Install Angular CLI (Command Line Interface):

npm install -g @angular/cli

Create a new Angular project:

ng new my-angular-app
cd my-angular-app
ng serve

2. Project Structure Overview

Understand key folders:

- `src/app/` – where your components, services, and modules live
- `app.component.ts` – root component
- `app.module.ts` – root module

Create Components

```
ng generate component hello-world
```

4. Data Binding

- **Interpolation:** `{{ title }}`
- **Property Binding:** `[src]="imageUrl"`
- **Event Binding:** `(click)="sayHello()"`
- **Two-way Binding:** `[(ngModel)]="name"`

You'll need to import `FormsModule` for `ngModel`

5. Directives

- `*ngIf`, `*ngFor`, `ngClass`, `ngStyle`

```
<div *ngIf="isLoggedIn">Welcome!</div>
<ul>
  <li *ngFor="let user of users">{{ user.name }}</li>
</ul>
```

Services and Dependency Injection

Generate a service:

```
ng generate service user
```

Inject it into a component via constructor:

```
constructor(private userService: UserService) {}
```

7. Routing

Set up in `app-routing.module.ts`:

```
const routes: Routes = [
  { path: 'home', component: HomeComponent },
  { path: 'about', component: AboutComponent }
];
```

Use links:

```
<a routerLink="/home">Home</a>
```

8. HTTP Requests

Import `HttpClientModule` in `app.module.ts`:

```
import { HttpClientModule } from '@angular/common/http';
```

Use `HttpClient` in your service to make API calls:

```
this.http.get('https://api.example.com/data')
```

9. Forms (Template-driven or Reactive)

You can start with template-driven forms, and later learn reactive forms for more complex needs.

Template-Driven Forms (Beginner-Friendly)

These forms are declared directly in your HTML, using Angular directives like `[(ngModel)]`, and are easier to set up for small forms.

Pros of Template-Driven Forms

- Simple syntax
- Easy for small forms
- Built-in validation (`required`, `email`, etc.)

Reactive Form

```
<form [formGroup]="studentForm" (ngSubmit)="onSubmit()">
  <label>Name:</label>
  <input type="text" formControlName="name" />
  <div *ngIf="studentForm.get('name')?.touched &&
studentForm.get('name')?.invalid">
    Name is required.
  </div>

  <label>Email:</label>
  <input type="email" formControlName="email" />
  <div *ngIf="studentForm.get('email')?.touched &&
studentForm.get('email')?.invalid">
```

```
   Valid email is required.
  </div>

  <label>Gender:</label>
  <select formControlName="gender">
   <option value="male">Male</option>
   <option value="female">Female</option>
  </select>

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

<p *ngIf="submitted && studentForm.valid">
  ✅ Registration submitted: {{ studentForm.value | json }}
</p>
```

Why Use Reactive Forms?

- Full control over validation logic
- Easy to test and debug
- Useful for dynamic forms (e.g., adding/removing fields)

10. Angular Routing & Navigation

- Set up routes using `RouterModule`
- Use `<router-outlet>` and `<a [routerLink]>`
- Handle dynamic routes with parameters (e.g., `/student/:id`)
- Navigation guards (e.g., `AuthGuard`)

11. HTTP Client & API Integration

- **Use `HttpClientModule` to call REST APIs**
- **`GET`, `POST`, `PUT`, `DELETE` methods**
- **Handle errors with `catchError`**
- **Show loading indicators and success/error messages**

13. Component Communication

- **@Input() – Pass data from parent to child**
- **@Output() + `EventEmitter` – Send events from child to parent**
- **Shared services for sibling communication**

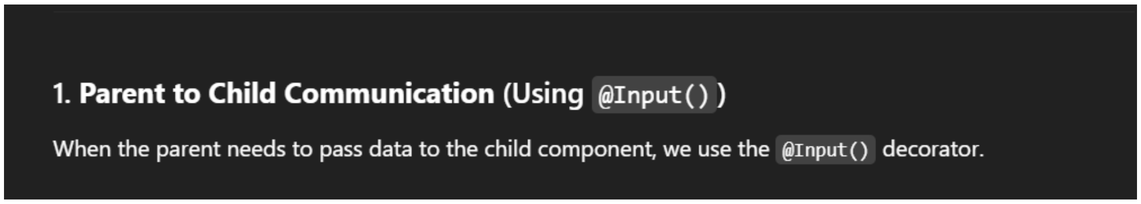
14. Pipes

- **Built-in pipes: `date`, `uppercase`, `currency`, `async`, etc.**
- **Create custom pipes for formatting data**

15. Lifecycle Hooks

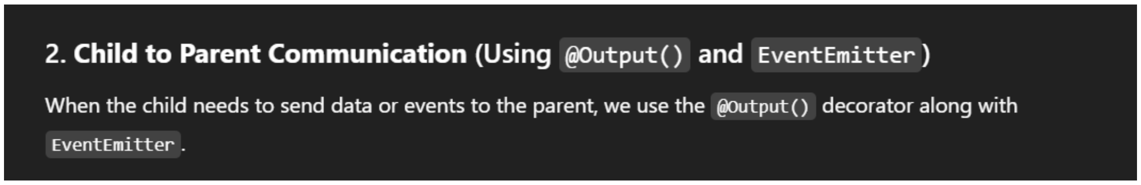
- **`ngOnInit()`, `ngOnDestroy()`, `ngOnChanges()`, etc.**
- **Useful for initialization and cleanup logic**

PARENT CHILD INTERACTION



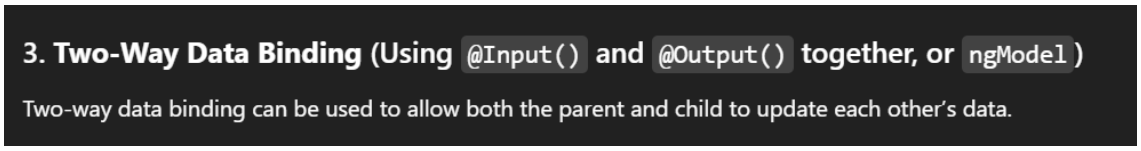
1. Parent to Child Communication (Using `@Input()`)

When the parent needs to pass data to the child component, we use the `@Input()` decorator.



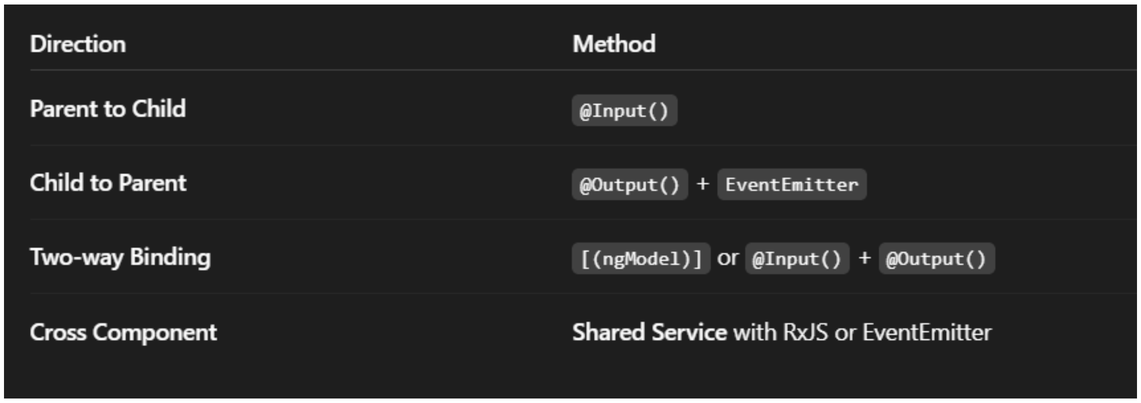
2. Child to Parent Communication (Using `@Output()` and `EventEmitter`)

When the child needs to send data or events to the parent, we use the `@Output()` decorator along with `EventEmitter`.



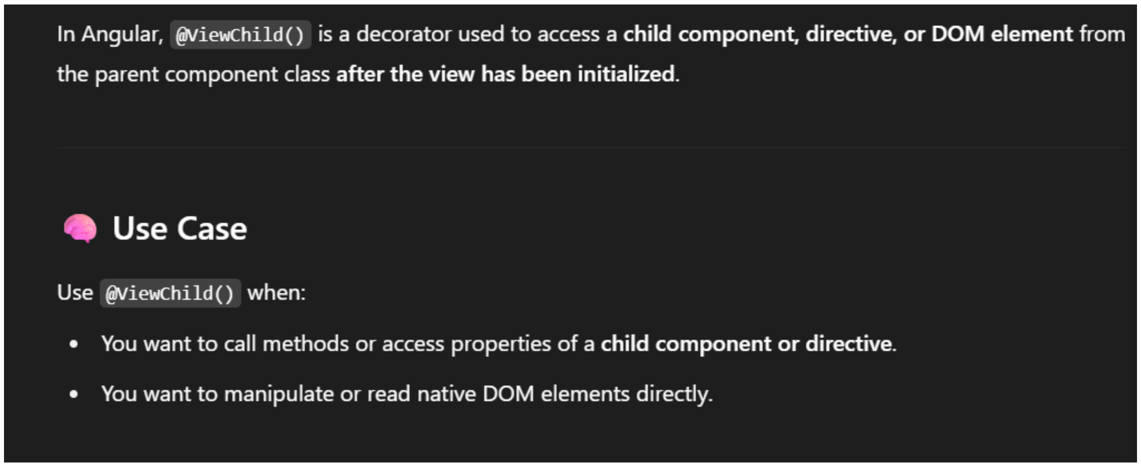
3. Two-Way Data Binding (Using `@Input()` and `@Output()` together, or `ngModel`)

Two-way data binding can be used to allow both the parent and child to update each other's data.



Direction	Method
Parent to Child	`@Input()`
Child to Parent	`@Output()` + `EventEmitter`
Two-way Binding	`[(ngModel)]` or `@Input()` + `@Output()`
Cross Component	**Shared Service** with RxJS or EventEmitter

VIEW CHILD



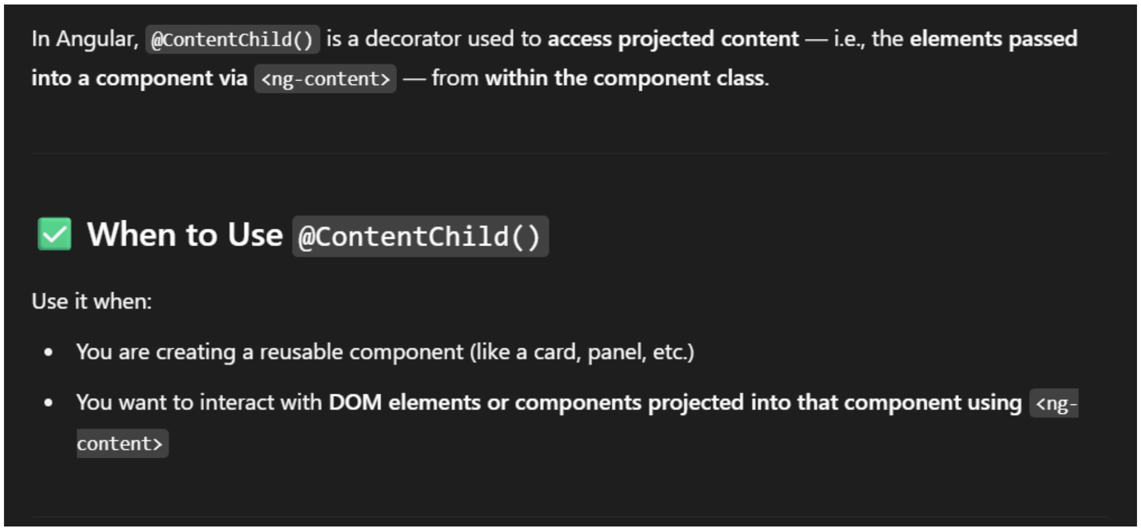
In Angular, `@ViewChild()` is a decorator used to access a **child component, directive, or DOM element** from the parent component class **after the view has been initialized**.

🧠 Use Case

Use `@ViewChild()` when:

- You want to call methods or access properties of a **child component or directive**.
- You want to manipulate or read native DOM elements directly.

CONTENT CHILD



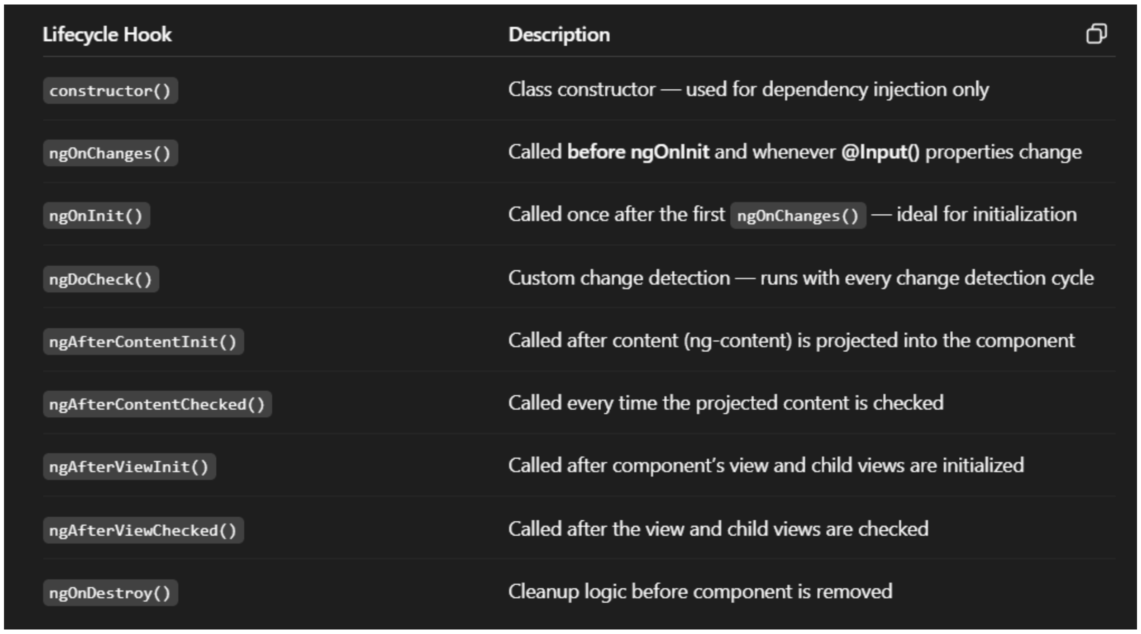
In Angular, `@ContentChild()` is a decorator used to **access projected content** — i.e., the **elements passed into a component via** `<ng-content>` — from **within the component class**.

✅ When to Use `@ContentChild()`

Use it when:

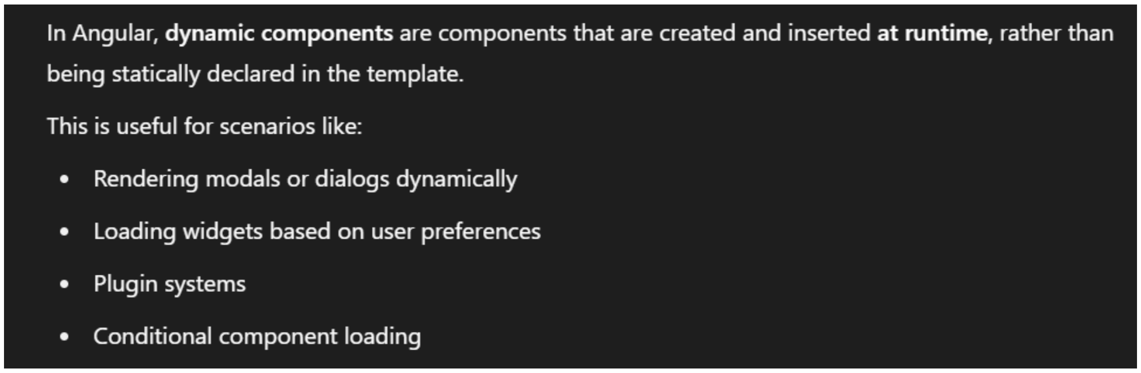
- You are creating a reusable component (like a card, panel, etc.)
- You want to interact with **DOM elements or components projected into that component using** `<ng-content>`

Lifecycle Hooks



Lifecycle Hook	Description
`constructor()`	Class constructor — used for dependency injection only
`ngOnChanges()`	Called **before ngOnInit** and whenever **@Input()** properties change
`ngOnInit()`	Called once after the first `ngOnChanges()` — ideal for initialization
`ngDoCheck()`	Custom change detection — runs with every change detection cycle
`ngAfterContentInit()`	Called after content (ng-content) is projected into the component
`ngAfterContentChecked()`	Called every time the projected content is checked
`ngAfterViewInit()`	Called after component's view and child views are initialized
`ngAfterViewChecked()`	Called after the view and child views are checked
`ngOnDestroy()`	Cleanup logic before component is removed

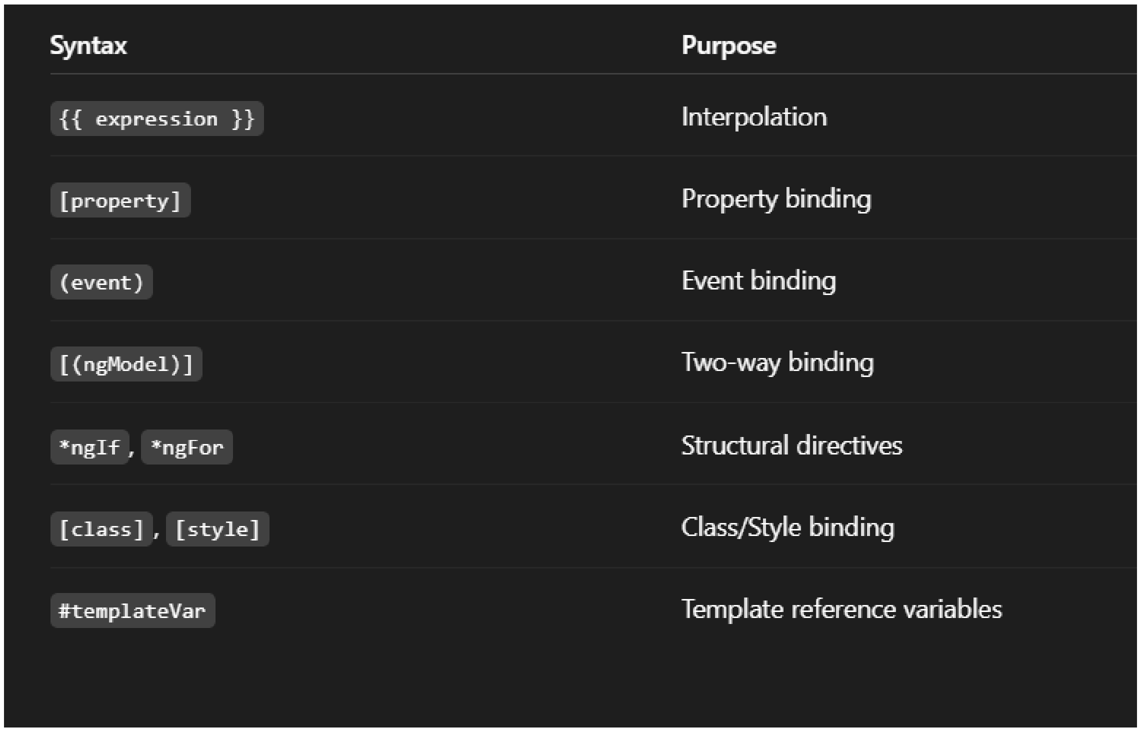
DYNAMIC COMPONENTS



In Angular, **dynamic components** are components that are created and inserted **at runtime**, rather than being statically declared in the template.

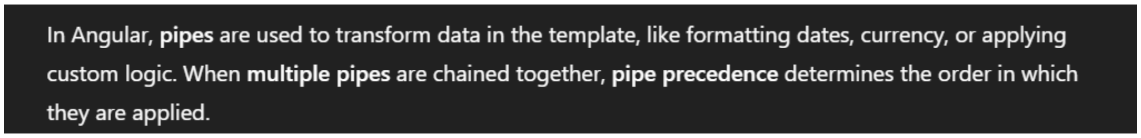
This is useful for scenarios like:

- Rendering modals or dialogs dynamically
- Loading widgets based on user preferences
- Plugin systems
- Conditional component loading

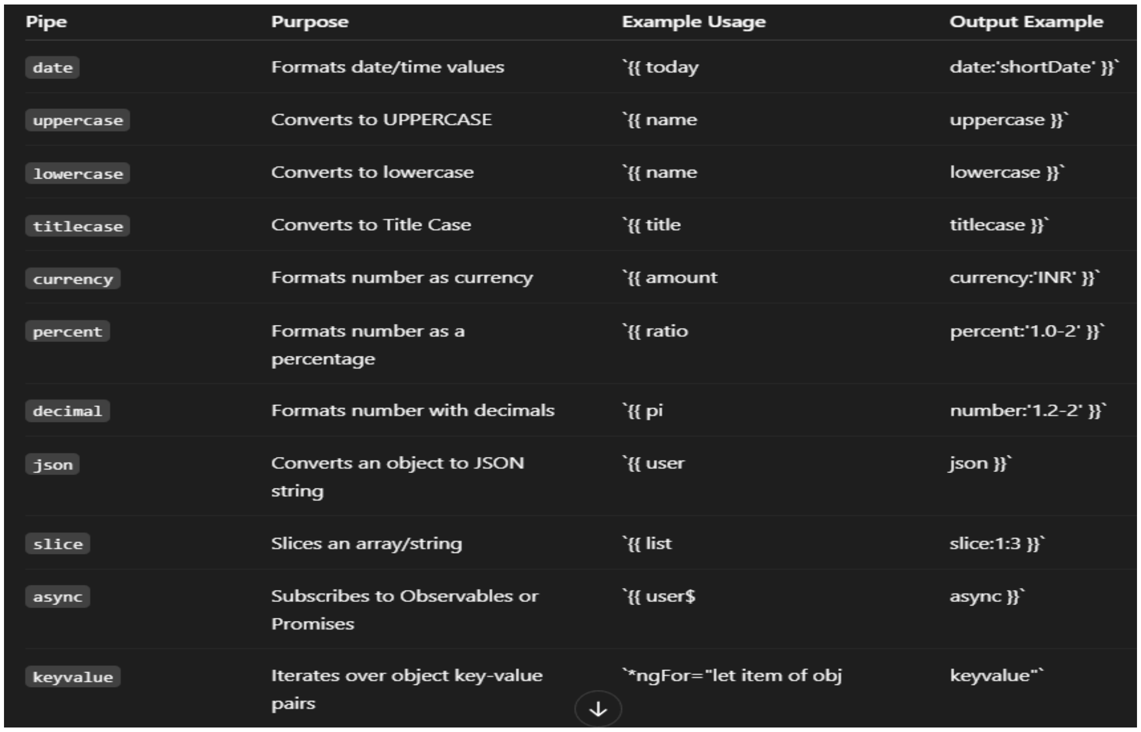


Syntax	Purpose
`{{ expression }}`	Interpolation
`[property]`	Property binding
`(event)`	Event binding
`[(ngModel)]`	Two-way binding
`*ngIf`, `*ngFor`	Structural directives
`[class]`, `[style]`	Class/Style binding
`#templateVar`	Template reference variables

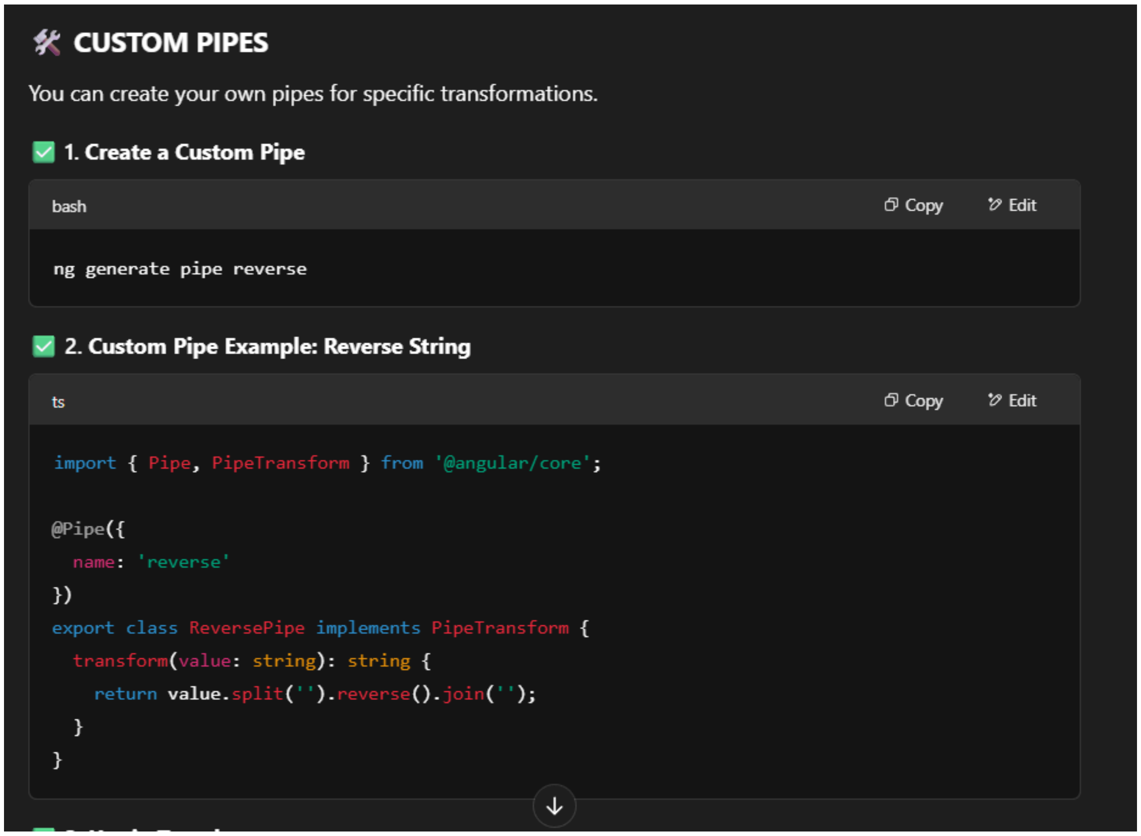
PIPES



In Angular, **pipes** are used to transform data in the template, like formatting dates, currency, or applying custom logic. When **multiple pipes** are chained together, **pipe precedence** determines the order in which they are applied.



Pipe	Purpose	Example Usage	Output Example
`date`	Formats date/time values	`{{ today	date:'shortDate' }}`
`uppercase`	Converts to UPPERCASE	`{{ name	uppercase }}`
`lowercase`	Converts to lowercase	`{{ name	lowercase }}`
`titlecase`	Converts to Title Case	`{{ title	titlecase }}`
`currency`	Formats number as currency	`{{ amount	currency:'INR' }}`
`percent`	Formats number as a percentage	`{{ ratio	percent:'1.0-2' }}`
`decimal`	Formats number with decimals	`{{ pi	number:'1.2-2' }}`
`json`	Converts an object to JSON string	`{{ user	json }}`
`slice`	Slices an array/string	`{{ list	slice:1:3 }}`
`async`	Subscribes to Observables or Promises	`{{ user$	async }}`
`keyvalue`	Iterates over object key-value pairs	`*ngFor="let item of obj	keyvalue"`



You can create your own pipes for specific transformations.

✅ 1. Create a Custom Pipe

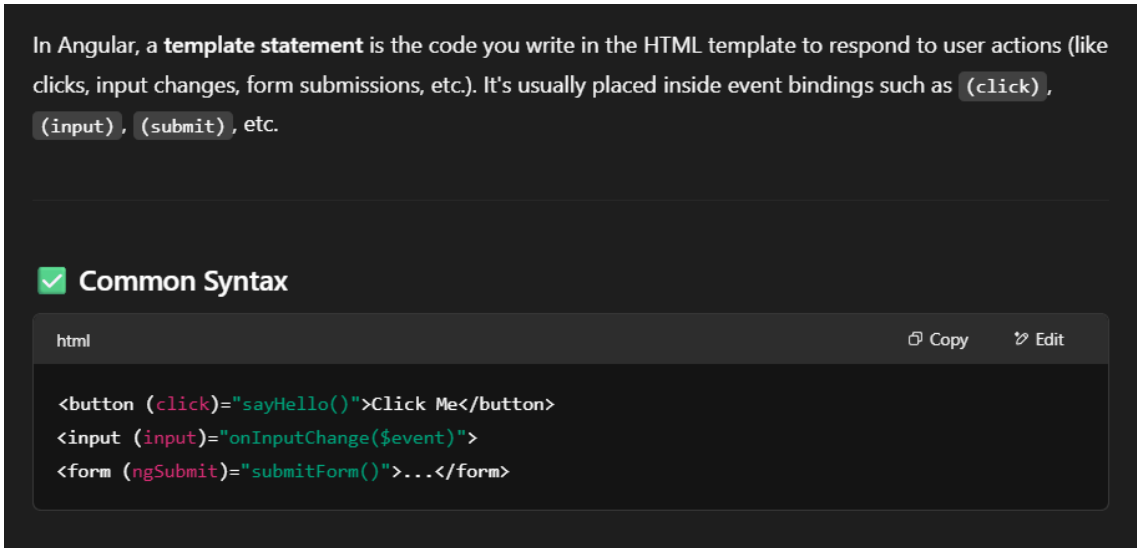
```bash
ng generate pipe reverse
```

✅ 2. Custom Pipe Example: Reverse String

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

@Pipe({
  name: 'reverse'
})
export class ReversePipe implements PipeTransform {
  transform(value: string): string {
    return value.split('').reverse().join('');
  }
}
```

TEMPLATE STATEMENT

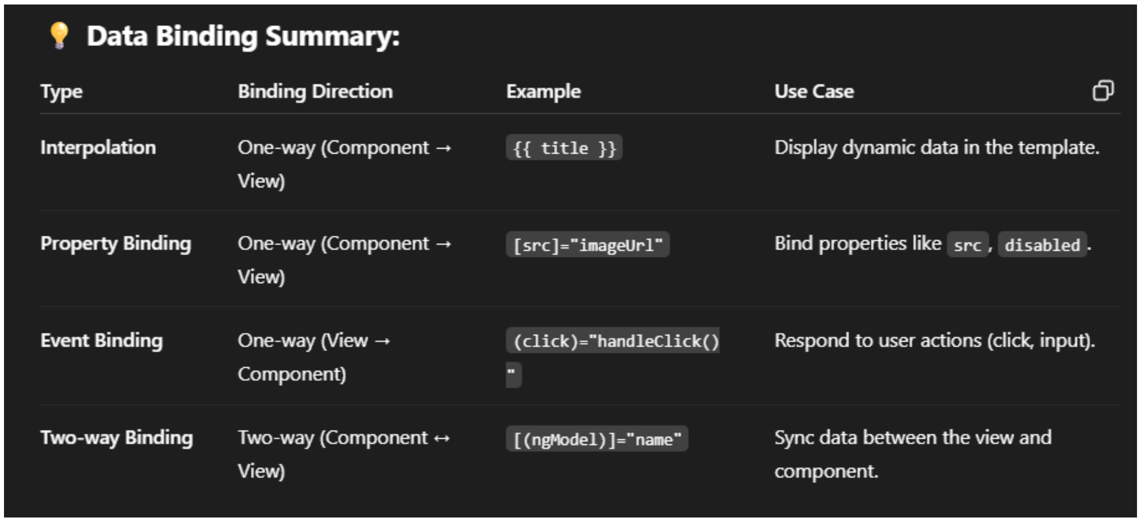


In Angular, a **template statement** is the code you write in the HTML template to respond to user actions (like clicks, input changes, form submissions, etc.). It's usually placed inside event bindings such as `(click)`, `(input)`, `(submit)`, etc.

✅ Common Syntax

```html
<button (click)="sayHello()">Click Me</button>
<input (input)="onInputChange($event)">
<form (ngSubmit)="submitForm()">...</form>
```

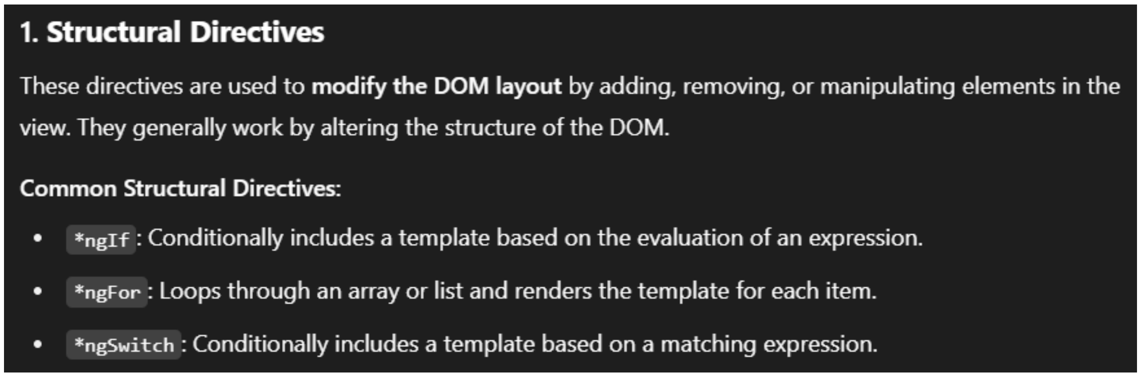
DATA BINDING



💡 Data Binding Summary:

Type	Binding Direction	Example	Use Case
Interpolation	One-way (Component → View)	`{{ title }}`	Display dynamic data in the template.
Property Binding	One-way (Component → View)	`[src]="imageUrl"`	Bind properties like `src`, `disabled`.
Event Binding	One-way (View → Component)	`(click)="handleClick()"`	Respond to user actions (click, input).
Two-way Binding	Two-way (Component ↔ View)	`[(ngModel)]="name"`	Sync data between the view and component.

DIRECTIVES

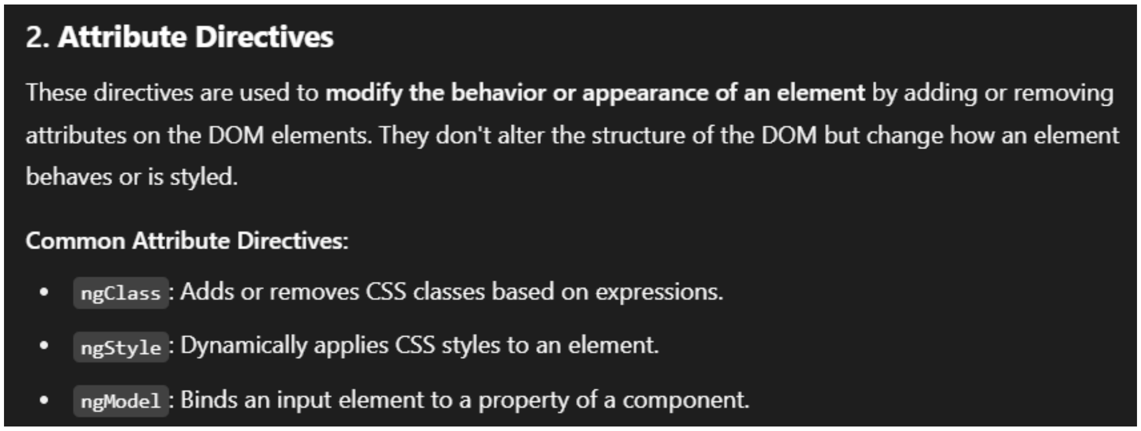


1. Structural Directives

These directives are used to **modify the DOM layout** by adding, removing, or manipulating elements in the view. They generally work by altering the structure of the DOM.

Common Structural Directives:

- `*ngIf`: Conditionally includes a template based on the evaluation of an expression.
- `*ngFor`: Loops through an array or list and renders the template for each item.
- `*ngSwitch`: Conditionally includes a template based on a matching expression.

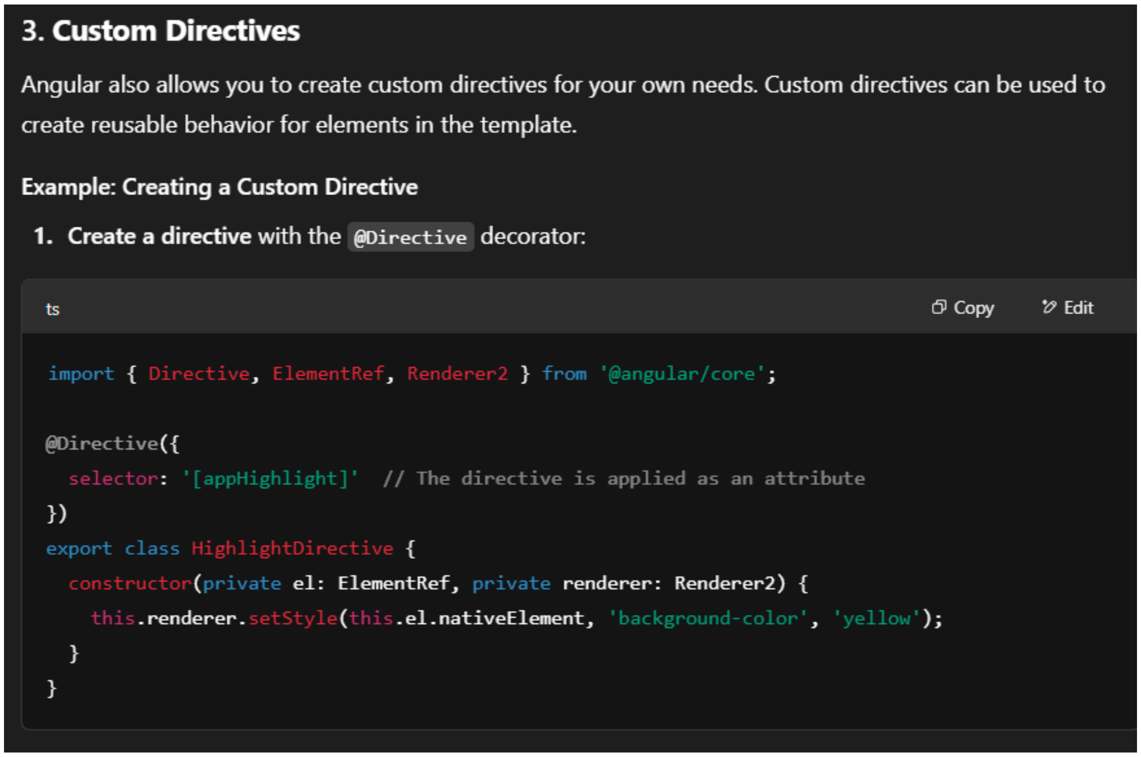


2. Attribute Directives

These directives are used to **modify the behavior or appearance of an element** by adding or removing attributes on the DOM elements. They don't alter the structure of the DOM but change how an element behaves or is styled.

Common Attribute Directives:

- `ngClass`: Adds or removes CSS classes based on expressions.
- `ngStyle`: Dynamically applies CSS styles to an element.
- `ngModel`: Binds an input element to a property of a component.



3. Custom Directives

Angular also allows you to create custom directives for your own needs. Custom directives can be used to create reusable behavior for elements in the template.

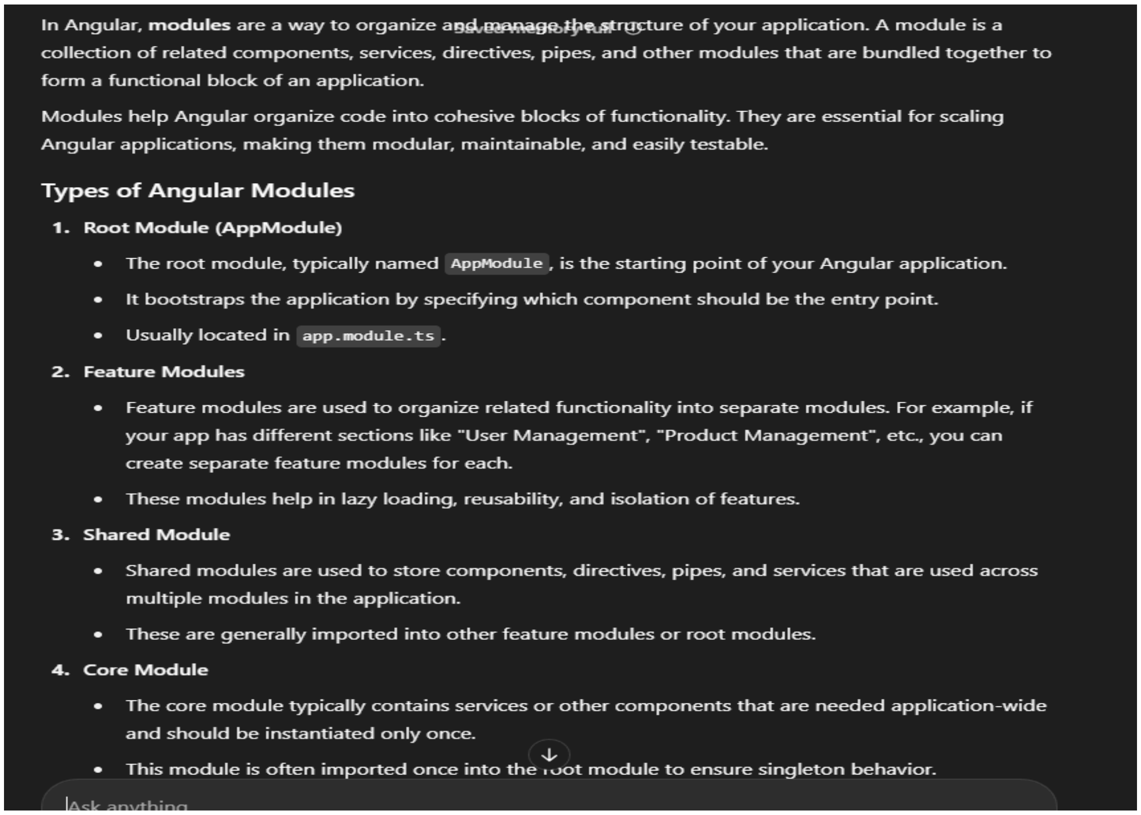
Example: Creating a Custom Directive

1. **Create a directive** with the `@Directive` decorator:

```ts
import { Directive, ElementRef, Renderer2 } from '@angular/core';

@Directive({
  selector: '[appHighlight]'  // The directive is applied as an attribute
})
export class HighlightDirective {
  constructor(private el: ElementRef, private renderer: Renderer2) {
    this.renderer.setStyle(this.el.nativeElement, 'background-color', 'yellow');
  }
}
```

MODULE



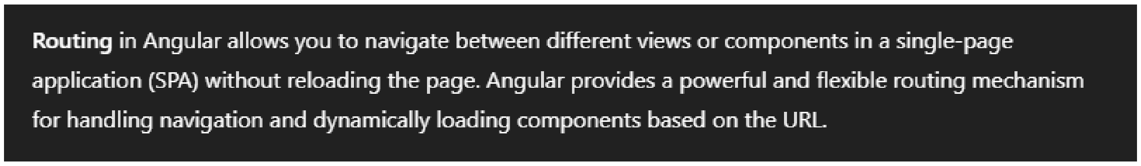
In Angular, **modules** are a way to organize and manage the structure of your application. A module is a collection of related components, services, directives, pipes, and other modules that are bundled together to form a functional block of an application.

Modules help Angular organize code into cohesive blocks of functionality. They are essential for scaling Angular applications, making them modular, maintainable, and easily testable.

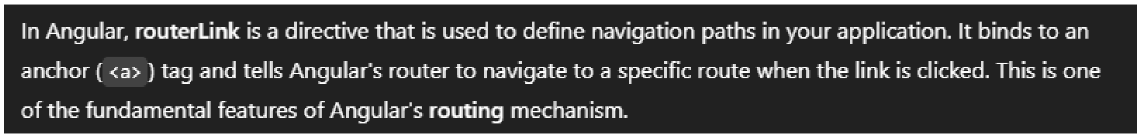
Types of Angular Modules

1. **Root Module (AppModule)**
   - The root module, typically named `AppModule`, is the starting point of your Angular application.
   - It bootstraps the application by specifying which component should be the entry point.
   - Usually located in `app.module.ts`.
2. **Feature Modules**
   - Feature modules are used to organize related functionality into separate modules. For example, if your app has different sections like "User Management", "Product Management", etc., you can create separate feature modules for each.
   - These modules help in lazy loading, reusability, and isolation of features.
3. **Shared Module**
   - Shared modules are used to store components, directives, pipes, and services that are used across multiple modules in the application.
   - These are generally imported into other feature modules or root modules.
4. **Core Module**
   - The core module typically contains services or other components that are needed application-wide and should be instantiated only once.
   - This module is often imported once into the root module to ensure singleton behavior.

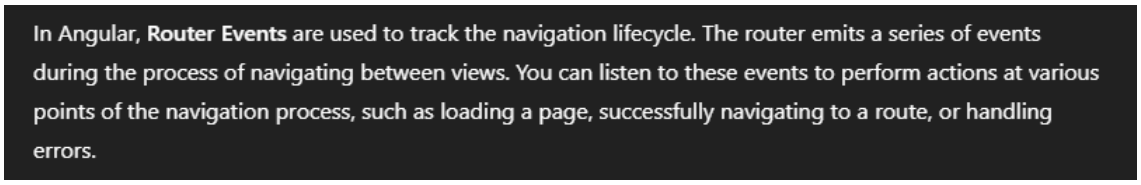
ROUTING


Routing in Angular allows you to navigate between different views or components in a single-page application (SPA) without reloading the page. Angular provides a powerful and flexible routing mechanism for handling navigation and dynamically loading components based on the URL.

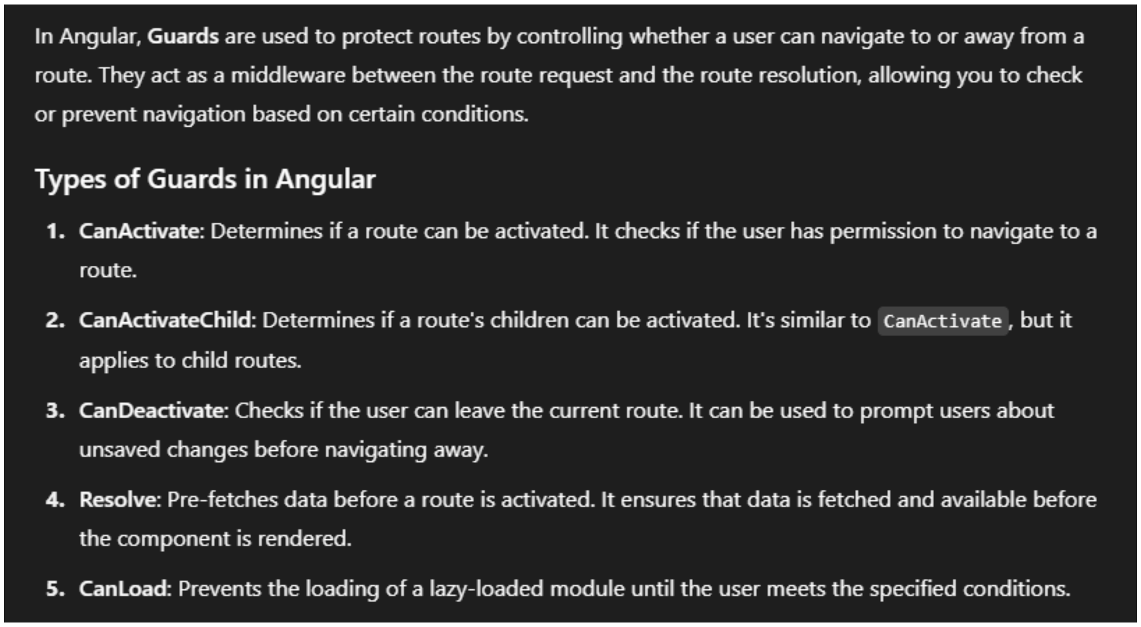
ROUTER LINKS


In Angular, **routerLink** is a directive that is used to define navigation paths in your application. It binds to an anchor (`<a>`) tag and tells Angular's router to navigate to a specific route when the link is clicked. This is one of the fundamental features of Angular's **routing** mechanism.

ROUTER EVENTS


In Angular, **Router Events** are used to track the navigation lifecycle. The router emits a series of events during the process of navigating between views. You can listen to these events to perform actions at various points of the navigation process, such as loading a page, successfully navigating to a route, or handling errors.

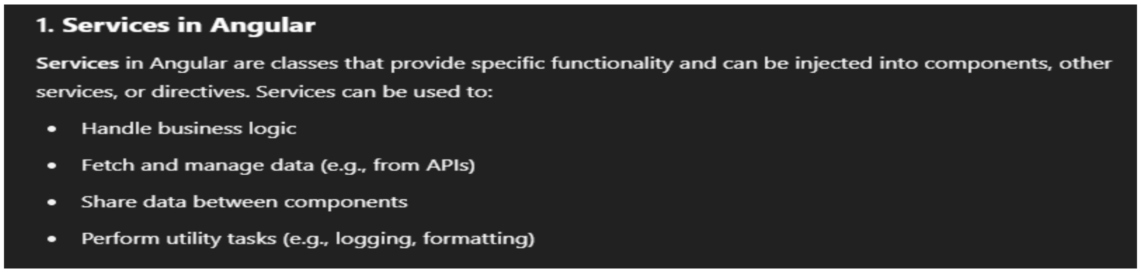
GUARDS


In Angular, **Guards** are used to protect routes by controlling whether a user can navigate to or away from a route. They act as a middleware between the route request and the route resolution, allowing you to check or prevent navigation based on certain conditions.

Types of Guards in Angular

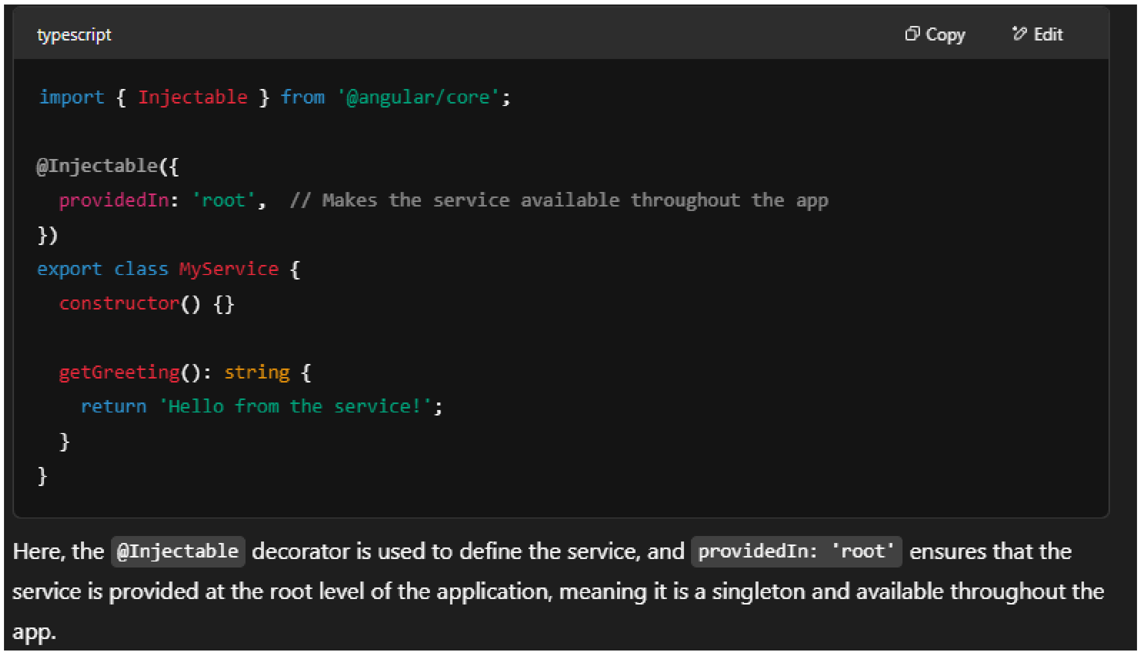
1. **CanActivate**: Determines if a route can be activated. It checks if the user has permission to navigate to a route.
2. **CanActivateChild**: Determines if a route's children can be activated. It's similar to `CanActivate`, but it applies to child routes.
3. **CanDeactivate**: Checks if the user can leave the current route. It can be used to prompt users about unsaved changes before navigating away.
4. **Resolve**: Pre-fetches data before a route is activated. It ensures that data is fetched and available before the component is rendered.
5. **CanLoad**: Prevents the loading of a lazy-loaded module until the user meets the specified conditions.

SERVICES


1. Services in Angular

Services in Angular are classes that provide specific functionality and can be injected into components, other services, or directives. Services can be used to:

- Handle business logic
- Fetch and manage data (e.g., from APIs)
- Share data between components
- Perform utility tasks (e.g., logging, formatting)



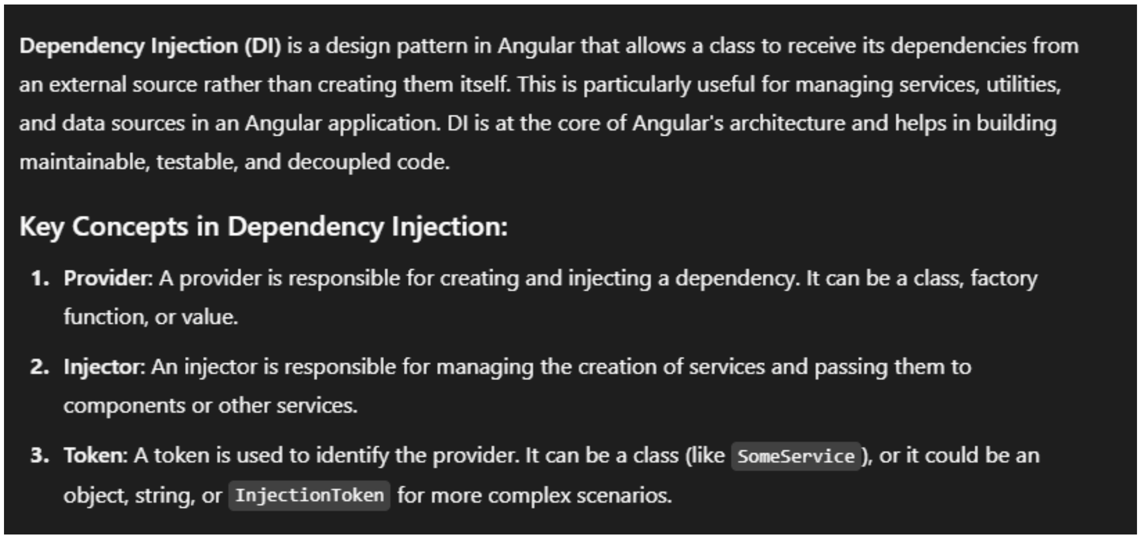
```typescript
import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'root',  // Makes the service available throughout the app
})
export class MyService {
  constructor() {}

  getGreeting(): string {
    return 'Hello from the service!';
  }
}
```

Here, the `@Injectable` decorator is used to define the service, and `providedIn: 'root'` ensures that the service is provided at the root level of the application, meaning it is a singleton and available throughout the app.

DEPENDENCY INJECTION

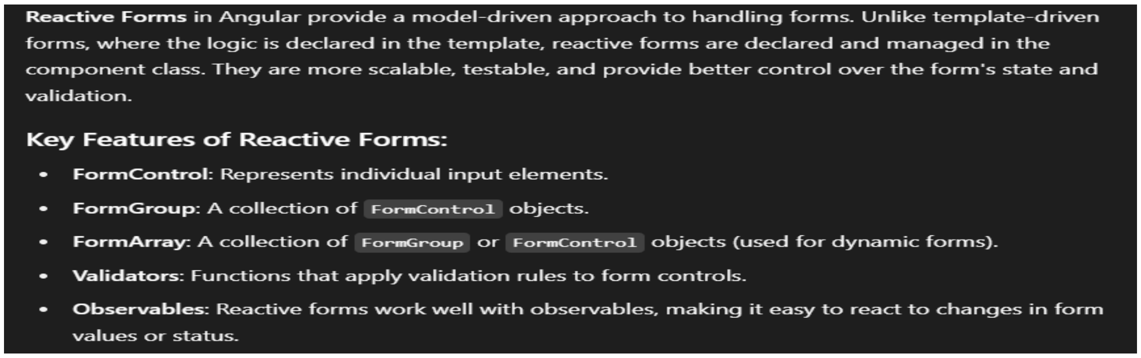


Dependency Injection (DI) is a design pattern in Angular that allows a class to receive its dependencies from an external source rather than creating them itself. This is particularly useful for managing services, utilities, and data sources in an Angular application. DI is at the core of Angular's architecture and helps in building maintainable, testable, and decoupled code.

Key Concepts in Dependency Injection:

1. **Provider**: A provider is responsible for creating and injecting a dependency. It can be a class, factory function, or value.
2. **Injector**: An injector is responsible for managing the creation of services and passing them to components or other services.
3. **Token**: A token is used to identify the provider. It can be a class (like `SomeService`), or it could be an object, string, or `InjectionToken` for more complex scenarios.

REACTIVE FORMS

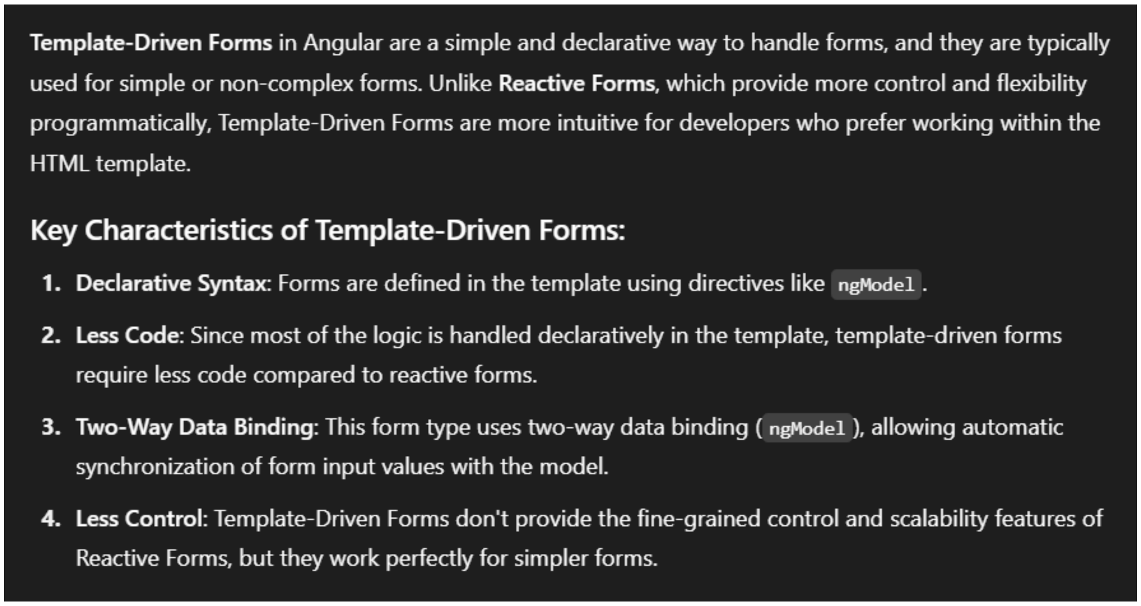


Reactive Forms in Angular provide a model-driven approach to handling forms. Unlike template-driven forms, where the logic is declared in the template, reactive forms are declared and managed in the component class. They are more scalable, testable, and provide better control over the form's state and validation.

Key Features of Reactive Forms:

- **FormControl**: Represents individual input elements.
- **FormGroup**: A collection of `FormControl` objects.
- **FormArray**: A collection of `FormGroup` or `FormControl` objects (used for dynamic forms).
- **Validators**: Functions that apply validation rules to form controls.
- **Observables**: Reactive forms work well with observables, making it easy to react to changes in form values or status.

TEMPLATE DRIVEN FORM

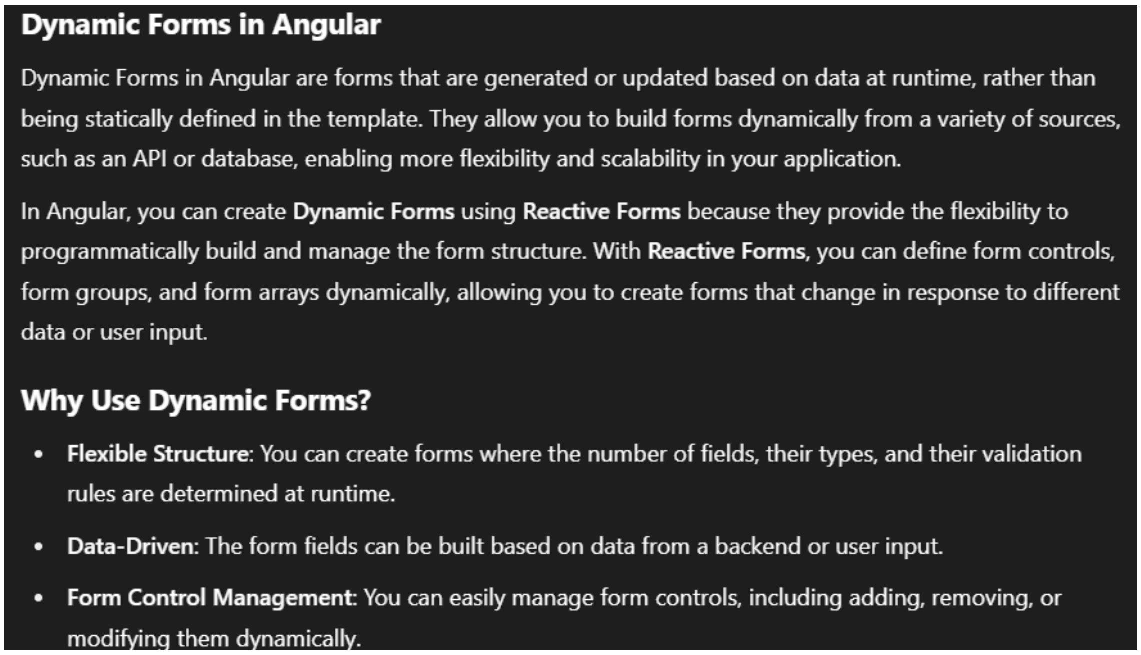


Template-Driven Forms in Angular are a simple and declarative way to handle forms, and they are typically used for simple or non-complex forms. Unlike **Reactive Forms**, which provide more control and flexibility programmatically, Template-Driven Forms are more intuitive for developers who prefer working within the HTML template.

Key Characteristics of Template-Driven Forms:

1. **Declarative Syntax**: Forms are defined in the template using directives like `ngModel`.
2. **Less Code**: Since most of the logic is handled declaratively in the template, template-driven forms require less code compared to reactive forms.
3. **Two-Way Data Binding**: This form type uses two-way data binding (`ngModel`), allowing automatic synchronization of form input values with the model.
4. **Less Control**: Template-Driven Forms don't provide the fine-grained control and scalability features of Reactive Forms, but they work perfectly for simpler forms.

DYNAMIC FORMS



Dynamic Forms in Angular

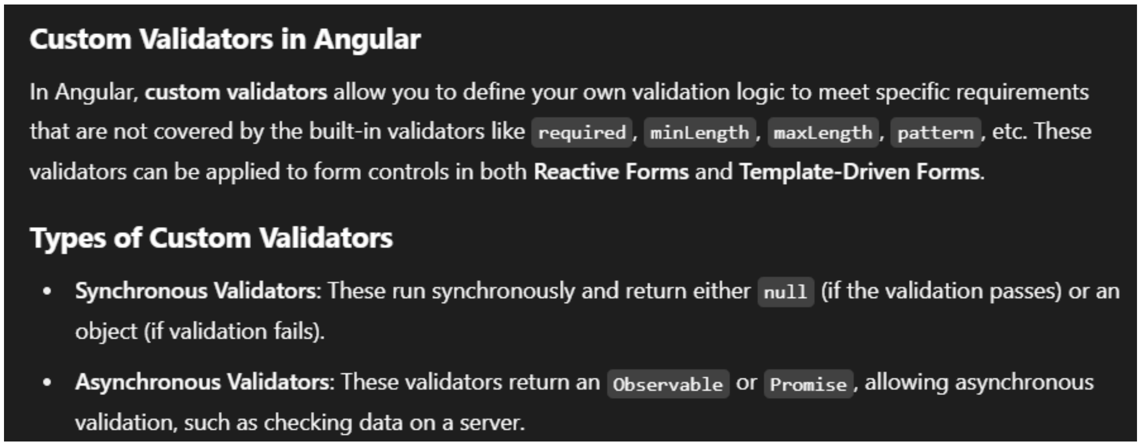
Dynamic Forms in Angular are forms that are generated or updated based on data at runtime, rather than being statically defined in the template. They allow you to build forms dynamically from a variety of sources, such as an API or database, enabling more flexibility and scalability in your application.

In Angular, you can create **Dynamic Forms** using **Reactive Forms** because they provide the flexibility to programmatically build and manage the form structure. With **Reactive Forms**, you can define form controls, form groups, and form arrays dynamically, allowing you to create forms that change in response to different data or user input.

Why Use Dynamic Forms?

- **Flexible Structure**: You can create forms where the number of fields, their types, and their validation rules are determined at runtime.
- **Data-Driven**: The form fields can be built based on data from a backend or user input.
- **Form Control Management**: You can easily manage form controls, including adding, removing, or modifying them dynamically.

CUSTOM VALIDATORS



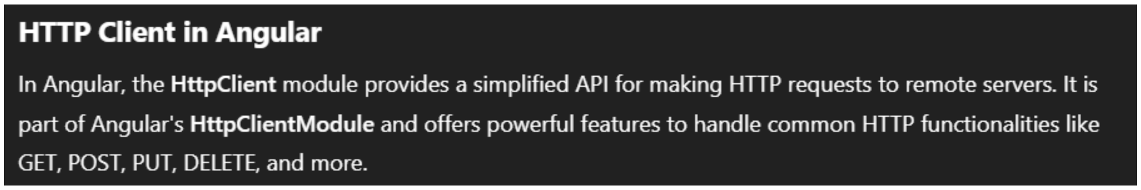
Custom Validators in Angular

In Angular, **custom validators** allow you to define your own validation logic to meet specific requirements that are not covered by the built-in validators like `required`, `minLength`, `maxLength`, `pattern`, etc. These validators can be applied to form controls in both **Reactive Forms** and **Template-Driven Forms**.

Types of Custom Validators

- **Synchronous Validators**: These run synchronously and return either `null` (if the validation passes) or an object (if validation fails).
- **Asynchronous Validators**: These validators return an `Observable` or `Promise`, allowing asynchronous validation, such as checking data on a server.

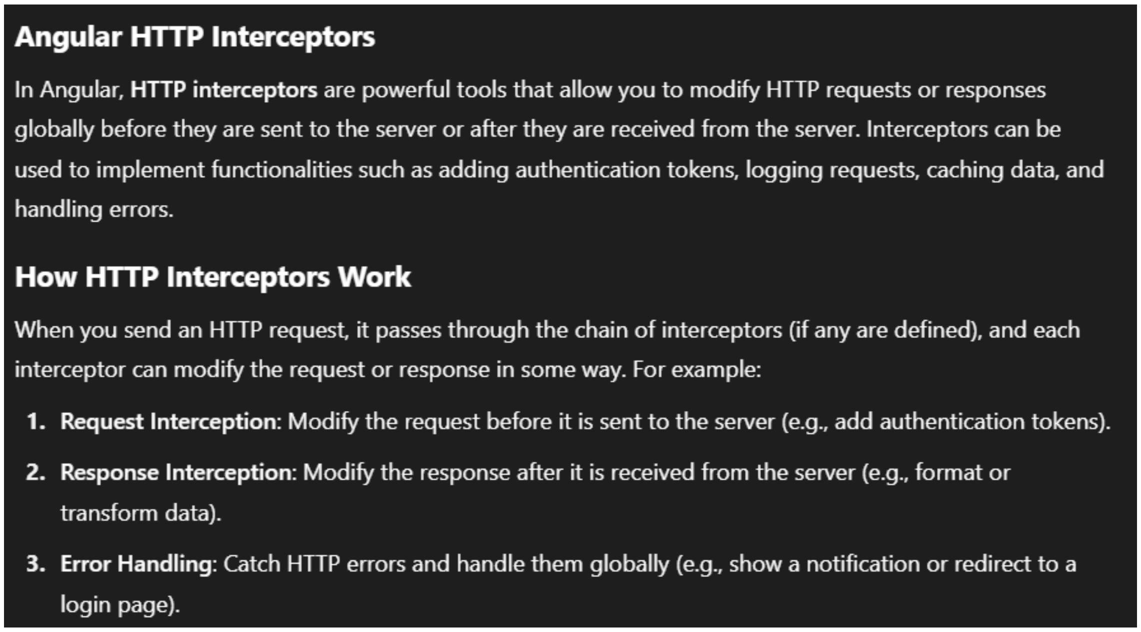
HTTP CLIENT



HTTP Client in Angular

In Angular, the **HttpClient** module provides a simplified API for making HTTP requests to remote servers. It is part of Angular's **HttpClientModule** and offers powerful features to handle common HTTP functionalities like GET, POST, PUT, DELETE, and more.

INTERCEPTORS



Angular HTTP Interceptors

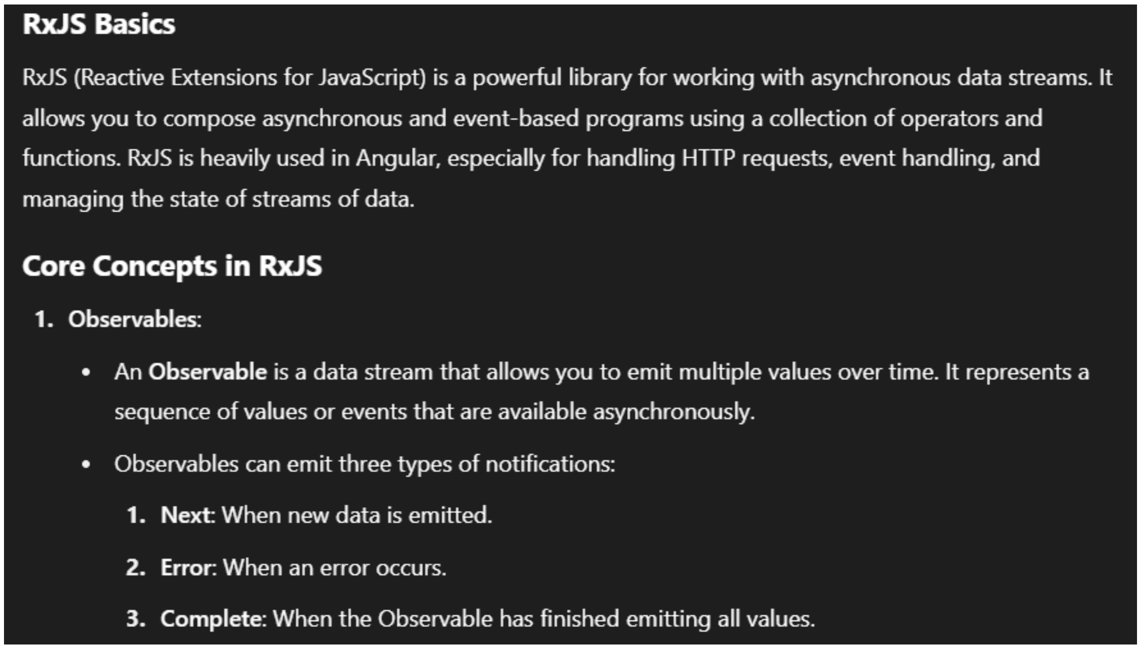
In Angular, **HTTP interceptors** are powerful tools that allow you to modify HTTP requests or responses globally before they are sent to the server or after they are received from the server. Interceptors can be used to implement functionalities such as adding authentication tokens, logging requests, caching data, and handling errors.

How HTTP Interceptors Work

When you send an HTTP request, it passes through the chain of interceptors (if any are defined), and each interceptor can modify the request or response in some way. For example:

1. **Request Interception**: Modify the request before it is sent to the server (e.g., add authentication tokens).
2. **Response Interception**: Modify the response after it is received from the server (e.g., format or transform data).
3. **Error Handling**: Catch HTTP errors and handle them globally (e.g., show a notification or redirect to a login page).

RXJS

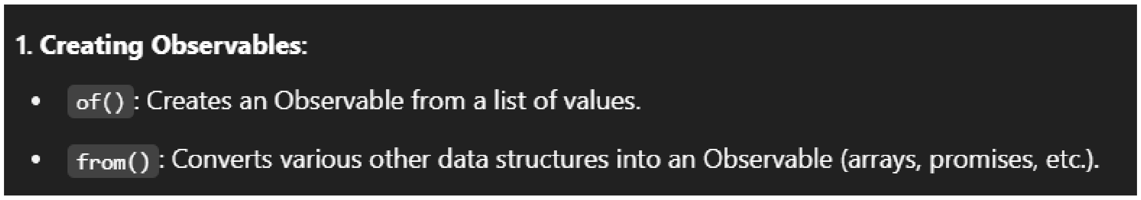


RxJS Basics

RxJS (Reactive Extensions for JavaScript) is a powerful library for working with asynchronous data streams. It allows you to compose asynchronous and event-based programs using a collection of operators and functions. RxJS is heavily used in Angular, especially for handling HTTP requests, event handling, and managing the state of streams of data.

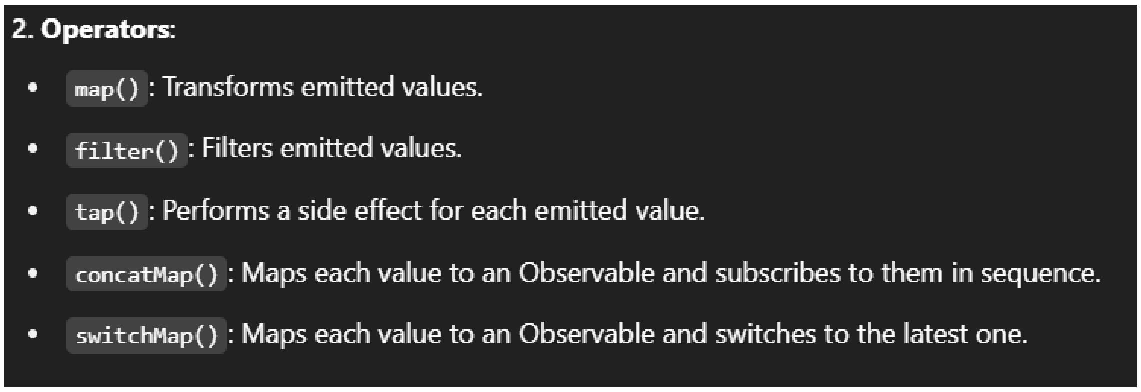
Core Concepts in RxJS

1. **Observables:**
   - An **Observable** is a data stream that allows you to emit multiple values over time. It represents a sequence of values or events that are available asynchronously.
   - Observables can emit three types of notifications:
     1. **Next:** When new data is emitted.
     2. **Error:** When an error occurs.
     3. **Complete:** When the Observable has finished emitting all values.



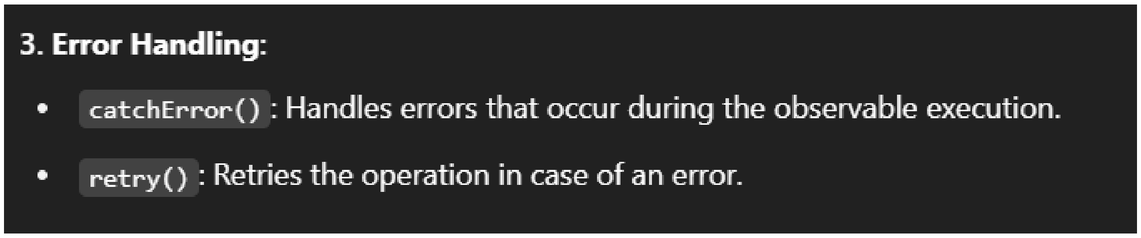
1. Creating Observables:

- `of()`: Creates an Observable from a list of values.
- `from()`: Converts various other data structures into an Observable (arrays, promises, etc.).



2. Operators:

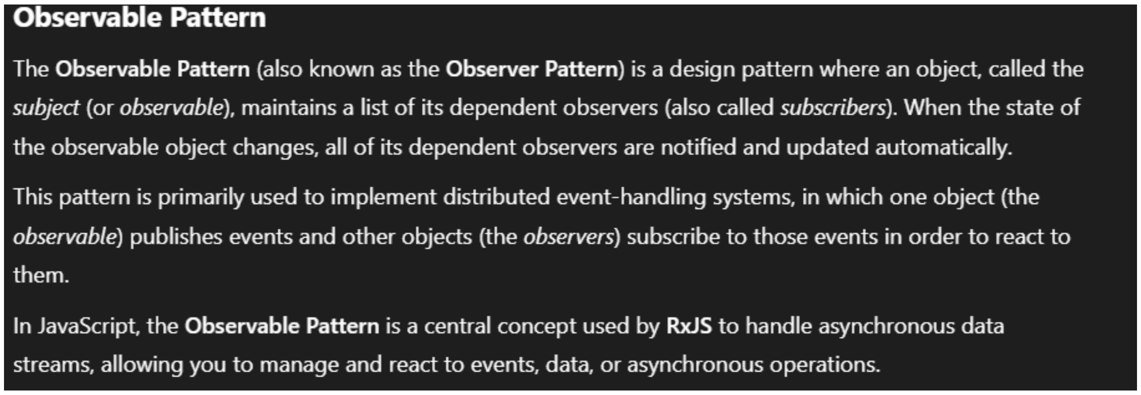
- `map()`: Transforms emitted values.
- `filter()`: Filters emitted values.
- `tap()`: Performs a side effect for each emitted value.
- `concatMap()`: Maps each value to an Observable and subscribes to them in sequence.
- `switchMap()`: Maps each value to an Observable and switches to the latest one.



3. Error Handling:

- `catchError()`: Handles errors that occur during the observable execution.
- `retry()`: Retries the operation in case of an error.

OBSERVABLE PATTERN



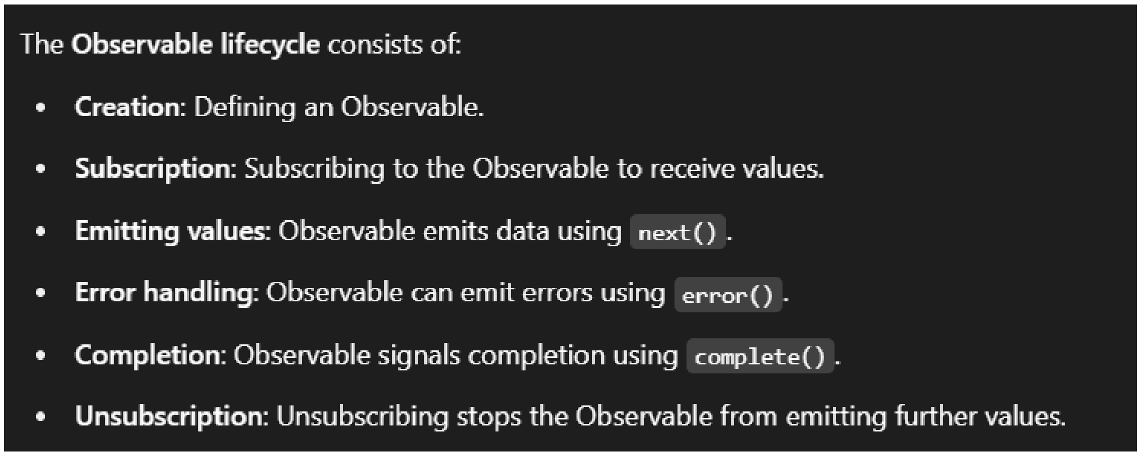
Observable Pattern

The **Observable Pattern** (also known as the **Observer Pattern**) is a design pattern where an object, called the *subject* (or *observable*), maintains a list of its dependent observers (also called *subscribers*). When the state of the observable object changes, all of its dependent observers are notified and updated automatically.

This pattern is primarily used to implement distributed event-handling systems, in which one object (the *observable*) publishes events and other objects (the *observers*) subscribe to those events in order to react to them.

In JavaScript, the **Observable Pattern** is a central concept used by **RxJS** to handle asynchronous data streams, allowing you to manage and react to events, data, or asynchronous operations.

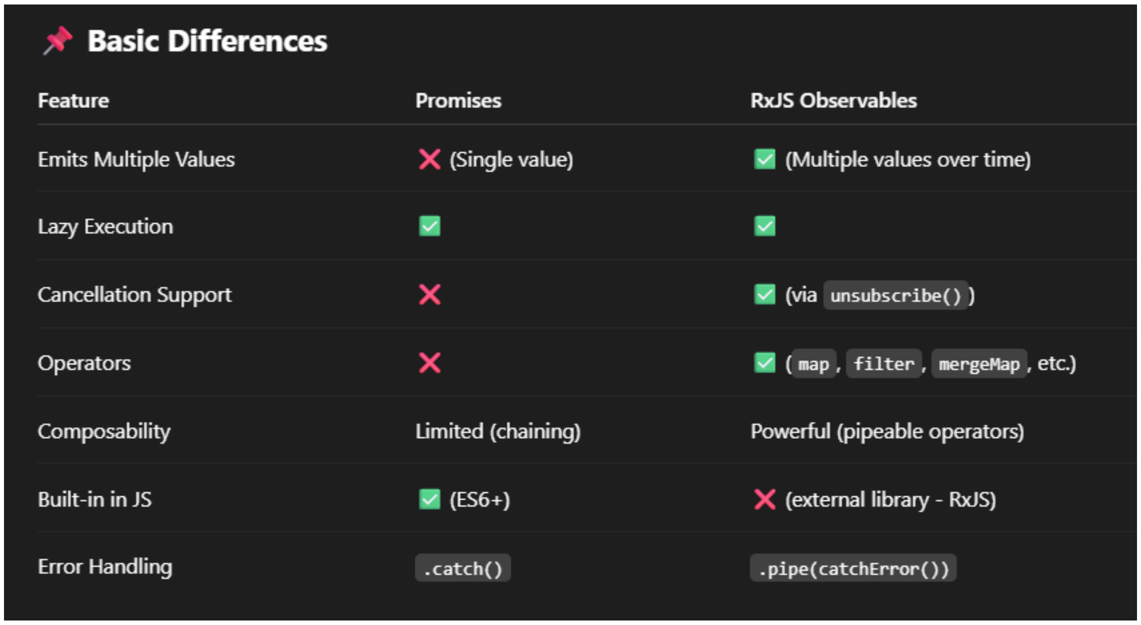
OBSERVABLE LIFECYCLE



The **Observable lifecycle** consists of:

- **Creation**: Defining an Observable.
- **Subscription**: Subscribing to the Observable to receive values.
- **Emitting values:** Observable emits data using `next()`.
- **Error handling**: Observable can emit errors using `error()`.
- **Completion**: Observable signals completion using `complete()`.
- **Unsubscription**: Unsubscribing stops the Observable from emitting further values.

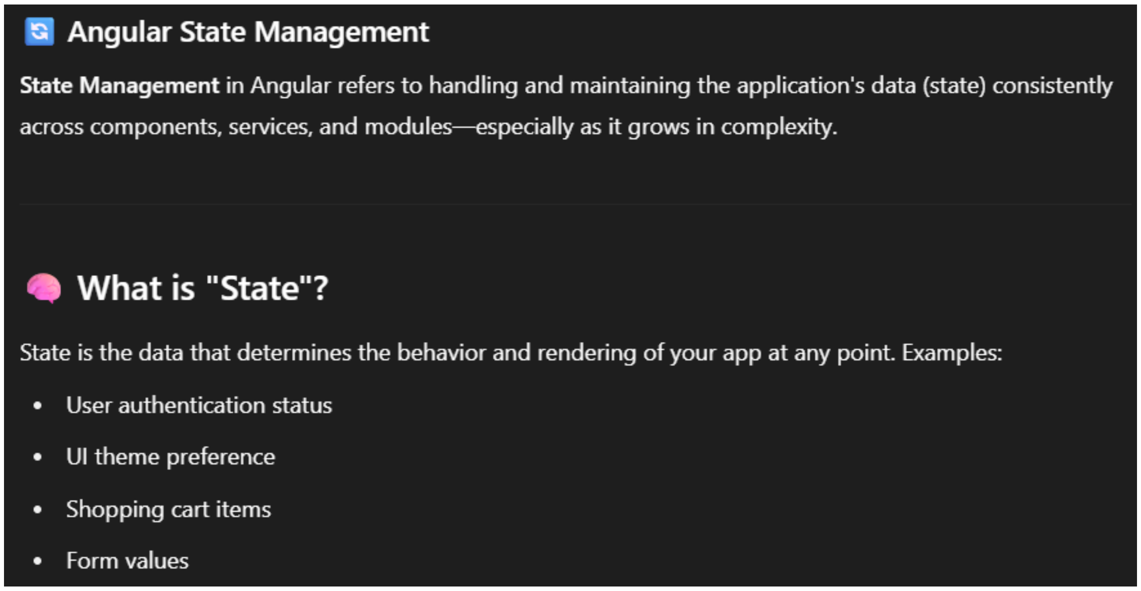
RXJS VS PROMISES



📌 Basic Differences

Feature	Promises	RxJS Observables
Emits Multiple Values	❌ (Single value)	✅ (Multiple values over time)
Lazy Execution	✅	✅
Cancellation Support	❌	✅ (via `unsubscribe()`)
Operators	❌	✅ (`map`, `filter`, `mergeMap`, etc.)
Composability	Limited (chaining)	Powerful (pipeable operators)
Built-in in JS	✅ (ES6+)	❌ (external library - RxJS)
Error Handling	`.catch()`	`.pipe(catchError())`

STATE MANAGEMENT



🔄 Angular State Management

State Management in Angular refers to handling and maintaining the application's data (state) consistently across components, services, and modules—especially as it grows in complexity.

🧠 What is "State"?

State is the data that determines the behavior and rendering of your app at any point. Examples:

- User authentication status
- UI theme preference
- Shopping cart items
- Form values