Angular 19 Development Training

25 Sep – 26 Sep

Instructor - https://in.linkedin.com/in/suryakantsurve

Course Outline

- Node.js
- ES6/ES7 Fundamentals
- Typescript
- Reactive Programming Basics (RxJS)
- Angular
- Angular CLI
- Building Blocks of Angular
- Components
- Templates and Views
- Events
- Directives
- Pipes
- Services
- FormsModule
- HttpClient Module
- RouterModule
- Testing
- Angular Deployment
- Signals
- Standalone Components

Course Objectives

- Create single-page applications with Angular.
- Understand the architecture behind an Angular application and how to use it.
- Develop modern, responsive and scalable web applications with Angular.
- Gain deep understanding of the Angular

Prerequisites

- Knowledge of HTML and JavaScript

Lab Setup

- Node.js - v^18.19.1 or newer
- Text editor - We recommend Visual Studio Code
- Terminal - Required for running Angular CLI commands
- Development Tool - To improve your development workflow, we recommend the Angular Language Service

ES6

Let

- ES6 provides a new way of declaring a variable by using the let keyword.

- The let keyword is similar to the var keyword, except that these variables are blocked-scope.

```
let x = 10;
if (x == 10) {
   let x = 20;
   console.log(x); // 20:  reference x inside the block
}
console.log(x); // 10: reference at the begining of the script
```

Const

- ES6 provides a new way of declaring a constant by using the const keyword. The const keyword creates a read-only reference to a value.

```
const RATE = 0.1;
RATE = 0.2; // TypeError
```

for loop

- ES6 provides a new construct called for...of that allows you to create a loop iterating over iterable objects such as arrays, maps, and sets.

```
let scores = [75, 80, 95];
for (let score of scores) {
        console.log(score);
}
```

Default Parameters

- In JavaScript, default function parameters allow you to initialize named parameters with default values if no values or undefined are passed into the function.

```
function say(message='Hi') {
    console.log(message);
}

say(); // 'Hi'
say('Hello') // 'Hello'
```

Rest Parameters

- ES6 provides a new kind of parameter so-called rest parameter that has a prefix of three dots (...). A rest parameter allows you to represent an indefinite number of arguments as an array.

```
function sum(...args) {
    let total = 0;
    for (const a of args) {
        total += a;
    }
    return total;
}
sum(1, 2, 3);
```

Spread Operator

- ES6 provides a new operator called spread operator that consists of three dots (...). The spread operator allows you to spread out elements of an iterable object such as an array, map, or set.

```
const odd = [1,3,5];
const combined = [2,4,6, ...odd];
console.log(combined);
```

Template Literals

- In ES6, you create a template literal by wrapping your text in backticks (`)

```
let firstName = 'John',
lastName = 'Doe';

let greeting = `Hi ${firstName}, ${lastName}`;
console.log(greeting); //
```

Destructuring Assignment

- ES6 introduces a new feature called destructuring assignment, which lets you destructure properties of an object or elements of an array into individual variables.

```
function getScores() {
  return [70, 80, 90];
}

let [x, y, z] = getScores();

console.log({ x, y, z });
```

Object Destructuring

```
let person = {
    firstName: 'John',
    lastName: 'Doe',
    currentAge: 28
};

let { firstName, lastName} = person;
```

Arrow Functions

- Arrow functions in JavaScript provide a concise syntax for writing function expressions. They are defined using the => (arrow) notation.

- Key Differences from Traditional Functions
  - this binding: Arrow functions do not have their own this context. They inherit the this value from the surrounding lexical scope.
  - arguments object: Arrow functions do not have their own arguments object.
  - new keyword: Arrow functions cannot be used as constructors (i.e., they cannot be called with the new keyword).

ES6 Modules

- ES6 introduced the concept of modules. A module is a JavaScript file that executes in strict mode. It means that any variables or functions declared in the module won't be added automatically to the global scope.

```
function display(message) {
  const el = document.createElement('div');
  el.innerText = message;
  document.body.appendChild(el);
}
export { display };
```

```
import { display } from './lib.js';
display('Hi');
```

ES6 Class

- ES6 introduced a new syntax for declaring a class as shown in this example:

```
class Person {
    constructor(name) {
        this.name = name;
    }
    getName() {
        return this.name;
    }
}
let john = new Person("John Doe");
let name = john.getName();
console.log(name); // "John Doe"
```

Getters and Setters

```
class Person {
    constructor(name) {
        this._name = name;
    }
    get name() {
        return this._name;
    }
    set name(newName) {
        newName = newName.trim();
        if (newName === '') {
            throw 'The name cannot be empty';
        }
        this._name = newName;
    }
}
```

Promises

- A promise is an object that encapsulates the result of an asynchronous operation.

```
function getUsers() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve([
        { username: 'john', email: 'john@test.com' },
        { username: 'jane', email: 'jane@test.com' },
      ]);
    }, 1000);
  });
}
-----------------------------------------
const promise = getUsers();
promise.then((users) => {
  console.log(users);
});
```

ES6 collections

- Map object holds key-value pairs. Keys are unique in a Map's collection. In other words, a key in a Map object only appears once.

```
const map1 = new Map();
map1.set("a", 1);
map1.set("b", 2);
map1.set("c", 3);
console.log(map1.get("a"));// Expected output: 1
```

Typescript

TypeScript

- TypeScript is a strongly-typed, object-oriented, compiled language developed by Microsoft.
- It is a superset of JavaScript, and essentially, it adds static typing with optional type annotations to JavaScript.
- TypeScript is designed for the development of large applications and transcompiles to JavaScript.
- Because TypeScript is a superset of JavaScript, existing JavaScript programs are also valid TypeScript programs.

Key Features

- Static Typing: Types are explicitly defined, allowing for compile-time checking and reducing runtime errors.
- Object-Oriented Programming: Supports classes, interfaces, inheritance, and other OOP concepts.
- ES6 Features: Includes support for the latest ECMAScript features, such as arrow functions, classes, and modules.
- Tooling: Provides excellent tooling support, including code completion, refactoring, and debugging.
- Compatibility: Compiles to clean, readable JavaScript code that can run on any platform.

Usage

- TypeScript is commonly used for both front-end and back-end development.
- On the front-end, it is often used with frameworks like React, Angular, and Vue.
- On the back-end, it can be used with Node.js to build server-side applications.
- To install TypeScript, you need to have Node.js and npm (Node Package Manager) installed.
- You can then install TypeScript globally using the following command:
  - npm install -g typescript
  - tsc your_file_name.ts

Everyday Types

- The primitives: string, number, and Boolean

```
let myName: string = "Alice";
```

- Arrays - Array<number> , Array<string>

- Return Type Annotations

```
function getFavoriteNumber(): number {
return 26;
}
```

- Object Types

```
function printCoord(pt: { x: number; y: number }) {
console.log("The coordinate's x value is " + pt.x);
console.log("The coordinate's y value is " + pt.y);
}
```

Type Aliases

- A type alias is exactly that - a name for any type. The syntax for a type alias is:

```
type Point = {
  x: number;
  y: number;
};

// Exactly the same as the earlier example
function printCoord(pt: Point) {
  console.log('The coordinate's x value is ' + pt.x);
  console.log('The coordinate's y value is ' + pt.y);
}
 printCoord({ x: 100, y: 100 });
```

Interfaces

- An interface declaration is another way to name an object type:

```
interface Point {
  x: number;
  y: number;
}

function printCoord(pt: Point) {
  console.log("The coordinate's x value is " + pt.x);
  console.log("The coordinate's y value is " + pt.y);
}

printCoord({ x: 100, y: 100 });
```

null and undefined

- JavaScript has two primitive values used to signal absent or uninitialized value: null and undefined.

- TypeScript has two corresponding types by the same names. How these types behave depends on whether you have the strictNullChecks option on.

- With strictNullChecks on, when a value is null or undefined, you will need to test for those values before using methods or properties on that value.

```
function doSomething(x: string | null) {
    if (x === null) {
    // do nothing
    } else {
    console.log("Hello, " + x.toUpperCase());
    }
    }
```

Number enums

- Enums allow a developer to define a set of named constants. Using enums can make it easier to document intent, or create a set of distinct cases. TypeScript provides both numeric and string-based enums.

```
enum Direction {
Up = 1,
Down,
Left,
Right,
}
```

- Above, we have a numeric enum where Up is initialized with 1. All of the following members are auto-incremented from that point on. In other words, Direction.Up has the value 1, Down has 2, Left has 3, and Right has 4.

```
enum UserResponse {
  No = 0,
  Yes = 1,
}

function respond(recipient: string, message: UserResponse): void {
  // ...
}

respond("Princess Caroline", UserResponse.Yes);
```

String enums


```
enum Direction {
  Up = "UP",
  Down = "DOWN",
  Left = "LEFT",
  Right = "RIGHT",
}
```

for..of statements

- for..of loops over an iterable object, invoking the Symbol.iterator property on the object. Here is a simple for..of loop on an array:

```
let someArray = [1, "string", false];
for (let entry of someArray) {
  console.log(entry); // 1, "string", false
}
```

Classes

```
class Person {
    ssn: string;
    firstName: string;
    lastName: string;
    constructor(ssn: string, firstName: string, lastName: string) {
        this.ssn = ssn;
        this.firstName = firstName;
        this.lastName = lastName;
    }
    getFullName(): string {
        return `${this.firstName} ${this.lastName}`;
    }
}
```

Access Modifiers

- Access modifiers change the visibility of the properties and methods of a class. TypeScript provides three access modifiers:
  - private
  - protected
  - public

readonly

- TypeScript provides the readonly modifier that allows you to mark the properties of a class immutable. The assignment to a readonly property can only occur in one of two places:
  - In the property declaration.
  - In the constructor of the same class.

```
class Person {
   readonly birthDate: Date;

   constructor(birthDate: Date) {
      this.birthDate = birthDate;
   }
}
```

TypeScript Getters and Setters

```
class Person {
  private _name: string;
  constructor(name: string) {
    this._name = name;
  }
  get name(): string {
    return this._name;
  }
  set name(newName: string) {
    if (newName.length > 0) {
      this._name = newName;
    }
  }
}
const person = new Person("Alice");
console.log(person.name);
person.name = "Bob";
console.log(person.name);
```

RxJS

Reactive Programming

- Reactive programming is an essential part of modern web applications.
- RxJS allows you to create reactive programs with JavaScript to better serve your users.
- RxJS is a library used to create asynchronous programs using observable sequences.
- Reactive programs are structured around events rather than sequential top-down execution of iterative code. This allows them to respond to a trigger event regardless of when what stage the program is on.

Asynchronous vs Synchronous

- One of the main concepts in reactive programming is synchronous vs asynchronous data. In short, synchronous data is delivered one at a time, as soon as possible.

- Asynchronous data waits for a set event and is then delivered all at once through a "callback". Asynchronous data is more popular in reactive programming because it fits well with the paradigm's event-based approach.

RXJS

- RxJS is one of the hottest libraries in web development today.
- Offering a powerful, functional approach for dealing with events and with integration points into a growing number of frameworks, libraries, and utilities.
- RxJS is more specifically a functional reactive programming tool featuring the observer pattern and the iterator pattern.
- Includes an adapted form of JavaScript's array functions (reduce, map, etc.) to handle asynchronous events as collections.

Observable

- An observable represents a stream, or source of data that can arrive over time.
- An observable's data is a stream of values that can then be transmitted synchronously or asynchronously.
- Consumers can then subscribe to observables to listen to all the data they transmit.
- Consumers can be subscribed to multiple observables at the same time.
- This data can then be transformed as it moves through the data pipeline toward the user.

Subscribers (observers)

VINSYS

- The Observable is only useful if someone consumes the value emitted by the observable. We call them observers or subscribers.

- The observers communicate with the Observable using callbacks

- The observer must subscribe to the observable to receive the value from the observable. While subscribing, it optionally passes the three callbacks. next(), error() & complete()

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

// Create an Observable out of a promise
const data = from(fetch('/api/endpoint'));
// Subscribe to begin listening for async result
data.subscribe({
  next(response) { console.log(response); },
  error(err) { console.error('Error: ' + err); },
  complete() { console.log('Completed'); }
});
```

Operators

```
import { of } from 'rxjs';
import { map } from 'rxjs/operators';
/*
 *  'of' allows you to deliver values in a sequence
 *  In this case, it will emit 1,2,3,4,5 in order.
 */
const dataSource = of(1, 2, 3, 4, 5);

// subscribe to our source observable
const subscription = dataSource
  .pipe(
    // add 1 to each emitted value
    map(value => value + 1)
  )
  // log: 2, 3, 4, 5, 6
  .subscribe(value => console.log(value));
```

```
import { of } from 'rxjs';
import { filter } from 'rxjs/operators';

const dataSource = of(1, 2, 3, 4, 5);

// subscribe to our source observable
const subscription = dataSource
  .pipe(
    // only accept values 2 or greater
    filter(value => value >= 2)
  )
  // log: 2, 3, 4, 5
  .subscribe(value => console.log(value));
```

Introduction to Node.js

- Node.js is an open-source and cross-platform JavaScript runtime environment.

- Node.js runs the V8 JavaScript engine, the core of Google Chrome, outside of the browser.

NPM

- npm is the standard package manager for Node.js
- npm installs, updates and manages downloads of dependencies of your project.
- Dependencies are pre-built pieces of code, such as libraries and packages, that your Node.js application needs to work.
- If a project has a package.json file, by running npm install it will install everything the project needs, in the node_modules folder, creating it if it's not existing already.
- You can also install a specific package by running npm install <package>

npm init

npm install <package>

npm update

npm install <package-name>@<version>

npm run <task-name>

npm uninstall <package>

Angular

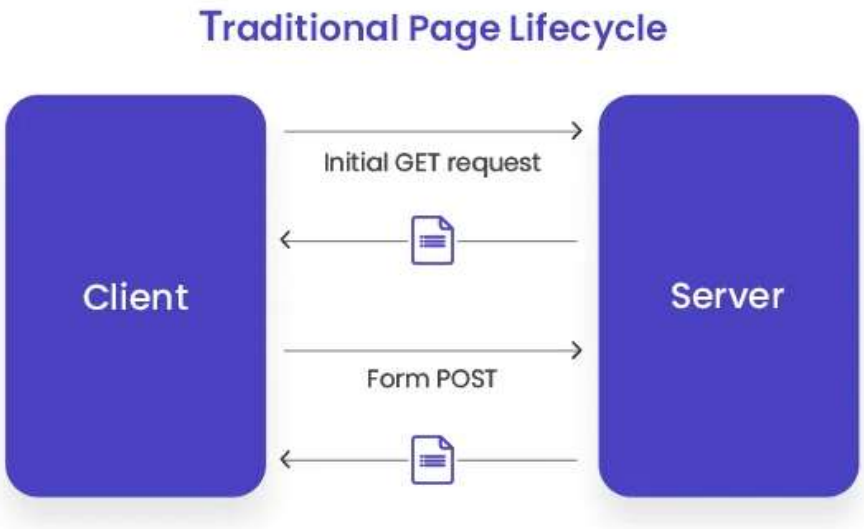
What is Angular?

- Angular is a web framework that empowers developers to build fast, reliable applications.

- Maintained by a dedicated team at Google, Angular provides a broad suite of tools, APIs, and libraries to simplify and streamline your development workflow.

- Angular gives you a solid platform on which to build fast, reliable applications that scale with both the size of your team and the size of your codebase.

SPA vs MPA


Traditional Page Lifecycle
Client
Server
Initial GET request
Form POST


SPA Lifecycle
Client
Server
Initial GET request
AJAX
JSON

PWA

- A Progressive Web App (PWA) is a web application that can be installed on a device and used like a native app.

- PWAs leverage web technologies (HTML, CSS, JavaScript) but offer a user experience similar to native apps, such as offline access, push notifications, and a dedicated app icon.

- They are installable through the web browser and can be accessed from the home screen or start menu

Key Features of PWAs

- Installable
- Offline Capabilities
- Push Notifications
- Single Codebase
- Performance
- User Experience:

Install Angular CLI

- npm install -g @angular/cli
- ng new my-first-angular-app
- cd my-first-angular-app
- npm start

Components

- Components are the main building blocks of Angular applications.
- Each component represents a part of a larger web page.
- Organizing an application into components helps provide structure to your project, clearly separating code into specific parts that are easy to maintain and grow over time.

Defining a component

- Every component has a few main parts:
  - A @Component decorator that contains some configuration used by Angular.
  - An HTML template that controls what renders into the DOM.
  - A CSS selector that defines how the component is used in HTML.
  - A TypeScript class with behaviors, such as handling user input or making requests to a server.

user-profile.ts

```
// user-profile.ts
@Component({
  selector: 'user-profile',
  templateUrl: 'user-profile.html',
  styleUrl: 'user-profile.css',
})
export class UserProfile {
  // Component behavior is defined in here
}
```

user-profile.html

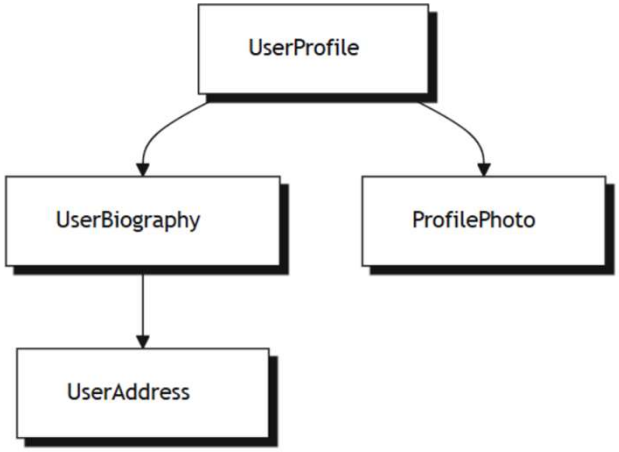

```
<h1>Use profile</h1>
<p>This is the user profile page</p>
```

user-profile.css


```
h1 {
  font-size: 3em;
}
```

Using components

- You build an application by composing multiple components together. For example, if you are building a user profile page, you might break the page up into several components like this:



Component selectors

- Every component defines a CSS selector that determines how the component is used:

```
@Component({
  selector: 'profile-photo',
  ...
})
export class ProfilePhoto { }

@Component({
  template: `
    <profile-photo />
    <button>Upload a new profile photo</button>`,
  ...,
})
export class UserProfile { }
```

Styling components

- Components can optionally include CSS styles that apply to that component's DOM:

```
@Component({
  selector: 'profile-photo',
  template: `<img src="profile-photo.jpg" alt="Your profile photo">`,
  styles: ` img { border-radius: 50%; } `,
})
export class ProfilePhoto { }
```

- You can also choose to write your styles in separate files:

```
@Component({
  selector: 'profile-photo',
  templateUrl: 'profile-photo.html',
  styleUrl: 'profile-photo.css',
})
export class ProfilePhoto { }
```

Accepting data with input properties

- When you use a component, you commonly want to pass some data to it. A component specifies the data that it accepts by declaring inputs:

```
import {Component, input} from '@angular/core';
@Component({/*...*/})
export class CustomSlider {
  // Declare an input named 'value' with a default value of zero.
  value = input(0);
}
```

This lets you bind to the property in a template:

```
<custom-slider [value]="50" />
```

Type for the input

```
@Component({/*...*/})
export class CustomSlider {
  // Produces an InputSignal<number | undefined> because `value` may not be set.
  value = input<number>();
}
```

Template syntax

- Every Angular component has a template that defines the DOM that the component renders onto the page.

- By using templates, Angular is able to automatically keep your page up-to-date as data changes.

- Templates are based on HTML syntax, with additional features such as built-in template functions, data binding, event listening, variables, and more.

Render dynamic text with text interpolation

- You can bind dynamic text in templates with double curly braces, which tells Angular that it is responsible for the expression inside and ensuring it is updated correctly. This is called text interpolation.

```
@Component({
  template: `
    <p> Your color preference is {{ theme }} </p>
  `,
  ...
})
export class AppComponent {
  theme = 'dark';
}
```

Binding dynamic properties and attributes

- Angular supports binding dynamic values into object properties and HTML attributes with square brackets.

- Every HTML element has a corresponding DOM representation. For example, each <button> HTML element corresponds to an instance of HTMLButtonElement in the DOM. In Angular, you use property bindings to set values directly to the DOM representation of the element.

```
<button [disabled]="isFormValid">Save</button>

<img [src]="profilePhotoUrl" alt="The current user's profile photo">
```

CSS class and style property bindings

- Angular supports additional features for binding CSS classes and CSS style properties to elements.

- You can create a CSS class binding to conditionally add or remove a CSS class on an element based on whether the bound value is truthy or falsy.

```
<!-- When `isExpanded` is truthy, add the `expanded` CSS class. -->
<ul [class.expanded]="isExpanded">
```

CSS classes

```
@Component({
  template: `
    <ul [class]="listClasses"> ... </ul>
    <section [class]="sectionClasses"> ... </section>
    <button [class]="buttonClasses"> ... </button>
  `,
  ...
})
export class UserProfile {
  listClasses = 'full-width outlined';
  sectionClasses = ['expandable', 'elevated'];
  buttonClasses = {
    highlighted: true,
    embiggened: false,
  };
}
```

CSS style properties

- You can also bind to CSS style properties directly on an element.

```
<!-- Set the CSS `display` property based on the `isExpanded` property. -->
<section [style.display]="isExpanded ? 'block' : 'none'">

<!-- Set the CSS `height` property to a pixel value based on the
`sectionHeightInPixels` property. -->
<section [style.height.px]="sectionHeightInPixels">
```

Lifecycle

- A component's lifecycle is the sequence of steps that happen between the component's creation and its destruction.
- Each step represents a different part of Angular's process for rendering components and checking them for updates over time.
- In your components, you can implement lifecycle hooks to run code during these steps.
- Lifecycle hooks that relate to a specific component instance are implemented as methods on your component class.

- Constructor - Runs when Angular instantiates the component.
- ngOnInit - Runs once after Angular has initialized all the component's inputs.
- ngOnChanges - Runs every time the component's inputs have changed.
- ngDoCheck - Runs every time this component is checked for changes.
- ngAfterContentInit - Runs once after the component's content has been initialized.
- ngAfterContentChecked - Runs every time this component content has been checked for changes.
- ngAfterViewInit - Runs once after the component's view has been initialized.
- ngAfterViewChecked - Runs every time the component's view has been checked for changes.
- afterRender - Runs every time all components have been rendered to the DOM.
- ngOnDestroy - Runs once before the component is destroyed.

```
@Component({
  template: `
    <ul [style]="listStyles"> ... </ul>
    <section [style]="sectionStyles"> ... </section>
  `,
  ...
})
export class UserProfile {
  listStyles = 'display: flex; padding: 8px';
  sectionStyles = {
    border: '1px solid black',
    'font-weight': 'bold',
  };
}
```

ng-content

- <ng-content> is a special element that accepts markup or a template fragment and controls how components render content.

```
// ./base-button/base-button.component.ts
import { Component } from '@angular/core';
@Component({
  selector: 'button[baseButton]',
  template: `
      <ng-content />
  `,
})
export class BaseButton {}
```

```
// ./app.component.ts
import { Component } from '@angular/core';
import { BaseButton } from './base-button/base-button.component.ts';
@Component({
  selector: 'app-root',
  imports: [BaseButton],
  template: `
    <button baseButton>
      Next <span class="icon arrow-right" />
    </button>
  `,
})
export class AppComponent {}
```

Creating a template fragment

```
<p>This is a normal element</p>
<ng-template>
  <p>This is a template fragment</p>
</ng-template>
```

Grouping elements with ng-container

- <ng-container> is a special element in Angular that groups multiple elements together or marks a location in a template without rendering a real element in the DOM.

```
<!-- Component template -->
<section>
  <ng-container>
    <h3>User bio</h3>
    <p>Here's some info about the user</p>
  </ng-container>
</section>
```

Local template variables with @let

- Use @let to declare a variable whose value is based on the result of a template expression. Angular automatically keeps the variable's value up-to-date with the given expression.

```
@let name = user.name;
@let greeting = 'Hello, ' + name;
@let data = data$ | async;
@let pi = 3.1459;
@let coordinates = {x: 50, y: 100};
@let longExpression = 'Lorem ipsum dolor sit amet, consectetur adipiscing elit ' +
                      'sed do eiusmod tempor incididunt ut labore et dolore magna ' +
                      'Ut enim ad minim veniam...';
```

Referencing the value of @let

```
@let user = user$ | async;
@if (user) {
  <h1>Hello, {{user.name}}</h1>
  <user-avatar [photo]="user.photo"/>
  <ul>
    @for (snack of user.favoriteSnacks; track snack.id) {
      <li>{{snack.name}}</li>
    }
  </ul>
  <button (click)="update(user)">Update profile</button>
}
```

Template reference variables

- Template reference variables give you a way to declare a variable that references a value from an element in your template.

<!-- Create a template reference variable named "taskInput", referring to the HTMLInputElement. -->

<input #taskInput placeholder="Enter task name">

Conditionally display content with @if, @else-if and @else

```
@if (a > b) {
  <p>{{a}} is greater than {{b}}</p>
}

@if (a > b) {
  {{a}} is greater than {{b}}
} @else if (b > a) {
  {{a}} is less than {{b}}
} @else {
  {{a}} is equal to {{b}}
}

@if (user.profile.settings.startDate; as startDate) {
  {{ startDate }}
}
```

Repeat content with the @for block


```
@for (item of items; track item.id) {
  {{ item.name }}
}
```

Why is track in @for blocks important?

- The track expression allows Angular to maintain a relationship between your data and the DOM nodes on the page. This allows Angular to optimize performance by executing the minimum necessary DOM operations when the data changes.

- Using track effectively can significantly improve your application's rendering performance when looping over data collections.

- Select a property that uniquely identifies each item in the track expression. If your data model includes a uniquely identifying property, commonly id or uuid, use this value. If your data does not include a field like this, strongly consider adding one.

Contextual variables in @for blocks

VINSYS

- Inside @for blocks, several implicit variables are always available:

- $count - Number of items in a collection iterated over
- $index - Index of the current row
- $first - Whether the current row is the first row
- $last - Whether the current row is the last row
- $even - Whether the current row index is even
- $odd - Whether the current row index is odd

@empty block


```
@for (item of items; track item.name) {
  <li> {{ item.name }}</li>
} @empty {
  <li aria-hidden="true"> There are no items. </li>
}
```

@switch block

```
@switch (userPermissions) {
  @case ('admin') {
    <app-admin-dashboard />
  }
  @case ('reviewer') {
    <app-reviewer-dashboard />
  }
  @case ('editor') {
    <app-editor-dashboard />
  }
  @default {
    <app-viewer-dashboard />
  }
}
```

Adding event listeners

- Angular supports defining event listeners on an element in your template by specifying the event name inside parentheses along with a statement that runs every time the event occurs.

```
@Component({
  template: `
    <input type="text" (keyup)="updateField()" />
  `,
  ...
})
export class AppComponent{
  updateField(): void {
    console.log('Field is updated!');
  }
}
```

Event Argument

- In every template event listener, Angular provides a variable named $event that contains a reference to the event object.

```
@Component({
  template: `
    <input type="text" (keyup)="updateField($event)" />
  `,
  ...
})
export class AppComponent {
  updateField(event: KeyboardEvent): void {
    console.log(`The user pressed: ${event.key}`);
  }
}
```

Using key modifiers

```
@Component({
  template: `
    <input type="text" (keyup)="updateField($event)" />
  `,
  ...
})
export class AppComponent {
  updateField(event: KeyboardEvent): void {
    if (event.key === 'Enter') {
      console.log('The user pressed enter in the text field.');
    }
  }
}
```

Filter the Events

```
@Component({
  template: `
    <input type="text" (keyup.enter)="updateField($event)" />
  `,
  ...
})
export class AppComponent{
  updateField(event: KeyboardEvent): void {
    console.log('The user pressed enter in the text field.');
  }
}
```

Additional key modifiers

```
<!-- Matches shift and enter -->
<input type="text" (keyup.shift.enter)="updateField($event)" />

<!-- Matches alt and left shift -->
<input type="text" (keydown.code.alt.shiftleft)="updateField($event)" />
```

Pipes

- Pipes are a special operator in Angular template expressions that allows you to transform data declaratively in your template.

- Angular pipes use the vertical bar character (|)

```
import { Component } from '@angular/core';
import { CurrencyPipe, DatePipe, TitleCasePipe } from '@angular/common';
@Component({
  selector: 'app-root',
  imports: [CurrencyPipe, DatePipe, TitleCasePipe],
  template: `
    <main>
      <!-- Transform the company name to title-case and
      transform the purchasedOn date to a locale-formatted string -->
<h1>Purchases from {{ company | titlecase }} on {{ purchasedOn | date }}</h1>
                  <!-- Transform the amount to a currency-formatted string -->
      <p>Total: {{ amount | currency }}</p>
    </main>
  `,
})
export class ShoppingCartComponent {
  amount = 123.45;
  company = 'acme corporation';
  purchasedOn = '2024-07-08';
}
```

Combining multiple pipes in the same expression


```
<p>The event will occur on {{ scheduledOn | date | uppercase }}.</p>
```

Passing parameters to pipes


```
<p>The event will occur at {{ scheduledOn | date:'hh:mm' }}.</p>

<p>The event will occur at {{ scheduledOn | date:'hh:mm':'UTC' }}.</p>
```

Creating custom pipes

- You can define a custom pipe by implementing a TypeScript class with the @Pipe decorator. A pipe must have two things:
  - A name, specified in the pipe decorator
  - A method named transform that performs the value transformation.

```
// kebab-case.pipe.ts
import { Pipe, PipeTransform } from '@angular/core';
@Pipe({
  name: 'kebabCase',
})
export class KebabCasePipe implements PipeTransform {
  transform(value: string): string {
    return value.toLowerCase().replace(/ /g, '-');
  }
}
```

Directives

- Directives are classes that add additional behavior to elements in your Angular applications.

- Components
  - Used with a template. This type of directive is the most common directive type.

- Attribute directives
  - Change the appearance or behavior of an element, component, or another directive.

- Structural directives
  - Change the DOM layout by adding and removing DOM elements.

Built-in attribute directives

- NgClass
  - Adds and removes a set of CSS classes.
- NgStyle
  - Adds and removes a set of HTML styles.
- NgModel
  - Adds two-way data binding to an HTML form element.

Built-in structural directives

- NgIf - Conditionally creates or disposes of subviews from the template.
- NgFor - Repeat a node for each item in a list.
- NgSwitch - A set of directives that switch among alternative views.


```
<app-item-detail *ngIf="isActive" [item]="item"></app-item-detail>

<div *ngIf="currentCustomer">Hello, {{ currentCustomer.name }}</div>

<div *ngFor="let item of items">{{ item.name }}</div>

<div *ngFor="let item of items; let i=index">{{ i + 1 }} - {{ item.name }}</div>
```

Dependency Injection

- When you develop a smaller part of your system, like a module or a class, you may need to use features from other classes. For example, you may need an HTTP service to make backend calls.

- Dependency Injection, or DI, is a design pattern and mechanism for creating and delivering some parts of an application to other parts of an application that require them. Angular supports this design pattern and you can use it in your applications to increase flexibility and modularity.

Understanding dependency injection

- Two main roles exist in the DI system: dependency consumer and dependency provider.

- Angular facilitates the interaction between dependency consumers and dependency providers using an abstraction called Injector.

- When a dependency is requested, the injector checks its registry to see if there is an instance already available there. If not, a new instance is created and stored in the registry.

- Angular creates an application-wide injector (also known as the "root" injector) during the application bootstrap process.

Providing a dependency

- Consider a class called HeroService that needs to act as a dependency in a component.

- The first step is to add the @Injectable decorator to show that the class can be injected.

```
@Injectable()
class HeroService {}
```

At the Root Level

- The next step is to make it available in the DI by providing it. A dependency can be provided in multiple places:

- When you provide the service at the root level, Angular creates a single, shared instance of the HeroService and injects it into any class that asks for it.

```
@Injectable({
  providedIn: 'root'
})
class HeroService {}
```

At the Component level

- You can provide services at @Component level by using the providers field of the @Component decorator.

- In this case the HeroService becomes available to all instances of this component and other components and directives used in the template.

```
@Component({
  selector: 'hero-list',
  template: '...',
  providers: [HeroService]
})
class HeroListComponent {}
```

At the application root level using ApplicationConfig

- You can use the providers field of the ApplicationConfig (passed to the bootstrapApplication function) to provide a service or other Injectable at the application level.

```
export const appConfig: ApplicationConfig = {
    providers: [
      { provide: HeroService },
    ]
};
```

NgModule based applications

- @NgModule-based applications use the providers field of the @NgModule decorator to provide a service or other Injectable available at the application level.

- A service provided in a module is available to all declarations of the module, or to any other modules which share the same ModuleInjector.

Injecting/consuming a dependency

- Use Angular's inject function to retrieve dependencies.

```
import { inject } from "@angular/core";
export class HeroListComponent {
  private heroService = inject(HeroService);
}
```

When Angular discovers that a component depends on a service, it first checks if the injector has any existing instances of that service. If a requested service instance doesn't yet exist, the injector creates one using the registered provider, and adds it to the injector before returning the service to Angular.

Creating an injectable service

- ng generate service heroes/hero

Specifying a provider token

providers: [Logger]

- You can, however, configure DI to associate the Logger provider token with a different class or any other value. So when the Logger is injected, the configured value is used instead.

[{ provide: Logger, useClass: Logger }]

- The useClass provider key lets you create and return a new instance of the specified class.

Routing

- Routing helps you change what the user sees in a single-page app.
- In a single-page app, you change what the user sees by showing or hiding portions of the display that correspond to particular components, rather than going out to the server to get a new page.
- As users perform application tasks, they need to move between the different views that you have defined.
- To handle the navigation from one view to the next, you use the Angular Router. The Router enables navigation by interpreting a browser URL as an instruction to change the view.

Define your routes in your Routes array

- Each route in this array is a JavaScript object that contains two properties. The first property, path, defines the URL path for the route. The second property, component, defines the component Angular should use for the corresponding path.

```
const routes: Routes = [
  { path: 'first-component', component: FirstComponent },
  { path: 'second-component', component: SecondComponent },
];
```

ProvideRouter

- Import the routes into app.config.ts and add it to the provideRouter function. The following is the default ApplicationConfig using the CLI.

```
export const appConfig: ApplicationConfig = {
  providers: [provideRouter(routes)]
};
```

Add your routes to your application

```
<h1>Angular Router App</h1>
<nav>
  <ul>
    <li><a routerLink="/first-component" >First Component</a></li>
    <li><a routerLink="/second-component" >Second Component</a></li>
  </ul>
</nav>
<!-- The routed views render in the <router-outlet>-->
<router-outlet />
```

Setting up wildcard routes

- A well-functioning application should gracefully handle when users attempt to navigate to a part of your application that does not exist.

- To add this functionality to your application, you set up a wildcard route. The Angular router selects this route any time the requested URL doesn't match any router paths.

{ path: '**', component: <component-name> }

Displaying a 404 page


```
const routes: Routes = [
  { path: 'first-component', component: FirstComponent },
  { path: 'second-component', component: SecondComponent },
  { path: '**', component: PageNotFoundComponent },  // Wildcard route for a 404 page
];
```

Setting up redirects

- To set up a redirect, configure a route with the path you want to redirect from, the component you want to redirect to, and a pathMatch value that tells the router how to match the URL.

```
const routes: Routes = [
  { path: 'first-component', component: FirstComponent },
  { path: 'second-component', component: SecondComponent },
  { path: '',   redirectTo: '/first-component', pathMatch: 'full' }, // redirect to `first-component`
  { path: '**', component: PageNotFoundComponent },  // Wildcard route for a 404 page
];
```

Nesting routes

- As your application grows more complex, you might want to create routes that are relative to a component other than your root component. These types of nested routes are called child routes.

```
<h2>First Component</h2>
<nav>
  <ul>
    <li><a routerLink="child-a">Child A</a></li>
    <li><a routerLink="child-b">Child B</a></li>
  </ul>
</nav>
<router-outlet />
```

```
const routes: Routes = [
 {
   path: 'first-component',
   component: FirstComponent, // this is the component with the <router-outlet> in the template
   children: [
    {
      path: 'child-a', // child route path
      component: ChildAComponent, // child route component that the router renders
    },
    {
      path: 'child-b',
      component: ChildBComponent, // another child route component that the router renders
    },
   ],
 },
];
```

Setting the page title

```
const routes: Routes = [
 {
   path: 'first-component',
   title: 'First component',
   component: FirstComponent,  // this is the component with the <router-outlet> in the template
   children: [
    {
      path: 'child-a',  // child route path
      title: resolvedChildATitle,
      component: ChildAComponent,  // child route component that the router renders
    },
    {
      path: 'child-b',
      title: 'child b',
      component: ChildBComponent,  // another child route component that the router renders
    },
   ],
 },
];
const resolvedChildATitle: ResolveFn<string> = () => Promise.resolve('child a');
```

Accessing query parameters and fragments

- Sometimes, a feature of your application requires accessing a part of a route, such as a query parameter or a fragment. In this example, the route contains an id parameter we can use to target a specific hero page.

```
export const routes: Routes = [
  { path: 'hero/:id', component: HeroDetailComponent }
];
export const appConfig: ApplicationConfig = {
  providers: [provideRouter(routes)],
};
```

```
private readonly route = inject(ActivatedRoute);
private readonly router = inject(Router);
hero$: Observable<Hero>;
ngOnInit() {
  const heroId = this.route.snapshot.paramMap.get('id');
  this.hero$ = this.service.getHero(heroId);
}
gotoItems(hero: Hero) {
  const heroId = hero ? hero.id : null;
  // Pass along the hero id if available
  // so that the HeroList component can select that item.
  this.router.navigate(['/heroes', { id: heroId }]);
}
```

Lazy loading

- You can configure your routes to lazy load modules, which means that Angular only loads modules as needed, rather than loading all modules when the application launches.

```
const routes: Routes = [
  {
    path: 'lazy',
    loadComponent: () => import('./lazy.component').then(c => c.LazyComponent)
  }
];
```

Preventing unauthorized access

- Use route guards to prevent users from navigating to parts of an application without authorization. The following route guards are available in Angular:
  - canActivate
  - canActivateChild
  - canDeactivate
  - canMatch
  - Resolve
  - canLoad

ng generate guard your-guard

```
{
  path: '/your-path',
  component: YourComponent,
  canActivate: [yourGuardFunction],
}
```

Link parameters array

- A link parameters array holds the following ingredients for router navigation:
  - The path of the route to the destination component
  - Required and optional route parameters that go into the route URL

```
<a [routerLink]="['/heroes']">Heroes</a>
<a [routerLink]="['/hero', hero.id]">
  <span class="badge">{{ hero.id }}</span>{{ hero.name }}
</a>
```

Forms

- Applications use forms to enable users to log in, to update a profile, to enter sensitive information, and to perform many other data-entry tasks.

- Angular provides two different approaches to handling user input through forms: reactive and template-driven.

- Both capture user input events from the view, validate the user input, create a form model and data model to update, and provide a way to track changes.

Template-driven forms

- Rely on directives in the template to create and manipulate the underlying object model.
- They are useful for adding a simple form to an app, such as an email list signup form.
- They're straightforward to add to an app, but they don't scale as well as reactive forms.
- If you have very basic form requirements and logic that can be managed solely in the template, template-driven forms could be a good fit.

Reactive Forms

- Provide direct, explicit access to the underlying form's object model.
- Compared to template-driven forms, they are more robust: they're more scalable, reusable, and testable.
- If forms are a key part of your application, or you're already using reactive patterns for building your application, use reactive forms.

Setup in template-driven forms

- In template-driven forms, the form model is implicit, rather than explicit. The directive NgModel creates and manages a FormControl instance for a given form element.

```
import {Component} from '@angular/core';
import {FormsModule} from '@angular/forms';
@Component({
  selector: 'app-template-favorite-color',
  template: `
    Favorite Color: <input type="text" [(ngModel)]="favoriteColor">
  `,
  imports: [FormsModule],
})
export class FavoriteColorTemplateComponent {
  favoriteColor = '';
}
```

Setup in reactive forms

- With reactive forms, you define the form model directly in the component class. The [formControl] directive links the explicitly created FormControl instance to a specific form element in the view, using an internal value accessor.

```
import {Component} from '@angular/core';
import {FormControl, ReactiveFormsModule} from '@angular/forms';
@Component({
  selector: 'app-reactive-favorite-color',
  template: `
    Favorite Color: <input type="text" [formControl]="favoriteColorControl">
  `,
  imports: [ReactiveFormsModule],
})
export class FavoriteColorReactiveComponent {
  favoriteColorControl = new FormControl('');
}
```

Create a FormGroup instance

```
import {Component} from '@angular/core';
import {FormGroup, FormControl, ReactiveFormsModule} from '@angular/forms';
@Component({
  selector: 'app-profile-editor',
  templateUrl: './profile-editor.component.html',
  styleUrls: ['./profile-editor.component.css'],
  imports: [ReactiveFormsModule],
})
export class ProfileEditorComponent {
  profileForm = new FormGroup({
    firstName: new FormControl(''),
    lastName: new FormControl(''),
...
  });
...
}
```

Associate the FormGroup model and view

```
<form [formGroup]="profileForm">
  <label for="first-name">First Name: </label>
  <input id="first-name" type="text" formControlName="firstName" />
  <label for="last-name">Last Name: </label>
  <input id="last-name" type="text" formControlName="lastName" />
...
</form>
```

Creating nested form groups

```
import {Component} from '@angular/core';
import {FormGroup, FormControl, ReactiveFormsModule} from '@angular/forms';
@Component({
  selector: 'app-profile-editor',
  templateUrl: './profile-editor.component.html',
  styleUrls: ['./profile-editor.component.css'],
  imports: [ReactiveFormsModule],
})
export class ProfileEditorComponent {
  profileForm = new FormGroup({
    firstName: new FormControl(''),
    lastName: new FormControl(''),
    address: new FormGroup({
      street: new FormControl(''),
      city: new FormControl(''),
      state: new FormControl(''),
      zip: new FormControl(''),
    }),
  });
...
}
```

Group the nested form in the template

```
<div formGroupName="address">
   <h2>Address</h2>
   <label for="street">Street: </label>
   <input id="street" type="text" formControlName="street" />
   <label for="city">City: </label>
   <input id="city" type="text" formControlName="city" />
   <label for="state">State: </label>
   <input id="state" type="text" formControlName="state" />
   <label for="zip">Zip Code: </label>
   <input id="zip" type="text" formControlName="zip" />
 </div>
```

Validating input in template-driven forms

```
<input type="text" id="name" name="name" class="form-control"
          required minlength="4" appForbiddenName="bob"
          [(ngModel)]="actor.name" #name="ngModel">
      @if (name.invalid && (name.dirty || name.touched)) {
        <div class="alert">
          @if (name.hasError('required')) {
            <div>
              Name is required.
            </div>
          }
          @if (name.hasError('minlength')) {
            <div>
              Name must be at least 4 characters long.
            </div>
          }
          @if (name.hasError('forbiddenName')) {
            <div>
              Name cannot be Bob.
            </div>
          }
        </div>
      }
```

Validating input in reactive forms

```
actorForm: FormGroup = new FormGroup({
   name: new FormControl(this.actor.name, [
     Validators.required,
     Validators.minLength(4),
     forbiddenNameValidator(/bob/i), // <-- Here's how you pass in the custom validator.
   ]),
   role: new FormControl(this.actor.role),
   skill: new FormControl(this.actor.skill, Validators.required),
 });
 get name() {
   return this.actorForm.get('name');
 }
 get skill() {
   return this.actorForm.get('skill');
 }
```

```
<input type="text" id="name" class="form-control"
            formControlName="name" required>
      @if (name.invalid && (name.dirty || name.touched)) {
        <div class="alert alert-danger">
          @if (name.hasError('required')) {
            <div>
              Name is required.
            </div>
          }
          @if (name.hasError('minlength')) {
            <div>
              Name must be at least 4 characters long.
            </div>
          }
          @if (name.hasError('forbiddenName')) {
            <div>
              Name cannot be Bob.
            </div>
          }
        </div>
      }
```

Deferred loading with @defer

- Deferrable views, also known as @defer blocks, reduce the initial bundle size of your application by deferring the loading of code that is not strictly necessary for the initial rendering of a page. This often results in a faster initial load.

```
 @defer {
  <large-component />
}

@defer {
  <large-component />
} @placeholder {
  <p>Placeholder content</p>
}
```

```
@defer {
  <large-component />
} @placeholder (minimum 500ms) {
  <p>Placeholder content</p>
}

@defer {
  <large-component />
} @loading {
  <img alt="loading..." src="loading.gif" />
} @placeholder {
  <p>Placeholder content</p>
}
```

```
@defer (on interaction) {
  <large-cmp />
} @placeholder {
  <div>Large component placeholder</div>
}

@defer (on hover) {
  <large-cmp />
} @placeholder {
  <div>Large component placeholder</div>
}

@defer (on immediate) {
  <large-cmp />
} @placeholder {
  <div>Large component placeholder</div>
}
```

HttpClient

- Most front-end applications need to communicate with a server over the HTTP protocol, to download or upload data and access other back-end services.

- Angular provides a client HTTP API for Angular applications, the HttpClient service class in @angular/common/http.

- HttpClient is provided using the provideHttpClient helper function, which most apps include in the application providers in app.config.ts.

```
export const appConfig: ApplicationConfig = {
  providers: [
    provideHttpClient(),
  ]
};
```

Configuring features of HttpClient

- By default, HttpClient uses the XMLHttpRequest API to make requests. The withFetch feature switches the client to use the fetch API instead.

```
export const appConfig: ApplicationConfig = {
  providers: [
    provideHttpClient(
      withFetch(),
    ),
  ]
};
```

Making HTTP requests

```
http.get<Config>('/api/config').subscribe(config => {
  // process the configuration.
});
```

Fetching other types of data

- 'json' (default) - JSON data of the given generic type
- 'text' - string data
- 'arraybuffer‘ - ArrayBuffer containing the raw response bytes
- 'blob' - Blob instance

```
http.get('/images/dog.jpg', {responseType: 'arraybuffer'}).subscribe(buffer => {
  console.log('The image is ' + buffer.byteLength + ' bytes large');
});
```

Setting request headers

- Specify request headers that should be included in the request using the headers option.
- Passing an object literal is the simplest way of configuring request headers:

```
http.get('/api/config', {
  headers: {
    'X-Debug-Level': 'verbose',
  }
}).subscribe(config => {
  // ...
});
```

Interceptors

- Interceptors are middleware that allows common patterns around retrying, caching, logging, and authentication to be abstracted away from individual requests.

- Interceptors are generally functions which you can run for each request, and have broad capabilities to affect the contents and overall flow of requests and responses.

- You can install multiple interceptors, which form an interceptor chain where each interceptor processes the request or response before forwarding it to the next interceptor in the chain.

Defining an interceptor

- The basic form of an interceptor is a function which receives the outgoing HttpRequest and a next function representing the next processing step in the interceptor chain.

```
export function loggingInterceptor(req: HttpRequest<unknown>, next: HttpHandlerFn):
Observable<HttpEvent<unknown>> {
  console.log(req.url);
  return next(req);
}
```

Configuring interceptors

```
bootstrapApplication(AppComponent, {providers: [
  provideHttpClient(
    withInterceptors([loggingInterceptor, cachingInterceptor]),
  )
]});
```

Signal

- let x = 5;
- let y = 3;
- let z = x + y;
- console.log(z);

- x = 10;
- console.log(z);
- What if we want our variables to react to changes!

Signal

- A signal is a wrapper around a value that can notify interested consumers when that value changes.
- A signal is a variable + change notification
- A signal is reactive, and is called a "reactive primitive"
- A signal always has a value
- A signal is synchronous

```
const count = signal(0);

// Signals are getter functions - calling them reads their value.
console.log('The count is: ' + count());

To change the value of a writable signal, you can either .set() it directly:
count.set(3);
```

```
const x = signal(5);
const y = signal(3);
const z = computed(() => x() + y());
console.log(z()); // 8

x.set(10);
console.log(z()); // 13
```

```
quantity = signal(1);

qtyAvailable = signal([1, 2, 3, 4, 5, 6]);

selectedVehicle = signal<Vehicle>({
  id: 1,
  name: 'AT-AT',
  price: 19416.13
});

vehicles = signal<Vehicle[]>([]);
```

Effects

- An effect is an operation that runs whenever one or more signal values change. You can create an effect with the effect function:

```
effect(() => {
  console.log(`The current count is: ${count()}`);
});
```

```
@Component({
  signals: true,
  selector: 'counter',
  template: `
  <p>Counter: {{ count() }}</p>
  <button (click)="increment()">+1</button>
  <button (click)="decrement()">-1</button>`
})
export class CounterComponent {
  count = signal(0);
  increment(): void {
   this.count.update(value => value + 1);
  }
  decrement(): void {
   this.count.update(value => value - 1);
  }
}
```

```
const user = signal<User>({ id: '70a65491c1d6', name: 'John', age: 35 });
const isAdult = computed(() => user().age >= 18)); //Signal<boolean>
const color = computed(() => isAdult() ? 'green' : 'red'); //Signal<'green' | 'red'>

user.set({ id: 'c37de3232c4d', name: 'Andy', age: 22 });
```

Automatic Deployment

The Angular CLI command ng deploy executes the deploy CLI builder associated with your project. A number of third-party builders implement deployment capabilities to different platforms. You can add any of them to your project with ng add.

ng add @angular/fire

ng deploy

Manual deployment

- To manually deploy your application, create a production build and copy the output directory to a web server or content delivery network (CDN). By default, ng build uses the production configuration.

- ng build outputs the built artifacts to dist/my-app/ by default, however this path can be configured with the outputPath option in the @angular-devkit/build-angular:browser builder. Copy this directory to the server and configure it to serve the directory.

Testing

- The Angular CLI downloads and installs everything you need to test an Angular application with Jasmine testing framework.

- The ng test command builds the application in watch mode, and launches the Karma test runner.

- Inside the src/app folder the Angular CLI generated a test file for the AppComponent named app.component.spec.ts.

Code Coverage

- The Angular CLI can run unit tests and create code coverage reports. Code coverage reports show you any parts of your code base that might not be properly tested by your unit tests.

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

- When the tests are complete, the command creates a new /coverage directory in the project. Open the index.html file to see a report with your source code and code coverage values.

Testing in continuous integration

- Continuous integration (CI) servers let you set up your project repository so that your tests run on every commit and pull request.

$ ng test --no-watch --no-progress --browsers=ChromeHeadless

Testing services

- To check that your services are working as you intend, you can write tests specifically for them.

```
describe('ValueService', () => {
  let service: ValueService;
  beforeEach(() => {
    service = new ValueService();
  });
  it('#getValue should return real value', () => {
    expect(service.getValue()).toBe('real value');
  });
```

```
it('#getObservableValue should return value from observable', (done: DoneFn) => {
    service.getObservableValue().subscribe((value) => {
      expect(value).toBe('observable value');
      done();
    });
  });

  it('#getPromiseValue should return value from a promise', (done: DoneFn) => {
    service.getPromiseValue().then((value) => {
      expect(value).toBe('promise value');
      done();
    });
  });
});
```

Services with dependencies

```
@Injectable()
export class MasterService {
  public valueService = inject(ValueService);
  getValue() {
    return this.valueService.getValue();
  }
}
```

- MasterService delegates its only method, getValue, to the injected ValueService.

```
describe('MasterService without Angular testing support', () => {
    let masterService: MasterService;
    it('#getValue should return real value from the real service', () => {
      masterService = new MasterService(new ValueService());
      expect(masterService.getValue()).toBe('real value');
    });
```

Angular TestBed

- The TestBed is the most important of the Angular testing utilities. The TestBed creates a dynamically-constructed Angular test module that emulates an Angular @NgModule.

- The TestBed.configureTestingModule() method takes a metadata object that can have most of the properties of an @NgModule.

```
let service: ValueService;
   beforeEach(() => {
     TestBed.configureTestingModule({providers: [ValueService]});
...
   });
```

Component Testing

```
@Component({
  selector: 'app-banner',
  template: '<h1>{{title()}}</h1>',
  styles: ['h1 { color: green; font-size: 350%}'],
})
export class BannerComponent {
  title = signal('Test Tour of Heroes');
}
```

```
let component: BannerComponent;
  let fixture: ComponentFixture<BannerComponent>;
  let h1: HTMLElement;
  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [BannerComponent],
    });
    fixture = TestBed.createComponent(BannerComponent);
    component = fixture.componentInstance; // BannerComponent test instance
    h1 = fixture.nativeElement.querySelector('h1');
  });
it('should display original title', () => {
...
    expect(h1.textContent).toContain(component.title);
  });
```

Thank You

suryakant.surve@vinsys.com