Angular 19 Development

Instructor: suryakant.surve@vinsys.com

Frontend Frameworks/Libraries

Angular

- **Type**: Framework (Client-Side, UI development in Browser)
- **Language**: TypeScript
- **Use**: Build complex, large-scale applications with built-in solutions (routing, state management, HTTP handling).

React

- **Type**: Library (Focused on building UI Components)
- **Language**: JavaScript (with JSX)
- **Use**: Create fast and dynamic single-page applications (SPAs).

Vue.js

- **Type**: Framework
- **Language**: JavaScript
- **Use**: Simple and flexible UI building with a gentle learning curve.

Web Technologies

HTML (HyperText Markup Language)

- **Purpose**: Structure of a web page.
- **Example**:

```
<h1>Welcome to My Website</h1>
<p>This is a paragraph.</p>
```

CSS (Cascading Style Sheets)

- **Purpose**: Styling and layout of web pages.
- **Example**:

```
h1 {
  color: blue;
  text-align: center;
```

```
}
```

JavaScript (JS)

- **Purpose**: Scripting language to add interactivity.
- **Example**:

```
document.getElementById('btn').addEventListener('click', function() {
  alert('Button clicked!');
});
```

Browser Vendors

- **Google Chrome** – Built by Google
- **MS Edge** – Built by Microsoft (Chromium-based)
- **Mozilla Firefox** – Built by Mozilla Foundation

Organizations

W3C (World Wide Web Consortium)

- **Purpose**: Defines Web Standards (HTML, CSS, etc.)

ECMA (European Computer Manufacturers Association)

- **Purpose**: Standardizes scripting languages (ECMAScript → JavaScript standard)

JavaScript Engines

- **V8** – Used in Google Chrome
- **Chakra** – Used in Microsoft Edge (Legacy Edge, now Chromium-based uses V8)
- **SpiderMonkey** – Used in Mozilla Firefox

Node.js

- **What**: A runtime environment to run JavaScript outside of the browser.
- **Use**: Server-side development (build backend apps, APIs).

ES6 (ECMAScript 2015) Features

let and const

- **let**: Block-scoped variable
- **const**: Block-scoped, cannot be reassigned
- **Example**:

```
let name = 'John';
const age = 30;
```

Array Destructuring

- **Example**:

```
const [a, b] = [1, 2];
console.log(a); // 1
```

Object De-structuring

- **Example**:

```
const person = { name: 'John', age: 30 };
const { name, age } = person;
console.log(name); // John
```

Classes & Objects

- **Example**:

```
class Person {
  constructor(name) {
    this.name = name;
  }
  greet() {
    console.log(`Hello, ${this.name}`);
  }
}

const p = new Person('Alice');
p.greet(); // Hello, Alice
```

Spread Operator (...)

- **Example**:

```
const arr1 = [1, 2];
const arr2 = [...arr1, 3, 4];
console.log(arr2); // [1, 2, 3, 4]
```

Rest Parameter

- **Example**:

```
function sum(...numbers) {
  return numbers.reduce((a, b) => a + b);
}

console.log(sum(1, 2, 3)); // 6
```

Default Parameters

- **Example**:

```
function greet(name = 'Guest') {
  console.log(`Hello, ${name}`);
}

greet(); // Hello, Guest
```

forEach (Array method)

- **Example**:

```
 [1, 2, 3].forEach(num => console.log(num));
```

Template Literals

- **Example**:

```
const name = 'John';
console.log(`Hello, ${name}!`);
```

Programming Concepts

Synchronous vs Asynchronous Programming

- **Synchronous**: Code runs sequentially, one after another.
- **Asynchronous**: Code can run independently, without blocking execution.

Async Functions in JS

- **Example**:

```
async function fetchData() {
  return 'Data fetched';
```

```
}

fetchData().then(console.log);
```

Callback

- **Example**:

```
function greet(name, callback) {
  console.log('Hello ' + name);
  callback();
}

greet('John', function() {
  console.log('Callback executed');
});
```

Promise

- **Example**:

```
const promise = new Promise((resolve, reject) => {
  resolve('Success!');
});

promise.then(result => console.log(result));
```

Async/Await

- **Example**:

```
async function fetchData() {
  let result = await Promise.resolve('Fetched Data');
  console.log(result);
}

fetchData();
```

NPM (Node Package Manager)

◆ **npm init**

Purpose: Initializes a new Node.js project by creating a package.json file.

Usage:

npm init

You'll be prompted for:

- name: Project name
- version: Initial version
- description: Description of your project
- entry point: Main file (e.g., index.js)
- author, license, etc.

Example:

npm init

Creates:

```
{
  "name": "my-project",
  "version": "1.0.0",
  "description": "A sample project",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Your Name",
  "license": "ISC"
}
```

npm install

Purpose: Installs dependencies from the npm registry.

Usage:

npm install <package-name>

Types:

- **Local install** (default): adds to node_modules and updates package.json.
- **Global install** (-g): installs package system-wide.

Examples:

```
npm install express        # Installs Express.js locally
npm install -g typescript  # Installs TypeScript globally
```

package.json

Purpose: Manages project metadata and dependencies.

Sections:

- name, version: Basic info
- scripts: Custom CLI commands
- dependencies: Runtime packages
- devDependencies: Development-only packages

Example:

```
{
  "name": "ts-app",
  "version": "1.0.0",
  "main": "index.js",
  "scripts": {
    "start": "node index.js",
    "build": "tsc"
  },
  "dependencies": {
    "express": "^4.18.2"
  },
  "devDependencies": {
    "typescript": "^5.0.0"
  }
}
```

TypeScript

Introduction

TypeScript is a **typed superset of JavaScript** that compiles to plain JavaScript.

Benefits:

- Optional static typing
- Better tooling (e.g., IntelliSense)
- Modern features like interfaces, enums, access modifiers, etc.

Installing TypeScript

```
npm install -g typescript
```

To check version:

```
tsc --version
```

Compiling Your First TypeScript Program

1. Create a file:

```
// hello.ts
let message: string = "Hello, TypeScript!";
console.log(message);
```

2. Compile:

```
tsc hello.ts
```

3. Resulting hello.js:

```
var message = "Hello, TypeScript!";
console.log(message);
```

Basic Types

Type	Example
string	let name: string = "Alice"
number	let age: number = 30
boolean	let isOpen: boolean = true
any	let data: any = "Could be anything"
array	let scores: number[] = [90, 80]
tuple	let user: [string, number] = ["Bob", 25]
enum	See below

Example with enum:

```
enum Direction {
  Up,
  Down,
  Left,
  Right
}
let move: Direction = Direction.Up;
```

Boolean

```
let isDone: boolean = false;

if (!isDone) {
  console.log("Task is not done.");
```

```
}
```

Loops and Iterators

For loop:

```
for (let i = 0; i < 5; i++) {
  console.log(i);
}
```

For-of (arrays):

```
let colors = ['red', 'green', 'blue'];
for (let color of colors) {
  console.log(color);
}
```

For-in (objects):

```
let person = { name: "Tom", age: 30 };
for (let key in person) {
  console.log(`${key}: ${person[key as keyof typeof person]}`);
}
```

Interfaces

Purpose: Defines a structure or contract that an object must follow.

```
interface Person {
  name: string;
  age: number;
  greet(): void;
}

let user: Person = {
  name: "Alice",
  age: 25,
  greet() {
    console.log(`Hi, I'm ${this.name}`);
  }
};

user.greet(); // Output: Hi, I'm Alice
```

Classes

```
class Animal {
  name: string;

  constructor(name: string) {
```

```
    this.name = name;
  }

  move(distance: number = 0) {
    console.log(`${this.name} moved ${distance}m.`);
  }
}

let dog = new Animal("Dog");
dog.move(10);
```

Inheritance

```
class Animal {
  constructor(public name: string) {}

  move(distance: number = 0) {
    console.log(`${this.name} moved ${distance}m.`);
  }
}

class Dog extends Animal {
  bark() {
    console.log("Woof! Woof!");
  }
}

let puppy = new Dog("Buddy");
puppy.bark();        // Woof! Woof!
puppy.move(15);      // Buddy moved 15m.
```

Reactive Programming Basics (RxJS)

Introduction

Reactive programming is a programming paradigm focused on asynchronous data streams and the propagation of change. RxJS (Reactive Extensions for JavaScript) is a library for reactive programming using **Observables** to make it easier to compose asynchronous or callback-based code.

Use cases:

- Real-time data updates
- Event handling
- HTTP requests and responses
- WebSocket communication

Streams

A stream is a sequence of ongoing events ordered in time, like data pushed to you over time.

Example:

```
const stream = Rx.Observable.of(1, 2, 3);
stream.subscribe(x => console.log(x)); // 1, 2, 3
```

Observables

Observables are core building blocks in RxJS. They represent a collection of future values or events.

Example:

```
import { Observable } from 'rxjs';

const observable = new Observable(subscriber => {
  subscriber.next('Hello');
  subscriber.next('World');
  subscriber.complete();
});

observable.subscribe({
  next(x) { console.log(x); },
  complete() { console.log('Done'); }
});
```

Subscription

A subscription is an execution of an Observable. It is used to start receiving notifications.

Example:

```
const subscription = observable.subscribe(x => console.log(x));
subscription.unsubscribe(); // Stops receiving values
```

Angular

Introduction

Angular is a platform and framework for building client-side applications using HTML, CSS, and TypeScript. It is developed and maintained by *Google*.

Single Page Applications (SPA) vs Multi Page Applications (MPA)

Feature	SPA	MPA
Page Reload	No (uses routing)	Yes (each page is a new load)
Speed	Fast after initial load	Slower
Technology	Angular, React, Vue	Traditional HTML, PHP, JSP

Why Angular?

- Two-way data binding
- Dependency Injection
- Component-based architecture
- TypeScript support
- Built-in RxJS support
- Powerful CLI tools

Understanding Angular Versions

- Angular 1.x: Also known as AngularJS (uses controllers and $scope).
- Angular 2+: Complete rewrite using TypeScript, component-based.
- Latest Angular: Regular updates, Ivy rendering engine, improved performance.

MVC vs MVVM

Concept	MVC	MVVM
Pattern	Model-View-Controller	Model-View-ViewModel
Controller/VM	Handles UI logic	ViewModel binds UI and logic
Angular?	Initially MVC (AngularJS)	Now closer to MVVM via bindings

---

Angular CLI

Angular CLI is a command-line tool to scaffold, build, and maintain Angular applications.

Installing Angular CLI

Command:

npm install -g @angular/cli

Verify installation:

ng version

Components

How to Create Component?

You can manually create a component or use the Angular CLI:

Manual Steps:

- Create a TypeScript file for the class
- Create HTML, CSS, and add metadata

ng g c

Command to generate a component:

```
ng generate component component-name
# or
ng g c component-name
```

This will generate:

- .ts file (logic)
- .html file (template)
- .css file (styles)
- .spec.ts file (unit test)

Component Metadata

Defines how Angular should process, instantiate, and use the component.

Example:

```
@Component({
  selector: 'app-hello',
  templateUrl: './hello.component.html',
  styleUrls: ['./hello.component.css']
})
```

Component Context

It refers to the data and methods within a component class which are available to the template.

Example:

```
export class HelloComponent {
  message = "Hello Angular!";
}
```

HTML:

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

Component Communication

1. **Parent to Child (Input):**

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

2. **Child to Parent (Output):**

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

Component Lifecycle

Angular lifecycle hooks:

Hook	Description
ngOnInit()	Called once after component is created
ngOnChanges()	Called when input-bound props change
ngOnDestroy()	Called before the component is destroyed

Example:

```
ngOnInit() {
  console.log("Component Initialized");
}
```

Component Style

Each component can have its own scoped CSS file.

Example (hello.component.css):

```
p {
```

```
  color: blue;
}
```

Data Binding

1. **Interpolation:**

```
{{ title }}
```

2. **Property Binding:**

```
<img [src]="imageUrl">
```

3. **Event Binding:**

```
<button (click)="onClick()">Click</button>
```

4. **Two-Way Binding:**

```
<input [(ngModel)]="username">
```

Templates and Views

◆ Syntax

Angular templates use a **combination of standard HTML and Angular-specific syntax** (like {{ }}, [], (), *, etc.).

```
<h1>{{ title }}</h1>  <!-- Interpolation -->
<img [src]="imageUrl"> <!-- Property Binding -->
<button (click)="onClick()">Click</button> <!-- Event Binding -->
```

◆ HTML in Templates

You can use regular HTML tags inside Angular templates, enhanced with Angular directives and binding syntax.

```
<div>
  <h2>Welcome to {{ appName }}</h2>
  <ul>
    <li *ngFor="let user of users">{{ user.name }}</li>
  </ul>
</div>
```

In-line Templates

Define the template inside the component using template property.

```
@Component({
  selector: 'app-inline',
  template: `<h3>Inline Template Example - {{ title }}</h3>`
})
export class InlineComponent {
  title = 'Angular';
}
```

Interpolation

Use {{ expression }} to bind data from the component to the template.

```
<p>Hello, {{ userName }}!</p>
<p>2 + 2 = {{ 2 + 2 }}</p>
```

Binding Syntax

Type	Syntax	Example
Interpolation	{{ }}	{{ title }}
Property Binding	[prop]	[src]="imageUrl"
Event Binding	(event)	(click)="onClick()"

Attribute, Class, and Style Bindings

Attribute Binding:

```
<button [attr.aria-label]="label">Click</button>
```

Class Binding:

```
<div [class.active]="isActive">Toggle class</div>
```

Style Binding:

```
<p [style.color]="isRed ? 'red' : 'blue'">Styled Text</p>
```

Property Binding

Binds a DOM property to a component property.

```
<input [value]="name">
```

Template Input Variables

Defined using let-variableName. Common in structural directives like *ngFor.

```
<div *ngFor="let user of users">
  {{ user.name }}
</div>
```

Events

Event Binding

Respond to DOM events by updating the component.

```
<button (click)="onClick()">Click me</button>
```

$event Object

Passes the native event object.

```
<input (keyup)="onKeyUp($event)">

onKeyUp(event: KeyboardEvent) {
  console.log((<HTMLInputElement>event.target).value);
}
```

Using Template Reference Variable

Used to refer to a DOM element in a template.

```
<input #inputBox>
<button (click)="log(inputBox.value)">Log</button>

log(value: string) {
  console.log(value);
}
```

Directives

Structural Directives / Control Flow

These change the structure of the DOM.

- *ngIf
- *ngFor
- *ngSwitch

```
<p *ngIf="isLoggedIn">Welcome back!</p>

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

Built-in Directives

1. **ngIf** – conditionally add/remove element

   ```
   <p *ngIf="show">Hello</p>
   ```

2. **ngFor** – iterate over a collection

   ```
   <div *ngFor="let user of users">{{ user.name }}</div>
   ```

3. **ngClass** – dynamically set CSS classes

   ```
   <div [ngClass]="{ 'active': isActive }"></div>
   ```

4. **ngStyle** – dynamically set styles

   ```
   <div [ngStyle]="{ 'color': isRed ? 'red' : 'blue' }"></div>
   ```

Custom Directives

Create your own directive using @Directive.

```
@Directive({
  selector: '[appHighlight]'
})
export class HighlightDirective {
  constructor(el: ElementRef) {
    el.nativeElement.style.backgroundColor = 'yellow';
  }
}
```

Use it:

```
<p appHighlight>This text is highlighted</p>
```

Pipes

Built-In Pipes

Pipe

date
uppercase
lowercase
currency
percent

Parameterizing a Pipe

```
<!-- date pipe with parameter -->
<p>{{ today | date:'fullDate' }}</p>

<!-- currency pipe -->
<p>{{ cost | currency:'EUR':'symbol':'1.2-2' }}</p>
```

Chaining Pipes

```
<p>{{ name | lowercase | titlecase }}</p>
```

Component Interaction

1. Pass Data from Parent to Child with @Input() Binding

Description: Passes data from a parent component to a child component using @Input() decorator.

Example:

parent.component.ts

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

child.component.ts

```
@Component({
  selector: 'app-child',
  template: `<p>{{ childData }}</p>`
```

```
})
export class ChildComponent {
  @Input() childData: string;
}
```

2. ngOnChanges()

Description: Lifecycle hook called when any data-bound property of a directive changes.

Example:

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

  ngOnChanges(changes: SimpleChanges) {
    console.log('Changes detected:', changes);
  }
}
```

3. Child to Parent Communication

Using @Output() and EventEmitter:

Example:

child.component.ts

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

  sendMessage() {
    this.messageEvent.emit('Hello from Child!');
  }
}
```

parent.component.ts

```
@Component({
  selector: 'app-parent',
  template: `<app-child (messageEvent)="receiveMessage($event)"></app-child>`
```

```
})
export class ParentComponent {
  receiveMessage(msg: string) {
    console.log('Received:', msg);
  }
}
```

Services

1. Dependency Injection (DI)

Description: Angular injects required services into components/constructors.

```
constructor(private myService: MyService) { }
```

2. Angular Injector System

Core Concept: Hierarchical system that resolves dependencies using injectors (root, component-level, module-level).

3. DI Providers

Description: Providers define how to create a service. They are registered in:

- @Injectable({ providedIn: 'root' }) (recommended)
- providers array in modules or components

4. Using a Service to Access Data

my-data.service.ts

```
@Injectable({
  providedIn: 'root'
})
export class MyDataService {
  getData() {
    return ['One', 'Two', 'Three'];
  }
}
```

component.ts

```
constructor(private dataService: MyDataService) {}

ngOnInit() {
  const data = this.dataService.getData();
}
```

5. Injecting Service

Handled by Angular automatically via constructor injection.

6. Async Services

Description: Services can return data asynchronously using Promise or Observable.

7. Using Promise with Services

```
getData(): Promise<string> {
  return new Promise(resolve => {
    setTimeout(() => resolve('Data loaded'), 2000);
  });
}
```

8. Using Observables with Services

```
getObservableData(): Observable<string> {
  return of('Observable Data');
}
```

In Component:

```
this.myService.getObservableData().subscribe(data => console.log(data));
```

Routermodule

1. Importing the RouterModule and Routes

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

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

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

2. Configuring Routes

As shown above: Define paths and link to components.

3. Displaying Components using a RouterOutlet

In app.component.html:

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

4. Navigating with RouterLink

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

5. Accessing Parameters using ActivatedRoute

Route Definition:

```
{ path: 'user/:id', component: UserComponent }
```

In Component:

```
constructor(private route: ActivatedRoute) {}

ngOnInit() {
  const userId = this.route.snapshot.paramMap.get('id');
}
```

Creating Custom Modules

Generate a Module

```
ng g m my-module
```

This command creates:

src/app/my-module/my-module.module.ts

Use the Module

Import it into your AppModule or another parent module:

```
import { MyModule } from './my-module/my-module.module';

@NgModule({
  imports: [MyModule]
})
```

FormsModule - Template-driven Forms

What is FormsModule?

FormsModule is part of Angular that enables **template-driven forms**. These forms are declared and managed directly in the HTML template using Angular directives.

How to Import

```
import { FormsModule } from '@angular/forms';

@NgModule({
  imports: [FormsModule],
})
export class AppModule {}
```

Template-driven Forms

Template-driven forms are used when you want to keep your form logic in the **HTML template** rather than in the component class.

Example:

```
<!-- app.component.html -->
<form #userForm="ngForm" (ngSubmit)="onSubmit(userForm)">
  <input name="username" [(ngModel)]="user.username" required />
  <input type="email" name="email" [(ngModel)]="user.email" required email />
  <button type="submit">Submit</button>
</form>
ts
CopyEdit
// app.component.ts
export class AppComponent {
  user = { username: '', email: '' };

  onSubmit(form: NgForm) {
    console.log(form.value); // { username: '...', email: '...' }
  }
}
```

Form Validation

Angular provides built-in validators like required, email, minlength, etc., and allows for custom validators.

Built-in Validation Example

```
<input name="email" [(ngModel)]="email" required email #emailRef="ngModel" />
<div *ngIf="emailRef.invalid && emailRef.touched">
  <small *ngIf="emailRef.errors?.required">Email is required.</small>
  <small *ngIf="emailRef.errors?.email">Invalid email format.</small>
</div>
```

ngForm and ngModel Directives

- ngModel: Binds input fields to component properties using two-way binding.
- ngForm: Tracks form state and validity; applied automatically to <form> tags.

Example

```
<form #form="ngForm">
  <input name="firstName" [(ngModel)]="firstName" required />
</form>
```

ReactiveFormsModule - Reactive Forms

How to Import

```
import { ReactiveFormsModule } from '@angular/forms';

@NgModule({
  imports: [ReactiveFormsModule],
})
export class AppModule {}
```

FormGroup, FormControl, FormBuilder

- FormControl: Tracks the value and validation of an individual form control.
- FormGroup: Tracks the same for a group of controls.
- FormBuilder: Helper class to build forms using less code.

Example using FormGroup and FormControl:

```
// app.component.ts
import { FormGroup, FormControl } from '@angular/forms';

export class AppComponent {
  userForm = new FormGroup({
    name: new FormControl(''),
    email: new FormControl(''),
```

```
  });

  onSubmit() {
    console.log(this.userForm.value);
  }
}
<form [formGroup]="userForm" (ngSubmit)="onSubmit()">
  <input formControlName="name" />
  <input formControlName="email" />
  <button type="submit">Submit</button>
</form>
```

Example using FormBuilder:

```
import { FormBuilder } from '@angular/forms';

export class AppComponent {
  constructor(private fb: FormBuilder) {}

  userForm = this.fb.group({
    name: [''],
    email: [''],
  });
}
```

Dynamic Forms

Used when form structure is not known at compile time. Controls are added dynamically in TypeScript.

Example

```
this.userForm = new FormGroup({});
this.fields.forEach(field => {
  this.userForm.addControl(field.name, new FormControl(''));
});
```

Form Control States

Each form control has the following states:

State	Description
valid	The control has passed all validators
invalid	One or more validators have failed
dirty	The user has changed the value
pristine	The user has not yet changed the value

Example (checking states)

```
<input formControlName="email" #email="ngModel" />

<p *ngIf="email.dirty">You've changed the email!</p>
<p *ngIf="email.pristine">Email is still untouched.</p>
<p *ngIf="email.invalid">Invalid email.</p>
```

Route Guards - Secure Routes

Angular provides **route guards** to control navigation to and from components. They help in securing routes by deciding whether the route can be **activated**, **deactivated**, or **loaded**, etc.

CanActivate - Controlling Access to Routes

Used to prevent unauthorized access to certain routes (e.g., admin areas).

Use Case:

Only authenticated users can access /dashboard.

Steps:

1. Create a guard using Angular CLI:

   ```
   ng generate guard auth/auth
   ```

2. Implement the CanActivate interface:

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

   @Injectable({ providedIn: 'root' })
   export class AuthGuard implements CanActivate {
     constructor(private authService: AuthService, private router: Router) {}

     canActivate(): boolean {
       if (this.authService.isLoggedIn()) {
         return true;
       } else {
         this.router.navigate(['/login']);
         return false;
       }
     }
   ```

```
}
```

3. Apply to routes:

```
const routes: Routes = [
  { path: 'dashboard', component: DashboardComponent, canActivate: [AuthGuard] }
];
```

CanDeactivate - Preventing Unsaved Navigation

Used to warn users if they try to navigate away with unsaved changes.

Use Case:

Prevent users from leaving a form with unsaved changes.

Steps:

1. Create an interface:

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

2. Implement in a component (e.g., form):

```
export class ProfileEditComponent implements CanComponentDeactivate {
  hasUnsavedChanges = true;

  canDeactivate(): boolean {
    return !this.hasUnsavedChanges || confirm("You have unsaved changes. Do you
really want to leave?");
  }
}
```

3. Create a guard:

```
@Injectable({ providedIn: 'root' })
export class CanDeactivateGuard implements
CanDeactivate<CanComponentDeactivate> {
  canDeactivate(component: CanComponentDeactivate): boolean |
Observable<boolean> {
    return component.canDeactivate();
  }
}
```

4. Apply to routes:

```
const routes: Routes = [
```

```
  { path: 'edit-profile', component: ProfileEditComponent, canDeactivate:
[CanDeactivateGuard] }
];
```

HttpClient Module

Angular's HttpClient service allows communication with backend services over HTTP.

Setup

Import in your module:

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

@NgModule({
  imports: [HttpClientModule]
})
export class AppModule {}
```

Inject HttpClient in your service:

```
ts
CopyEdit
constructor(private http: HttpClient) {}
```

Communicating with the Server using HttpClient

You typically use HttpClient in services to:

- Fetch data (GET)
- Send data (POST, PUT)
- Delete resources (DELETE)
- Intercept requests/responses

Making HTTP GET Requests

```
getUsers(): Observable<User[]> {
  return this.http.get<User[]>('https://api.example.com/users');
}
```

Example Usage:

```
ts
CopyEdit
this.userService.getUsers().subscribe(users => this.users = users);
```

Making HTTP POST and PUT Requests

POST – Create New Data:

```
addUser(user: User): Observable<User> {
  return this.http.post<User>('https://api.example.com/users', user);
}
```

PUT – Update Existing Data:

```
ts
CopyEdit
updateUser(user: User): Observable<User> {
  return this.http.put<User>(`https://api.example.com/users/${user.id}`, user);
}
```

Issuing an HTTP DELETE Request

```
deleteUser(id: number): Observable<void> {
  return this.http.delete<void>(`https://api.example.com/users/${id}`);
}
```

Intercepting Requests and Responses

Use **HTTP interceptors** to modify requests globally (e.g., add tokens, log, handle errors).

Create an Interceptor:

```
ng generate interceptor auth
```

Example: Add Auth Token

```
@Injectable()
export class AuthInterceptor implements HttpInterceptor {
  intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
    const token = localStorage.getItem('authToken');
    if (token) {
      const cloned = req.clone({
        headers: req.headers.set('Authorization', `Bearer ${token}`)
      });
      return next.handle(cloned);
    }
    return next.handle(req);
  }
}
```

Register the Interceptor:

```
providers: [
  { provide: HTTP_INTERCEPTORS, useClass: AuthInterceptor, multi: true }]
```

Angular Deployment

Build Process

The Angular CLI provides a powerful build system that compiles your application for production deployment.

```
bash
```

```
# Basic production build
ng build --configuration production

# Equivalent shorthand
ng build --prod
```

What happens during build:

- TypeScript compilation to JavaScript
- Template and style processing
- Ahead-of-Time (AOT) compilation (for production)
- Code minification and optimization
- Tree-shaking to remove unused code
- Asset processing and copying

Example build output:

```
dist/my-app/
├── index.html
├── favicon.ico
├── assets/
│   ├── images/
│   └── data.json
└── <hash>.bundle.js
└── <hash>.css
```

Deploying Angular Applications

1. Static File Deployment (Most Common)

```
# Build for production
ng build --prod

# The dist/ folder contains all static files ready for deployment
```

Deployment Targets:

- **Netlify/Vercel**: Drag and drop dist folder or connect Git repository
- **Firebase Hosting**:

```
npm install -g firebase-tools
firebase login
firebase init hosting
ng build --prod
firebase deploy
```

- **AWS S3 + CloudFront**:

```
# Build and sync with S3
ng build --prod
aws s3 sync dist/my-app s3://my-bucket --delete
```

Angular Testing

Testing Fundamentals

describe(), it(), beforeEach(), afterEach()

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

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

  // Runs before each test in this describe block
  beforeEach(() => {
    TestBed.configureTestingModule({});
```

```
    service = TestBed.inject(CalculatorService);
  });

  // Runs after each test
  afterEach(() => {
    // Clean up if needed
  });

  // Individual test case
  it('should add two numbers correctly', () => {
    const result = service.add(2, 3);
    expect(result).toBe(5);
  });

  it('should subtract two numbers correctly', () => {
    const result = service.subtract(5, 3);
    expect(result).toBe(2);
  });
});
```

Testing Services

```
import { HttpClientTestingModule, HttpTestingController } from '@angular/common/http/testing';

describe('UserService', () => {
  let service: UserService;
  let httpMock: HttpTestingController;

  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [HttpClientTestingModule],
      providers: [UserService]
    });

    service = TestBed.inject(UserService);
    httpMock = TestBed.inject(HttpTestingController);
  });

  afterEach(() => {
    httpMock.verify(); // Verify no outstanding requests
  });

  it('should fetch users', () => {
    const mockUsers = [{ id: 1, name: 'John' }, { id: 2, name: 'Jane' }];

    service.getUsers().subscribe(users => {
      expect(users.length).toBe(2);
      expect(users).toEqual(mockUsers);
    });

    const req = httpMock.expectOne('https://api.example.com/users');
    expect(req.request.method).toBe('GET');
```

```
    req.flush(mockUsers);
  });
});
```

Testing Components

```
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { By } from '@angular/platform-browser';

describe('UserListComponent', () => {
  let component: UserListComponent;
  let fixture: ComponentFixture<UserListComponent>;

  beforeEach(async () => {
    await TestBed.configureTestingModule({
      imports: [UserListComponent], // For standalone components
      // OR for module-based:
      // declarations: [UserListComponent],
      // providers: [UserService]
    }).compileComponents();

    fixture = TestBed.createComponent(UserListComponent);
    component = fixture.componentInstance;
    fixture.detectChanges();
  });

  it('should create', () => {
    expect(component).toBeTruthy();
  });

  it('should display users', () => {
    component.users = [{ id: 1, name: 'John' }, { id: 2, name: 'Jane' }];
    fixture.detectChanges();

    const userElements = fixture.debugElement.queryAll(By.css('.user-item'));
    expect(userElements.length).toBe(2);

    const firstUserName = userElements[0].nativeElement.textContent;
    expect(firstUserName).toContain('John');
  });

  it('should emit userSelected event', () => {
    spyOn(component.userSelected, 'emit');
    const user = { id: 1, name: 'John' };

    component.selectUser(user);
    expect(component.userSelected.emit).toHaveBeenCalledWith(user);
  });
});
```

Code Coverage

Generate Code Coverage Report:

```
# Run tests with code coverage
```

```
ng test --no-watch --code-coverage

# For CI environments (headless browser, no progress)
ng test --no-watch --no-progress --browsers=ChromeHeadless --code-coverage
```

Coverage Configuration (angular.json):

```
{
  "projects": {
    "my-app": {
      "architect": {
        "test": {
          "options": {
            "codeCoverage": true,
            "codeCoverageExclude": [
              "src/test.ts",
              "src/**/*.spec.ts",
              "src/environments/*.ts"
            ]
          }
        }
      }
    }
  }
}
```

Coverage Report:

- Generated in coverage/ directory
- Open coverage/index.html in browser to view detailed report
- Shows statement, branch, function, and line coverage

Standalone Components

Basic Standalone Component

```
import { Component } from '@angular/core';
import { CommonModule } from '@angular/common';
import { RouterModule } from '@angular/router';

@Component({
  selector: 'app-user-list',
  standalone: true,
  imports: [CommonModule, RouterModule], // Explicitly import needed modules
  template: `
    <h2>Users</h2>
    <ul>
      <li *ngFor="let user of users">
        <a [routerLink]="['/user', user.id]">{{ user.name }}</a>
      </li>
    </ul>
  `
})
export class UserListComponent {
  users = [{ id: 1, name: 'John' }, { id: 2, name: 'Jane' }];
```

```
}
provideRouter() and provideHttpClient()
```

Bootstrapping with Standalone APIs:

```
// main.ts
import { bootstrapApplication } from '@angular/platform-browser';
import { provideRouter } from '@angular/router';
import { provideHttpClient } from '@angular/common/http';
import { AppComponent } from './app/app.component';
import { routes } from './app/routes';

bootstrapApplication(AppComponent, {
  providers: [
    provideRouter(routes),
    provideHttpClient(),
    // Other providers...
  ]
}).catch(err => console.error(err));
```

Using HttpClient in Standalone Components:

```
import { Component, inject } from '@angular/core';
import { HttpClient } from '@angular/common/http';

@Component({
  standalone: true,
  template: `...`
})
export class DataComponent {
  private http = inject(HttpClient);

  loadData() {
    this.http.get('/api/data').subscribe(data => {
      // handle data
    });
  }
}
Lazy Loading
Route-Based Lazy Loading
// app.routes.ts
import { Routes } from '@angular/router';

export const routes: Routes = [
  { path: '', component: HomeComponent },
  {
    path: 'users',
    loadComponent: () =>
      import('./users/users.component').then(c => c.UsersComponent)
  },
  {
    path: 'admin',
```

```
    loadChildren: () =>
      import('./admin/admin.routes').then(m => m.ADMIN_ROUTES)
  },
  {
    path: 'products',
    loadChildren: () =>
      import('./products/products.module').then(m => m.ProductsModule)
  }
];
Lazy Loading with Preloading
import { PreloadAllModules } from '@angular/router';

export const routes: Routes = [/*...*/];

// In main.ts or app.config.ts
provideRouter(routes, {
  preloadingStrategy: PreloadAllModules
})
```

Signals

Basic Signal Usage

```
import { Component, signal, computed, effect } from '@angular/core';

@Component({
  standalone: true,
  template: `
    <h2>Counter: {{ count() }}</h2>
    <p>Double: {{ doubleCount() }}</p>
    <button (click)="increment()">Increment</button>
  `
})
export class CounterComponent {
  count = signal(0);
  doubleCount = computed(() => this.count() * 2);

  constructor() {
    // Effect runs when dependencies change
    effect(() => {
      console.log('Count changed:', this.count());
    });
  }

  increment() {
    this.count.update(value => value + 1);
  }

  reset() {
    this.count.set(0);
  }
```

```
}
Signal-based Services
import { Injectable, signal } from '@angular/core';

export interface User {
  id: number;
  name: string;
  email: string;
}

@Injectable({ providedIn: 'root' })
export class UserService {
  private usersSignal = signal<User[]>([]);
  users = this.usersSignal.asReadonly(); // Expose read-only signal

  loadUsers() {
    // Simulate API call
    setTimeout(() => {
      const newUsers: User[] = [
        { id: 1, name: 'John', email: 'john@example.com' },
        { id: 2, name: 'Jane', email: 'jane@example.com' }
      ];
      this.usersSignal.set(newUsers);
    }, 1000);
  }

  addUser(user: User) {
    this.usersSignal.update(users => [...users, user]);
  }
}
```

Deferrable Views

Basic Deferrable View

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

@Component({
  standalone: true,
  template: `
    <h1>Main Content</h1>

    @defer (on viewport) {
      <app-heavy-component />
    } @placeholder {
      <div>Loading heavy component...</div>
    } @loading (minimum 1s) {
      <div>Loading...</div>
    } @error {
      <div>Failed to load component</div>
    }
```

```
  `
})
export class MainComponent {}
```

Defer with Triggers

```
template: `
  <!-- Defer when element is in viewport -->
  @defer (on viewport) {
    <app-content />
  }

  <!-- Defer on interaction -->
  <button #trigger>Load Content</button>
  @defer (on interaction(trigger)) {
    <app-interactive-content />
  }

  <!-- Defer on hover -->
  <div #hoverTrigger>Hover to load</div>
  @defer (on hover(hoverTrigger)) {
    <app-hover-content />
  }

  <!-- Defer when idle (after main thread is free) -->
  @defer (on idle) {
    <app-idle-content />
  }

  <!-- Defer on timer -->
  @defer (on timer(5s)) {
    <app-timed-content />
  }

  <!-- Defer when condition becomes true -->
  @defer (when shouldLoad()) {
    <app-conditional-content />
  }

  <!-- Multiple triggers -->
  @defer (on viewport; interaction; timer(10s)) {
    <app-multi-trigger-content />
  }
`
```

Prefetching with Defer

```
template: `
  @defer (on viewport; prefetch on idle) {
    <app-prefetched-content />
  }

  <!-- Or with explicit prefetch trigger -->
```

```
  <button #prefetchTrigger>Prefetch Content</button>
  @defer (on interaction; prefetch on interaction(prefetchTrigger)) {
    <app-explicit-prefetch />
  }
`
```

Testing Deferrable Views

```
import { ComponentFixture, fakeAsync, tick } from '@angular/core/testing';

it('should load deferred content on interaction', fakeAsync(() => {
  const button = fixture.debugElement.query(By.css('button'));
  button.triggerEventHandler('click', null);
  tick(); // Advance time
  fixture.detectChanges();

  const deferredContent = fixture.debugElement.query(By.css('app-heavy-component'));
  expect(deferredContent).toBeTruthy();
}));
```