:**

<ul>

<li **\*ngFor**="**let** item **of** items; **trackBy**: trackByFn">{{ item.name }}</li>

</ul>

trackByFn(index: number, item: any): number {

return item.id;

}

### 49. ****Standalone API (Bootstrap without Modules)****

[Go to Index](#Index)

New Angular 16+ feature to bootstrap standalone components.

**Example:**

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

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

**bootstrapApplication**(AppComponent);

### 50. ****Bootstrap with Modules****

[Go to Index](#Index)

The traditional way of bootstrapping Angular applications by defining modules.

**Example:**

// app.module.ts

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

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

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

@NgModule({

declarations: [**AppComponent**], // Declare components

imports: [BrowserModule], // Import required modules

**bootstrap**: [**AppComponent**], // Bootstrap the root component

})

export class AppModule {}

// main.ts

import { **platformBrowserDynamic** } from '@angular/platform-browser-dynamic';

import { **AppModule** } from './app/app.module';

**platformBrowserDynamic**().**bootstrapModule**(**AppModule**)

.catch(err => console.error(err));

This method ensures a modular architecture and is commonly used in larger, enterprise-level applications. It provides explicit structure and scalability by defining modules and their dependencies.

### 51. ****Component Store****

[Go to Index](#Index)

For local state management without **NgRx**.

**Example:**

@Injectable()

export class CounterStore extends ComponentStore<{ count: number }> {

constructor() {

super({ count: 0 });

}

readonly increment = this.updater(state => ({ count: state.count + 1 }));

}

### 52. ****Typed Forms****

[Go to Index](#Index)

Strongly typed reactive forms in Angular 14+.

**Example:**

const loginForm = new **FormGroup**({

username: new **FormControl**<string>('', Validators.required),

password: new **FormControl**<string>('', Validators.required),

});

### 53. ****ViewChild and ViewChildren****

[Go to Index](#Index)

Access child components or elements.

**Example:**

**@ViewChild**('child') childComponent!: **ChildComponent**;

ngAfterViewInit() {

console.log(this.childComponent.data);

}

### 54. ****Enhanced Component Styles****

[Go to Index](#Index)

Scoped styles with Angular’s encapsulation strategies like

**ViewEncapsulation.ShadowDom**.

**Example:**

@Component({

selector: 'app-root',

template: `<p>Scoped Styles</p>`,

styles: [`p { color: red; }`],

**encapsulation**: **ViewEncapsulation.ShadowDom**,

})

export class AppComponent {}

### 55. ****Standalone Routing****

[Go to Index](#Index)

Configure routing directly without NgModule.

**Example:**

import { **provideRouter**, Routes } from '@angular/router';

const routes: **Routes** = [{ path: '', component: HomeComponent }];

**bootstrapApplication**(AppComponent, {

providers: [**provideRouter**(routes)],

});

**56. Zone.js Configuration and Optimization**

[Go to Index](#Index)

Fine-tune **Zone.js** behavior to improve performance.

* Use **zone-flags.ts** to disable patching for specific APIs like **setTimeout** or **addEventListener** for high-performance scenarios.

**57. Strict Mode for TypeScript**

[Go to Index](#Index)

Enable Angular's strict typing mode for better code safety and reliability.

* Configure **strictTemplates** and **strict** options in **tsconfig.json** for robust development practices.

**58. Control Value Accessors (CVA)**

[Go to Index](#Index)

Create custom form control components using CVA for seamless integration with Angular's reactive forms.

import { **ControlValueAccessor**, **NG\_VALUE\_ACCESSOR** } from '@angular/forms';

@Component({

selector: 'custom-input',

providers: [

{

provide: **NG\_VALUE\_ACCESSOR**,

useExisting: forwardRef(() => CustomInputComponent),

multi: true,

},

],

})

export class CustomInputComponent implements **ControlValueAccessor** {

// Implement CVA methods

}

**59. ESM Build and Tree Shaking**

[Go to Index](#Index)

Utilize Angular's ESM builds for better tree-shaking, reducing bundle sizes for faster loading.

**60. Builder API for Custom CLI Builders**

[Go to Index](#Index)

Leverage Angular CLI's Builder API to create custom build workflows tailored to your project needs.

**61. Optional NgModules**

[Go to Index](#Index)

Simplify module dependency management with optional NgModules, introduced in Angular 16+.

**62. Efficient Component Reuse with Directives**

[Go to Index](#Index)

Convert reusable behaviors into directives instead of duplicating logic across multiple components.

**63. Signals and Effects with RxJS Integration**

[Go to Index](#Index)

Combine Angular Signals with RxJS Observables to create a hybrid reactive architecture.

**64. Scoped Lazy Loading**

[Go to Index](#Index)

Restrict lazy loading to specific routes or features to avoid loading unnecessary modules.

**65. Hybrid Applications**

[Go to Index](#Index)

Mix standalone components with traditional NgModules for progressive migration to modern Angular.

**66. Differential Loading**

[Go to Index](#Index)

Generate separate builds for modern and legacy browsers for optimized delivery using Angular CLI.

**67. Component Dev Kit (CDK)**

[Go to Index](#Index)

Use Angular CDK for utilities like drag-and-drop, virtual scrolling, overlays, and more.

**68. Configurable RendererFactory2**

[Go to Index](#Index)

Extend Angular's rendering capabilities by customizing RendererFactory2.

**69. Micro Frontends**

[Go to Index](#Index)

Use Angular's Module Federation for building and integrating micro frontends in enterprise applications.

**70. DevTools Integration**

[Go to Index](#Index)

Enhance debugging with Angular DevTools for component tree visualization and performance profiling.