Introduction to Angular & Web Fundamentals

A complete educational overview of HTML, CSS, and Angular Framework

CODETANTRA

Table of Contents

CODETANTRA

Revision of HTML

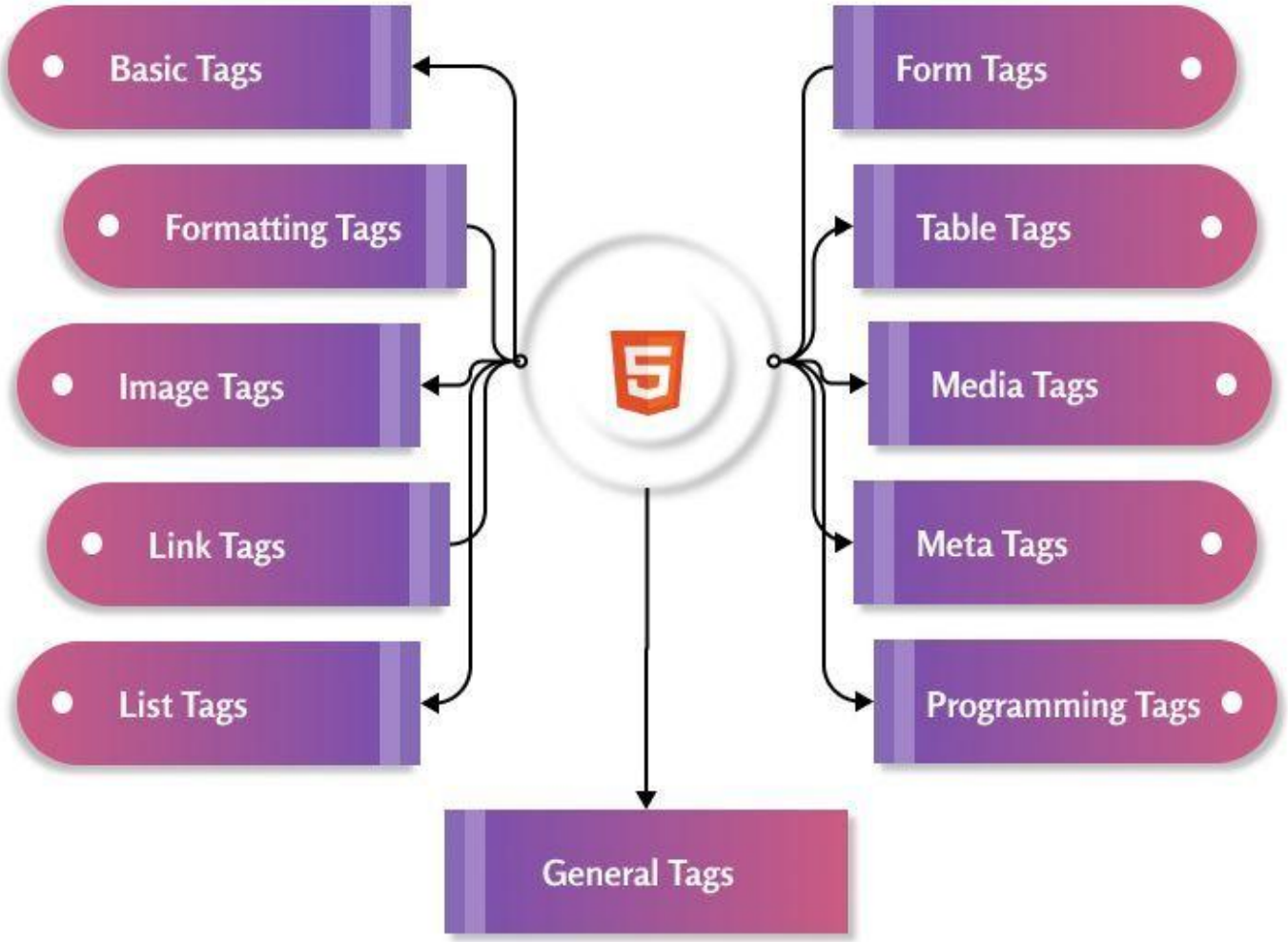
What is HTML?

- HTML (**HyperText Markup Language**) is the standard markup language for creating web pages.
- Defines the **structure and content** of a webpage.
- Uses **tags** (elements) enclosed within < > brackets

Key Features:

- Platform-independent and interpreted by browsers.
- Works with CSS (for styling) and JavaScript (for interactivity).
- HTML5 adds semantic and multimedia support.

HTML Tags List



CODETANTRA

HTML Code Example

Basic Structure of an HTML Document:

```
1  <!DOCTYPE html>
2  <html lang="en">
3    <head>
4      <meta charset="UTF-8">
5      <title>My First Page</title>
6    </head>
7    <body>
8      <h1>Welcome to HTML</h1>
9      <p>This is a paragraph.</p>
10   </body>
11 </html>
```

Explanation:

<!DOCTYPE html> → Defines HTML5 document type.
<html> → Root element.
<head> → Metadata (title, styles, scripts).
<body> → Visible page content.

CODETANTRA

HTML Tags and Their Descriptions (1/2)

Tag	Description	Example
<html>	Root element of an HTML page	<html> ... </html>
<head>	Contains metadata, title, links, scripts	<head> ... </head>
<title>	Sets the page title	<title>My Page</title>
<body>	Contains all visible page content	<body> ... </body>
<h1>-<h6>	Headings from largest to smallest	Hello
	Defines a paragraph	This is a paragraph.
<a>	Creates a hyperlink	Link
<img>	Embeds an image	<img src='image.jpg' alt='desc'>
, ,	Unordered / ordered lists	Item

CODETANTRA

HTML Tags and Their Descriptions (2/2)

Tag	Description	Example
	Defines a division or section	Content
	Inline container for text styling	Text
<table>	Creates a table	<table>...</table>
<tr>, <td>, <th>	Table row, data cell, and header	<tr><td>Data</td></tr>
<form>	Creates an input form	<form>...</form>
<input>	Defines input fields	<input type='text'>
<button>	Clickable button	<button>Click</button>
 	Inserts a line break	
<hr>	Inserts a horizontal line	<hr>
/	Bold or italic text	Text

HTML5 Semantic Tags:

<header> – Defines page header.
<nav> – Navigation bar.
<article> – Self-contained content (e.g., blog post).
<section> – Logical grouping of content.
<footer> – Footer section.

Media Tags:

<audio>, <video> – Embed media files.
<canvas> – Draw graphics using JavaScript.

Revision of CSS

What is CSS?

- CSS (**Cascading Style Sheets**) controls the **look and feel** of HTML pages.
- Separates **content (HTML)** from **presentation (design)**.
- Improves consistency, reusability, and ease of maintenance.

How CSS Works:

CSS rules are applied by **selectors** that target HTML elements.

Styles can be added:

Inline: Using the style attribute inside HTML tags.
Internal: Inside <style> tags within the <head> section.
External: Using **.css** files linked via <link rel="stylesheet" href="style.css">.

CSS Example

Basic Syntax:

```
selector {
  property: value;
}
```

Example:

```
h1 {
  color: #dd1b16;
  text-align: center;
  font-family: Arial, sans-serif;
}
```

Explanation:

Selector: Targets HTML elements (e.g., h1, .class, #id).
Property: The aspect you want to style (e.g., color, font-size).
Value: The setting applied to the property.

CSS Selectors - Basic Types

Selector Type	Syntax Example	Description
Universal Selector	* { margin: 0; }	Selects all elements on a page.
Element Selector	p { color: blue; }	Selects all <p> (paragraph) elements.
ID Selector	#title { font-size: 24px; }	Selects an element with a specific ID.
Class Selector	.highlight { background: yellow; }	Selects all elements with the specified class.
Group Selector	h1, h2, h3 { color: green; }	Applies the same style to multiple elements.

CSS Selectors - Advanced Types

Selector Type	Syntax Example	Description
Descendant Selector	div p { color: red; }	Selects <p> inside .
Child Selector	div > p { color: red; }	Selects direct child <p> of .
Adjacent Sibling Selector	h1 + p { color: blue; }	Selects the first <p> immediately after <h1>.
General Sibling Selector	h1 ~ p { color: gray; }	Selects all <p> elements after <h1>.
Attribute Selector	input[type="text"] { border: 1px solid; }	Selects elements based on attribute values.
Pseudo-class Selector	a:hover { color: red; }	Styles an element in a specific state.
Pseudo-element Selector	p::first-line { font-weight: bold; }	Styles a specific part of an element.

CSS – Box Model, Layouts

CSS Box Model:

Every HTML element is a box made up of:

Content – The actual text or image.

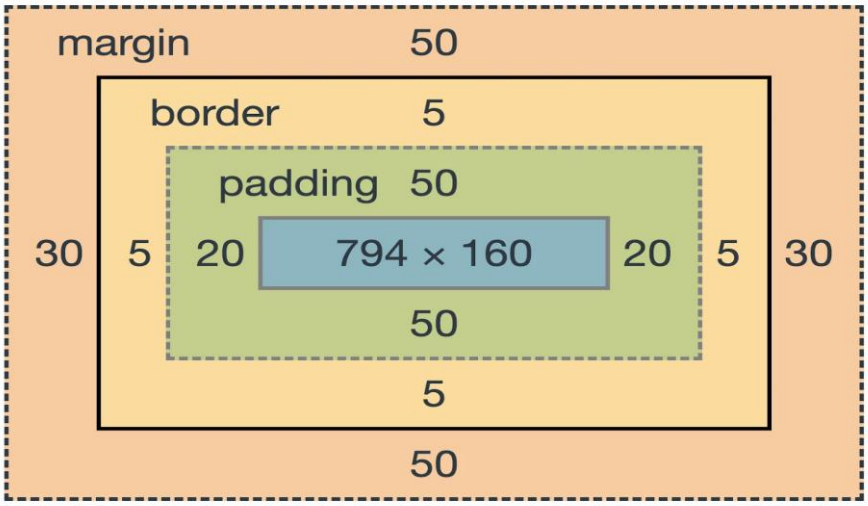
Padding – Space between content and border.

Border – Edge surrounding the padding.

Margin – Space outside the border.

Example Diagram:

(image showing Box Model layers — margin → border → padding → content)



CODETANTRA

CSS Example

Layout Techniques:

Display Property: block, inline, inline-block, none

Positioning: static, relative, absolute, fixed, sticky

Flexbox: Simplifies responsive layout design.

```
.container {
  display: flex;
  justify-content: center;
  align-items: center;
}
```

Grid Layout: Defines 2D layouts with rows & columns.

```
grid {
  display: grid;
  grid-template-columns: 1fr 1fr 1fr;
}
```

Introduction to Angular

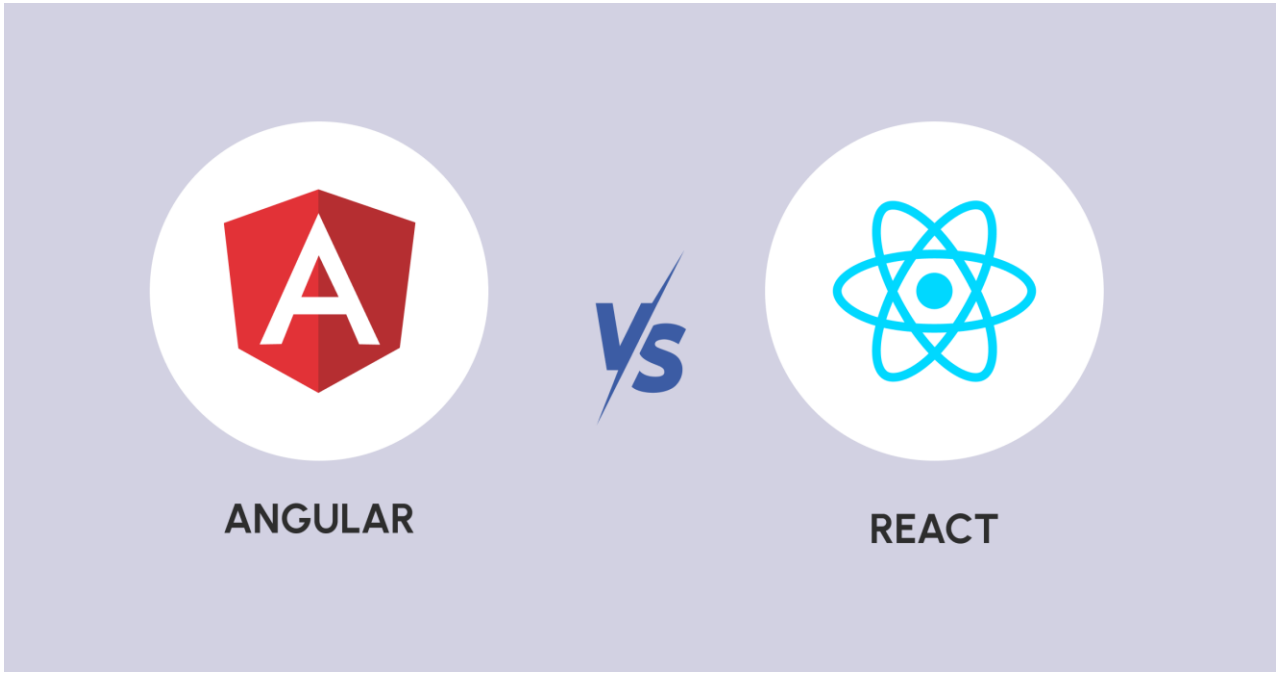
- Angular is a powerful Typescript based open-source web framework developed and maintained by Google, designed for building dynamic and scalable single-page applications (SPAs).

Why Angular?

- **Component-based architecture:** modular, reusable, and easy to maintain.
- **Reusable code & modular design:** promotes scalability and better structure.
- **Dependency Injection (DI):** efficient service management.
- **Powerful CLI tools:** automate project setup and component generation.
- **MVVM pattern:** separates UI from logic for maintainability.
- **Built-in testing tools:** support unit and integration testing.
- **TypeScript-first approach:** reduces bugs and enhances developer experience.

Angular vs React

- **Angular** — Developed by Google, a full-fledged framework.
- **React** — Developed by Meta, a flexible UI library.



Angular vs React - Visual Comparison

Feature	Angular	React
Type	Full-fledged Framework	JavaScript Library
Language	TypeScript	JavaScript + JSX
Architecture	MVC / MVVM	Component-based
DOM	Real DOM (Ivy Renderer)	Virtual DOM
Data Binding	Two-way	One-way
Performance	Stable but heavier	Lightweight, faster UI updates
Learning Curve	Steeper	Easier for beginners
Mobile Support	Ionic	React Native

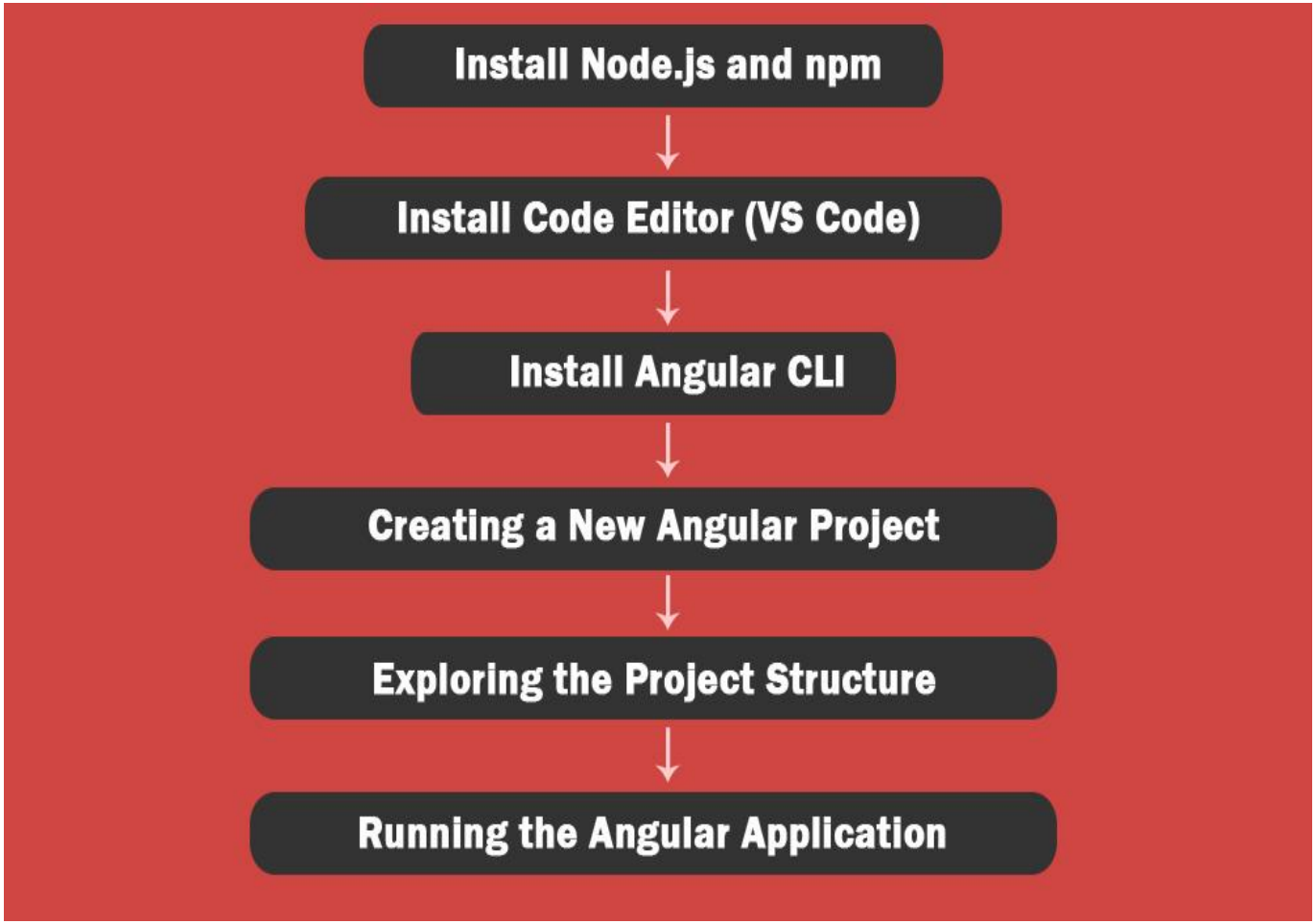
Comparison summary: Angular offers a complete ecosystem, while React focuses on flexibility and UI performance.

Key Features of Angular

Feature	Description
Component-Based Architecture	Applications are divided into reusable and modular components. Each component has its own logic, template, and style.
Two-Way Data Binding	Automatically synchronizes data between model and view, reducing the need for manual DOM updates.
Dependency Injection (DI)	Manages services and dependencies efficiently, improving reusability and maintainability.
Directives	Extend HTML with custom behavior. Structural (e.g., *ngIf, *ngFor) and Attribute (e.g., [ngClass], [ngStyle]).
Routing	Enables Single Page Application (SPA) navigation using Angular Router.

Feature	Description
RxJS and Reactive Programming	Manages asynchronous data streams and events using Observables.
Ahead-of-Time (AOT) Compilation	Compiles TypeScript and templates into optimized JavaScript before runtime for better performance.
Angular CLI	Command-line tool for project setup, development, testing, and deployment (ng serve, ng build, etc.).
Cross-Platform Development	Enables building web, mobile, and desktop applications from a single codebase.
Strong TypeScript Support	Built with TypeScript for type safety, scalability, and advanced IDE support.

Setting up Angular Application



Prerequisites

- Install Node.js and npm
- Verify installation: node -v, npm -v
- Use Visual Studio Code or similar IDE
- Basic knowledge: TypeScript, HTML, CSS
- Ensure stable internet connection

Tip: Prefer LTS version of Node.js

Angular CLI - Command Summary

Command	Purpose
npm install -g @angular/cli	Install Angular CLI globally
ng new <project-name>	Create a new Angular project
ng serve	Run development server
ng build	Build the application
ng build --prod	Build project for production
ng generate component <name>	Generate a new component
ng generate service <name>	Generate a new service
ng generate module <name>	Generate a new module
ng generate directive <name>	Generate a new directive
ng generate pipe <name>	Generate a new pipe
ng generate class <name>	Generate a new class
ng generate enum <name>	Generate a new enum

Tip: Angular CLI automates project setup, builds, and deployment tasks for faster development.

Install Angular CLI

- Angular CLI simplifies project creation and management.

Command:

```
npm install -g @angular/cli
```

Verify:

```
ng version
```

Purpose:

Automates creation, builds, and deployments.

Create a New Angular Project

- Command:
  `ng new my-angular-app`

  Setup Prompts:
  - Add routing? Yes
  - Choose stylesheet: CSS / SCSS / SASS

  Project Structure Example:

```
my-angular-app/
├── src/
│   ├── app/
│   ├── assets/
│   └── environments/
├── angular.json
├── package.json
```

Navigate and Run Application

- Commands:

  ```
  cd my-angular-app
  ng serve
  ```

  Default URL:

  http://localhost:4200

  Browser auto-reloads on file changes.

Generate Components

- Angular CLI helps generate key files automatically.

Examples:

```
ng generate component home
ng generate service data
ng generate module user
```

Benefit:

Saves time and ensures consistent structure.

Build and Deploy

- Build for production:

  ng build --prod

  Output:

  /dist folder with optimized files

  Deploy (example Firebase):

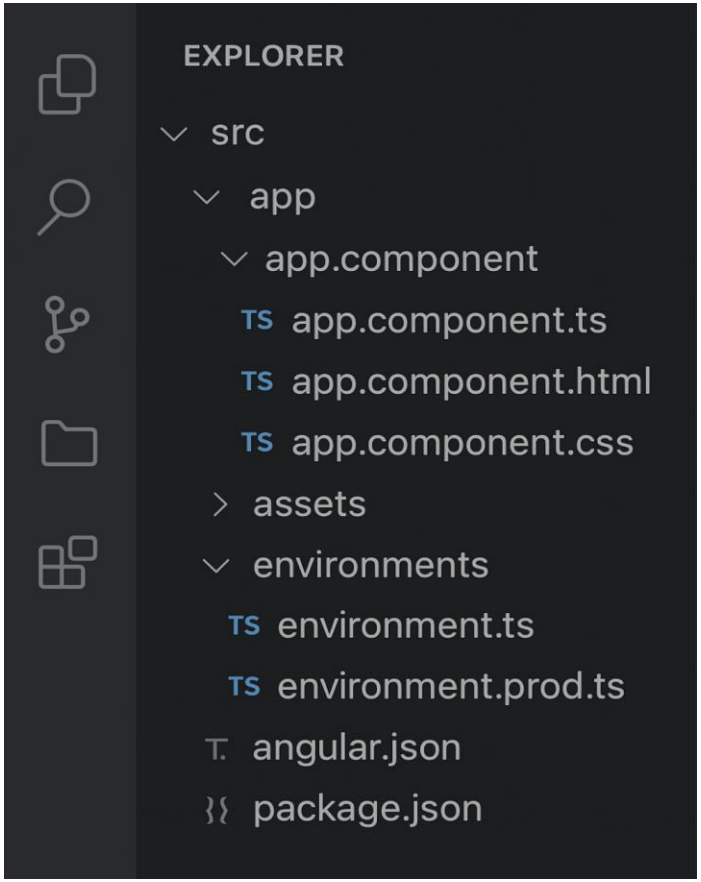
  firebase deploy

Angular Project Structure

Key Folders:

- src/app: Application code
- src/assets: Static files
- src/environments: Environment configs
- angular.json: Build & config file
- package.json: Dependencies

Each component has .ts, .html, and .css files.

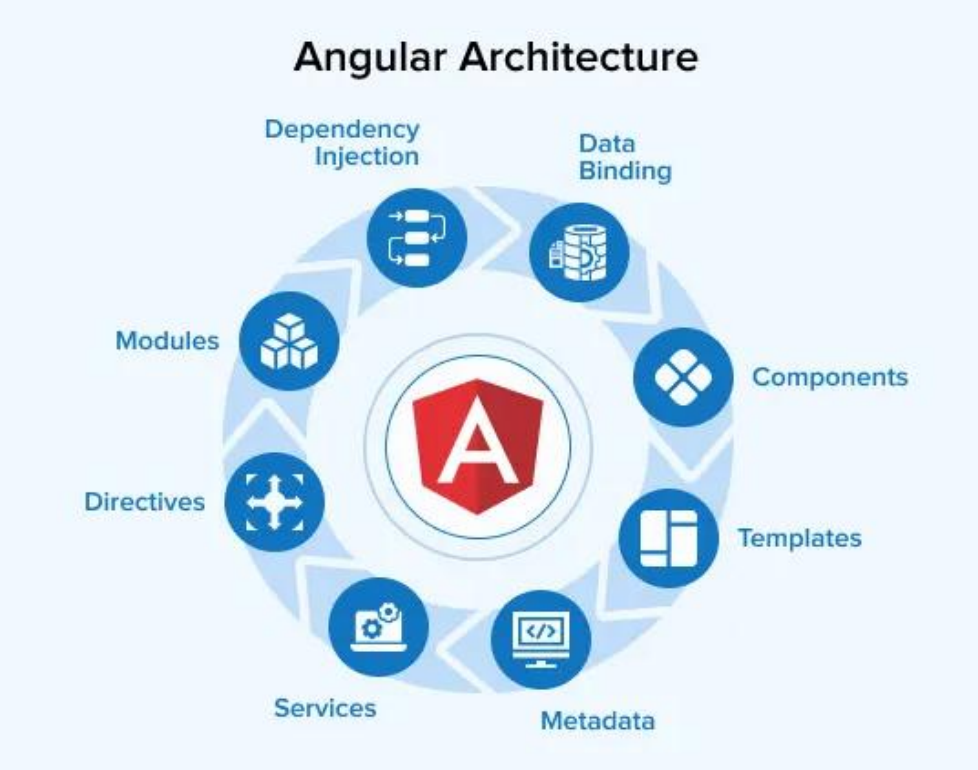


Angular Project Structure

Folder / File	Description
src/	Main source folder containing all application code.
src/app/	Houses the core app logic—components, modules, services, and routing files.
src/assets/	Stores static assets such as images, icons, and fonts.
src/environments/	Contains environment configuration files like environment.ts and environment.prod.ts.
angular.json	Project configuration file defining build options and file paths.
package.json	Lists dependencies, project metadata, and npm scripts.
tsconfig.json	Specifies TypeScript compiler options and paths.
node_modules/	Holds all installed npm packages required for the project.
main.ts	Application entry point that bootstraps the Angular app.
index.html	Main HTML file where Angular renders the application using <app-root>.
styles.css / styles.scss	Global stylesheet applied across all components.

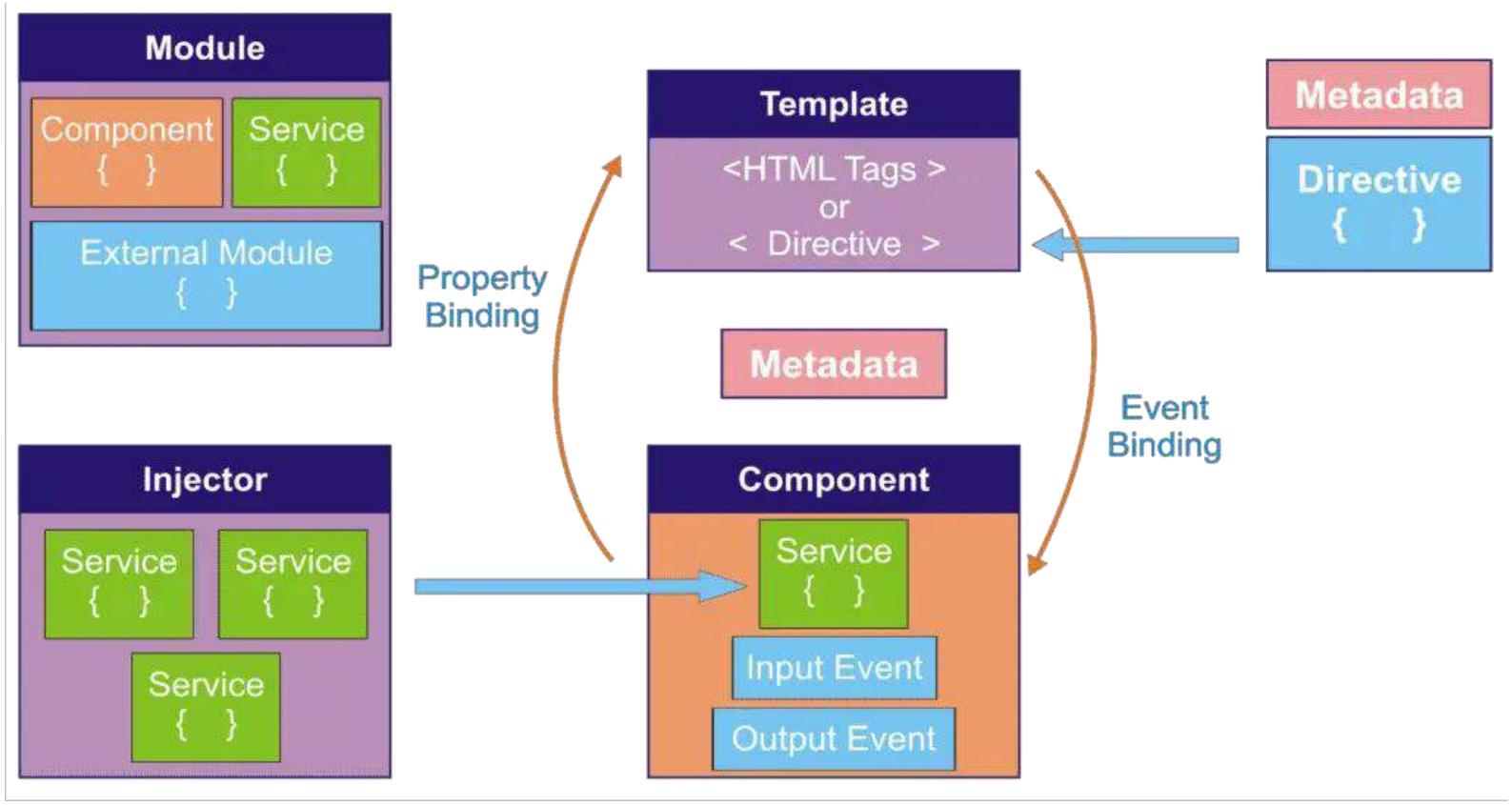
Angular Architecture

Modules (NgModule)
↓
Components
↓
Templates & Data Binding
↓
Directives
↓
Services & Dependency Injection
↓
Routing



Each component has an HTML template, CSS, and a TypeScript class.

Angular Architecture



Modules (NgModule)

- Modules (NgModule) organize an app into logical sections.
- Each module can import or export components, directives, and services.
- The root module (AppModule) bootstraps the application.
- Supports lazy loading for performance optimization.
- **Example:**

```
@NgModule({
    declarations: [AppComponent],
   imports: [BrowserModule],
   bootstrap: [AppComponent]
  })
```

Components

- Core building block of Angular – defines UI and logic.
- Each component has a TypeScript class, HTML template, and CSS style.
- Reusable and modular, controlling a portion of the view.
- **Example:**

```
1  @Component({
2           selector: 'app-hero', templateUrl: './hero.component.html'
3          })
```

Templates & Data Binding

- Templates define the HTML structure of a component view.
- Data binding connects component logic with the view.
- Types:
  - Interpolation {{ }}
  - Property binding [ ]
  - Event binding ( ),
  - Two-way binding[( )].
- **Example:**

```
1  <input [(ngModel)]='userName'>
2  <p>Hello {{userName}}!</p>
```

Directives

- Directives extend HTML with custom behavior.
- Structural Directives: modify DOM structure (e.g., *ngIf, *ngFor).
- Attribute Directives: modify element behavior/style (e.g., ngStyle, ngClass).
- **Example**:

```
1   <li *ngFor='let item of items'>{{item}}</li>
```

Services

- Services handle reusable logic and data across components.
- Ideal for API calls, data management, and shared functionality.
- Encourages separation of concerns and clean architecture.
- **Example**:

```
1  @Injectable({ providedIn: 'root' })
2  export class DataService { getData() { ... } }
```

Dependency Injection (DI)

- Dependency Injection (DI) supplies components with required services automatically.
- Promotes reusability, modularity, and testability.
- The injector creates and manages service instances.
- **Example:**

```
1  constructor(private dataService: DataService) {}
```

Dependency Injection using inject Method

- Allows injecting dependencies directly inside the class without constructors.
- Simplifies standalone components and services.
- **Example:**

```
1  import { inject } from '@angular/core';
2  import { HttpClient } from '@angular/common/http';
3
4  export class DataService {
5    private http = inject(HttpClient);
6
7    fetchData() {
8      return this.http.get('https://api.example.com/data');
9    }
10 }
```

Routing

- Enables navigation between views in single-page apps.
- Configured using RouterModule and route definitions.
- Supports advanced features like lazy loading (loads modules only when needed) and route guards (protects routes based on conditions).
- **Example:**

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

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

References

1. https://angular.io/docs
2. https://developer.mozilla.org/en-US/docs/Web/HTML
3. https://developer.mozilla.org/en-US/docs/Web/CSS
4. https://react.dev
5. https://nodejs.org/en/


thank you

Angular Deep Dive :

Components, Directives, Services, Data Binding, Templates, Forms & Pipes

Overview

Components

The reusable building blocks that make up the user interface.

Directives

Add behavior to elements and manipulate the DOM.

Services

Classes for business logic and data sharing across components.

Data Binding

Synchronize data between your component's logic and the UI view.

Templates

HTML with Angular-specific syntax to define a component's view.

Forms

Handle user input, validation, and submission for interactive experiences.

Pipes

Transform display data directly within your templates.

Angular Components

- Components are the fundamental building blocks of Angular applications, representing reusable UI pieces that make up everything you see on screen.
- Each component encapsulates a specific piece of functionality and visual design, making your codebase more organized, maintainable, and scalable.
- Every Angular component consists of three essential parts that work together to create functional, styled UI elements.

Anatomy of a Component

TypeScript Class

Contains the component logic, data properties, and methods that control behavior. This is where you define variables, handle events, and implement business logic.

HTML Template

Defines the component's view structure using HTML markup. This template displays data and responds to user interactions, creating the visual representation of your component.

CSS Styles

Provides component-specific styling that's encapsulated by default. These styles only affect the current component, preventing unintended side effects elsewhere in your application.

Creating a Component with Angular CLI

- **Basic CLI Command :**

  cmd: ng generate component profile

- This command creates a new component named "profile" with all necessary files in the proper folder structure.
- **What Gets Generated**
  - ✓ TypeScript component class file
  - ✓ HTML template file
  - ✓ CSS stylesheet file
  - ✓ Spec file for unit testing
  - ✓ Automatic registration in the module

Component Lifecycle Hooks

- Lifecycle hooks are special methods that let you tap into key moments in this lifecycle to run custom code at the right time.

1

ngOnInit

Runs once after component initialization. Perfect for fetching data, setting up subscriptions, or initializing properties. This is the most commonly used lifecycle hook.

2

ngOnChanges

Fires when input properties change. Use this to respond to data updates from parent components and perform related calculations or side effects.

3

ngOnDestroy

Called right before Angular destroys the component. Essential for cleanup tasks like unsubscribing from observables and detaching event handlers to prevent memory leaks.

Passing Data into Components

Input Binding with @Input()

- The **@Input()** decorator enables parent components to pass data down to child components, creating a unidirectional data flow. This pattern is essential for building reusable components

01

Declare Input Property

In the child component, mark a property with `@Input()` to indicate it accepts data from outside.

02

Pass Data from Parent

The parent component binds a value to the child's input property using property binding syntax.

03

Use Data in Child

The child component can now access and display the received data in its template or logic.

Child Component (ProductCard)

```
@Component({
  selector: 'app-product-card'
})
export class ProductCard {
  @Input() product: Product;
}
```

Parent Template (ProductList)

```
<app-product-card
  [product]="currentProduct">
</app-product-card>
```

Emitting Events from Components

Output Binding with @Output()

- **@Output()** enables children to communicate back up to their parents by emitting custom events.

Define the Output

Create an `EventEmitter` property decorated with `@Output()` in the child component. This will emit events that the parent can listen for.

```
@Output() liked = new
EventEmitter<number>();
```

Emit the Event

Call the `emit()` method when something happens in the child component, such as a button click or form submission.

```
onLikeClick() {
  this.liked.emit(this.postId);
}
```

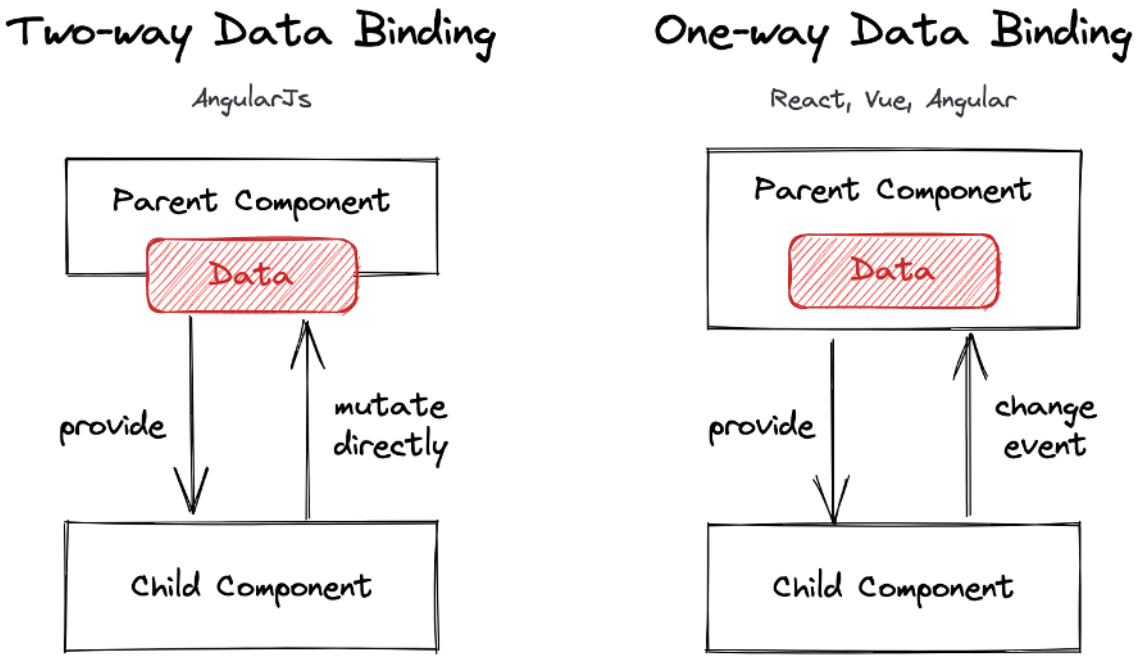
Handle in Parent

The parent component listens for the event using event binding syntax and responds with its own logic.

```
<app-like-button
  (liked)="handleLike($event)">
</app-like-button>
```

Data Binding

- Data binding is the bridge that connects your component's data with what users see on screen.
- Think of it as a communication channel between your TypeScript code and your HTML template.
- Angular offers four powerful ways to connect your component data with your template.



Interpolation

Display component data in your template using {{ }} syntax

Property Binding

Set HTML element properties dynamically from component data

Event Binding

Respond to user actions like clicks, key presses, and form submissions

Two-Way Binding

Sync data between component and view in both directions simultaneously

Interpolation: {{ }}

- Interpolation is the simplest form of data binding. You wrap a component property in double curly braces {{ }}

```
// Component
username = 'Ajay';

// Template
<h2>Welcome, {{username}}!</h2>
```

- One-way data binding: Data flows from component to view only.
- Interpolation works with any component property: numbers, dates, calculated values, or even the results of simple expressions.

Property Binding: Dynamic Control

Controlling HTML Properties from Your Component

- Property binding lets you dynamically set HTML element properties based on your component's data.
- It can control any property of an HTML element (e.g., disabled, value, src).
- **Syntax :** [propertyName]="componentValue"

```
// Component
isFormValid = false;

// Template
<button [disabled]="!isFormValid" >
  Submit
</button>
```

- One-way data binding: data flows from the component to the DOM.

Event Binding: Capturing User Actions

- Event binding is how your Angular application listens and responds to user interactions. Whether it's a click, key press, mouse movements, form submission, and more.
- event binding connects user actions to your component's methods.
- Use parentheses () around the event name to bind it to a method in your component.
- **Syntax :** (eventName)="methodName()"
- Event binding is one-way: from the view (template) to the component.
- Supports common DOM events like (click), (keyup), (input), (mouseover), etc.

```
1   // Component
2   showDetails = false;
3
4   toggleDetails() {
5     this.showDetails = !this.showDetails;
6   }
7
8   // Template
9   <button (click)="toggleDetails()" >
10    Show More
11  </button>
```

Two-Way Data Binding: [( )]

- Two-way data binding combines property binding and event binding into a single one.
- Synchronizes data between the component class and the template, allowing changes in one to instantly update the other.
- **Syntax :** [(two-way binding property)]="propertyName"

```
// component
export class AppComponent {
  userName: string = 'Ajay';
}

// template
<input [(ngModel)]="userName" />
<p>Hello, {{ userName }}!</p>
```

- Requires importing FormsModule.
- Typing in the input box updates userName and the greeting in real-time.

Angular Directives

- Directives are special markers in your templates that tell Angular how to modify or extend the behavior of DOM elements.
- Think of them as superpowers for your HTML, adding dynamic capabilities that ordinary HTML can’t provide.
- It enhance and manipulate the elements in your user interface without manually changing the DOM with JavaScript.
- Angular’s compiler recognizes these markers and applies the corresponding behavior, allowing declarative UI control directly in the template.

Structural vs. Attribute Directives

Structural Directives

Change the DOM Structure

Structural directives that change the DOM layout by adding or removing elements. They're marked with an asterisk (*) prefix.

- ***ngIf:** Conditionally add/remove elements
- ***ngFor:** Repeat elements for each item in a list
- ***ngSwitch:** Display one element from a set of alternatives

```
<div *ngFor="let product of products>
{{product.name}}
</div>
```

Perfect for product listings, showing items only when data exists, or displaying different content based on conditions.

Attribute Directives

Modify Element Behavior

Attribute directives change the appearance or behavior of existing elements without changing DOM structure.

- **ngStyle:** Dynamically apply inline styles
- **ngClass:** Conditionally apply CSS classes
- **ngModel:** Enable two-way data binding

```
<p [ngStyle]="{'color': isHovered?'blue':'black'}">
  Hover me
</p>
```

Ideal for hover effects, theme switching, form validation styling, and responsive design adjustments.

Custom Directives

- Custom directives are powerful features in Angular that allow developers to add specific, reusable behavior to HTML elements beyond what Angular's built-in directives provide.

1

Generate the Directive

Use Angular CLI to generate a directive using the command.This will create the necessary files for your directive.

```
ng generate directive highlight
```

2 Define the Directive with @Directive

Decorate the directive class with `@Directive` and provide a selector matching the attribute name. This selector is how you'll apply the directive in your HTML.

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

@Directive({
  selector: '[appHighlight]'
})
export class HighlightDirective {
  constructor() { }
}
```

3 Access the Host Element with ElementRef

Inject `ElementRef` in the constructor to access and manipulate the host element's DOM. This allows you to directly interact with the element the directive is attached to.

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

@Directive({
  selector: '[appHighlight]'
})
export class HighlightDirective {
  constructor(private el: ElementRef) { }
}
```

4 Add Event Listeners with @HostListener

Add event listeners with `@HostListener` to react to user actions (e.g., click, mouseover). This enables your directive to respond dynamically to user interaction.

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

@Directive({
  selector: '[appHighlight]'
})
export class HighlightDirective {
  constructor(private el: ElementRef) { }

  @HostListener('mouseenter') onMouseEnter() {
    this.highlight('yellow');
  }

  @HostListener('mouseleave') onMouseLeave() {
    this.highlight('');
  }

  private highlight(color: string) {
    this.el.nativeElement.style.backgroundColor = color;
  }
}
```

5 Example: Color-Changing Hover Directive

Here's a simple example where the directive changes the element's background color on hover. This demonstrates a practical application of `@HostListener` and `ElementRef`.

```
<p appHighlight> Hover over me to highlight! </p>
```

6 Import in Standalone Components

For standalone components, import the directive directly into the `imports` array of the `@Component` decorator. This makes the directive available for use within that specific standalone component.

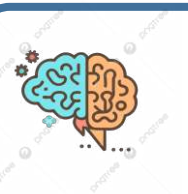
```
import { Component } from '@angular/core';
import { HighlightDirective } from './highlight.directive';

@Component({
  standalone: true,
  selector: 'app-component',
  template: `App.html`,
  imports: [HighlightDirective] // Import the directive here
})

export class MyStandaloneComponent {
  // ...
}
```

Angular Services

- Services are TypeScript classes that handle shared logic and data across your application.
- Act as the backbone of your app by handling heavy lifting such as:
  - ✓ Fetching data from APIs
  - ✓ Data storage
  - ✓ Business logic
- Services promote the single responsibility principle: components display data, services manage data.



Imagine building a social media app. You need to access user information—name, profile picture, preferences—across multiple pages:

Creating a Service

- **Basic CLI Command :**

  cmd : ng generate service weather

- This cmd generates a **weather.service.ts** file with the basic structure

```
1  import { Injectable } from '@angular/core';
2  import { HttpClient } from '@angular/common/http';
3
4  @Injectable({
5    providedIn: 'root'
6  })
7  export class WeatherService {
8
9    constructor(private http: HttpClient) { }
10
11   getTemperature(city: string) {
12     return this.http.get(`api/weather/${city}`);
13   }
14 }
```

Any component can inject this service and call getTemperature('New York') to display current weather data.

Dependency Injection

- Dependency Injection (DI) is Angular's way of providing services to components automatically.
- Instead of manually creating service instances, you simply declare what you need in the component's constructor, and Angular delivers it.
- This pattern promotes loose coupling and makes your code more flexible and testable.

01

Declare the Service

Create your service class with the `@Injectable` decorator

02

Register with Angular

Use `providedIn: 'root'` to make it available app-wide

03

Inject Where Needed

Add the service to component constructors to use it

Angular Forms

Template-Driven Forms

Similar to traditional HTML forms, these use directives like `ngModel` in your templates. They're quick to set up and perfect for simple forms like contact pages or login screens.

- Easy for beginners
- Less code in TypeScript
- Great for simple validation

Reactive Forms

Built programmatically in your TypeScript code using FormControl and FormGroup. These offer more control and are ideal for complex forms with dynamic fields or custom validation logic.

- More powerful and flexible
- Better for complex scenarios
- Easier to test

Template-Driven Forms

- Template-driven forms use ngModel to create two-way data binding between form fields and component properties.
- Angular automatically tracks form state such as valid, dirty, touched to help manage validation and UI feedback
- Supports form validation with native HTML5 attributes (e.g., required, minlength)
- Minimal setup, relies mainly on template directives
- Good for simple forms and basic validation needs

```
<form #contactForm="ngForm">
    <input type="text" name="name" [(ngModel)]="userName" required #nameField="ngModel" />

    <div *ngIf="nameField.invalid && nameField.touched">
        Name is required
    </div>

    <input type="email" name="email" [(ngModel)]="userEmail" required email />

    <button [disabled]="contactForm.invalid">
        Submit
    </button>
</form>
```

Reactive Forms

- Reactive forms give you fine-grained control over form behavior through code. You build the form structure using FormControl and FormGroup classes, making it easy to add dynamic fields, custom validators, and complex validation logic.
- Easier to add and remove form controls dynamically at runtime.
- Supports synchronous, predictable validation and form state changes.
- Facilitates unit testing by isolating form logic from the template.
- Suitable for complex forms requiring dynamic structure or advanced validation.

Template

```
<form [formGroup]="registrationForm">
    <input formControlName="email" />

    <div *ngIf="registrationForm.get('email')?.invalid && registrationForm.get('email')?.touched">
        <span *ngIf="registrationForm.get('email')?.errors?.['required']">
            Email is required
        </span>
        <span *ngIf="registrationForm.get('email')?.errors?.['email']">
            Invalid email format
        </span>
    </div>
</form>
```

TypeScript

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

registrationForm = new FormGroup({
  name: new FormControl('', Validators.required),
  email: new FormControl('', [Validators.required, Validators.email]),
  age: new FormControl('', [Validators.required, Validators.min(18)])
});
```

Angular Pipes

- Pipes are special tools in Angular templates that transform data before it is displayed. They let you change the format or appearance of data declaratively in the template using the pipe operator "|".
- **Syntax :** `{{ value | pipeName:argument1:argument2 }}`
- **Built-in Pipes:**
  - **DatePipe:** Formats dates
  - **UpperCasePipe:** Converts text to uppercase
  - **LowerCasePipe:** Converts text to lowercase
  - **CurrencyPipe:** Formats numbers as currency
  - **PercentPipe:** Formats numbers as percentage
  - **DecimalPipe:** Formats decimal numbers

Example :

```
1  {{ today | date:'fullDate' }}
2  {{ 'angular pipes' | uppercase }}
```

Custom Pipes

- Custom pipes allow you to create your own reusable data transformation functions beyond Angular's built-in pipes.
- **cmd:** ng generate pipe capitalize
- Implement the transform() method with logic to modify the input value and return the transformed output.

TypeScript:

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

@Pipe({ name: 'capitalize' })
export class CapitalizePipe implements PipeTransform {
  transform(value: string): string {
    if (!value) return value;
    return value.charAt(0).toUpperCase() + value.slice(1);
  }
}
```

Template:

```html
<p>{{ 'hello world' | capitalize }}</p>
```

Final Recap

Everything You've Learned

Components

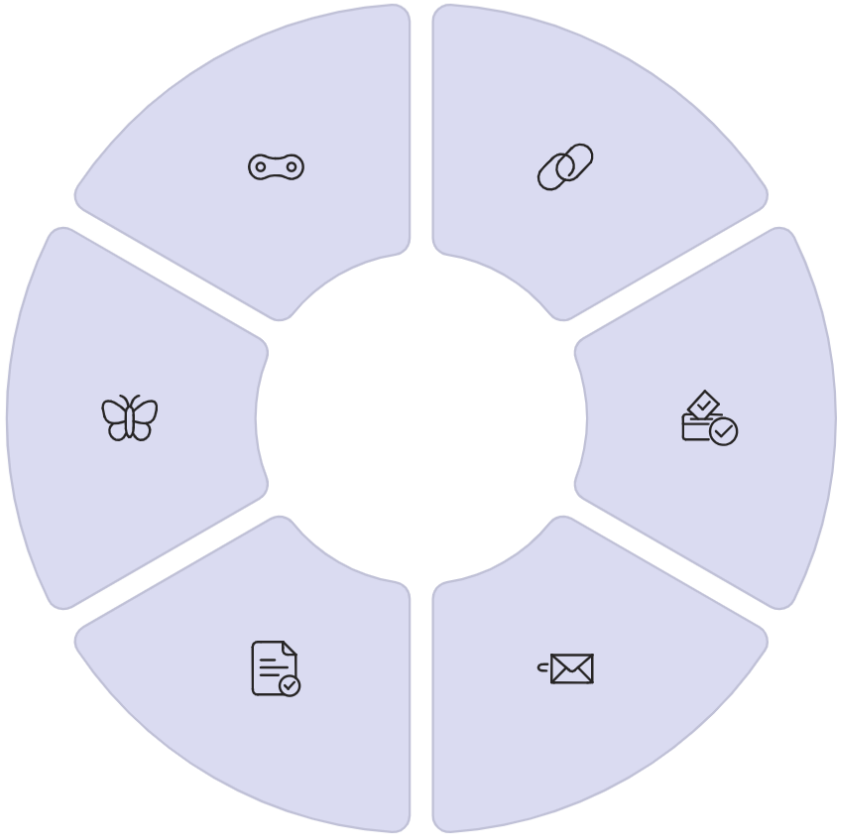
Building blocks of your UI with templates and logic

Pipes

Transforming and formatting data in templates

Forms

Capturing and validating user input effectively



Data Binding

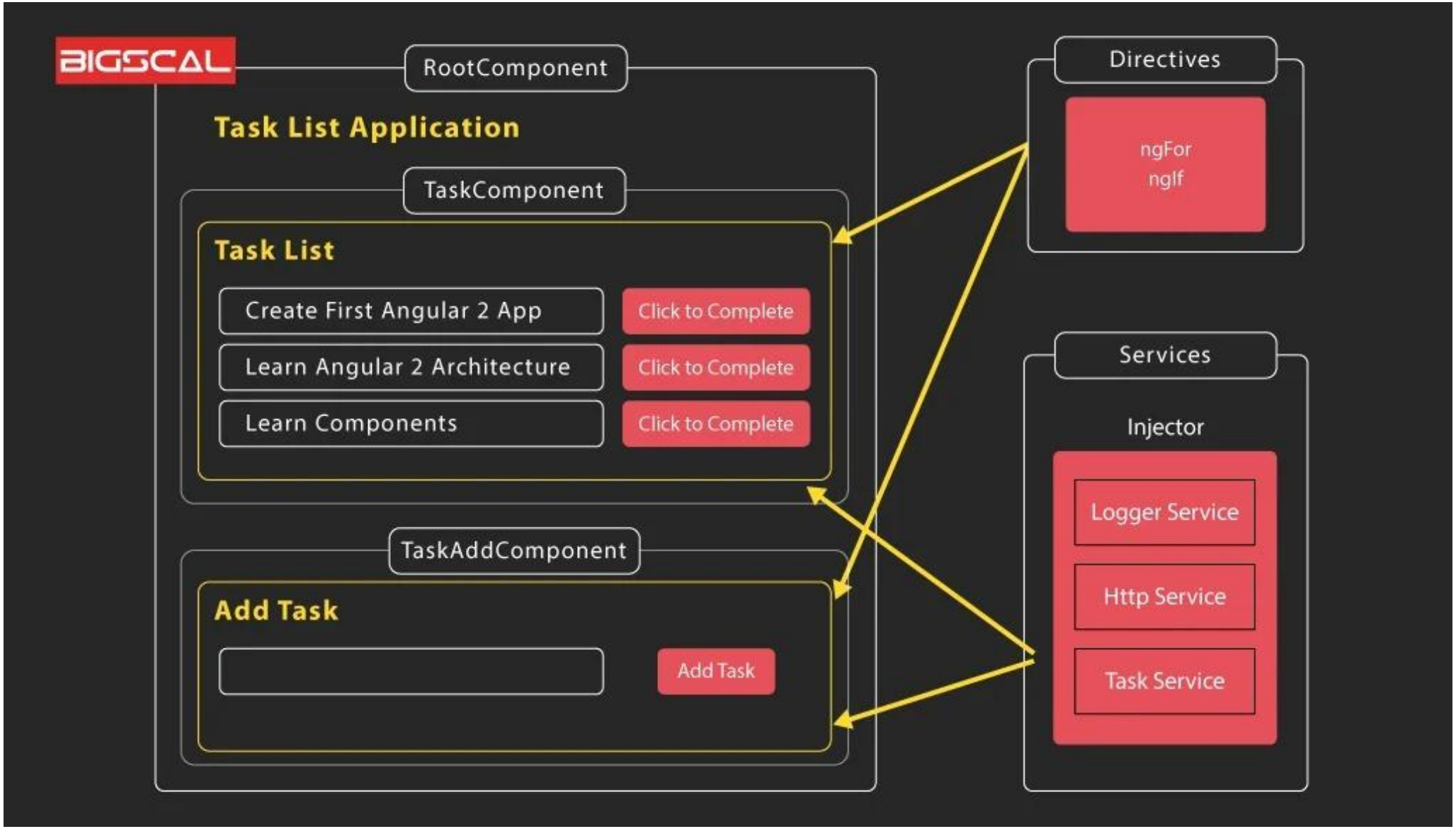
Connecting component data to your templates dynamically

Directives

Manipulating DOM and adding behavior to elements

Services & DI

Sharing logic and data across your application


BIGSCAL
RootComponent
Task List Application
TaskComponent
Task List
Create First Angular 2 App
Click to Complete
Learn Angular 2 Architecture
Click to Complete
Learn Components
Click to Complete
TaskAddComponent
Add Task
Add Task
Directives
ngFor
ngIf
Services
Injector
Logger Service
Http Service
Task Service


Thank You
CODETANTRA