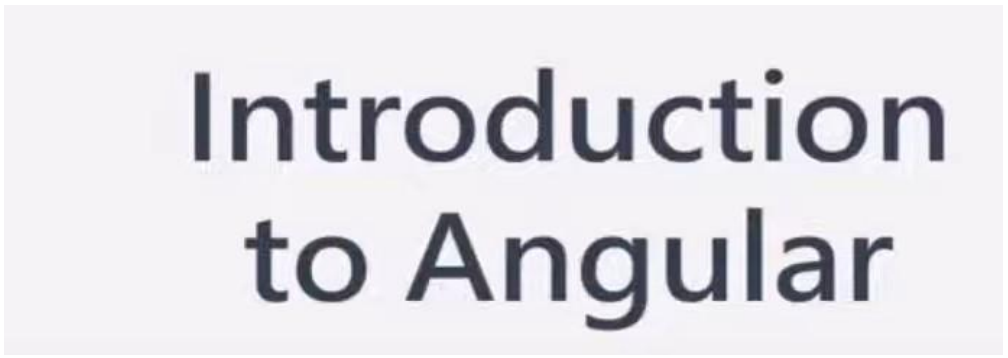
ANGULAR 16 NOTES 2024



Angular Course Overview

This Angular course covers everything about Angular step-by-step, from basic concepts to more advanced topics. The first lecture provides an introduction to Angular, including an overview of Angular as a framework, single page applications (SPA), and the different versions of Angular.

Angular is one of the most popular JavaScript framework for building client side applications.

1. What is Angular?
2. What is a single page application?
3. Why you should use angular?
4. Different versions of angular?

What is Angular?

Angular is a **JavaScript framework** used to build **single page applications (SPAs)** for both mobile and desktop. It uses **HTML**, **CSS**, and programming languages like **TypeScript** or **JavaScript** to create dynamic client-side applications.

- **Angular is a framework, not a programming language**.
- It provides **predefined methods, classes, and interfaces** to simplify development, reducing the need to write code from scratch.

1. Angular is a development platform for building a single-page application for mobile and desktop.
2. It is used for building client-side applications using HTML, CSS and a programming language like JavaScript or TypeScript.
3. Angular is not a programming language in itself like JavaScript.

Angular is a JavaScript Framework which allows us to create Single Page Applications (SPA).

What is a Framework

Framework vs. Single Page Application (SPA)

What is a Framework?

A framework is a platform that provides predefined classes and functions which can be reused in an application, making it easier to develop software by avoiding the need to write everything from scratch. Angular provides tested and verified classes and methods for tasks like **HTTP requests** to a server.

Framework

A framework is a collection of pre-defined classes and methods which provides APIs for performing different operations when used in an application

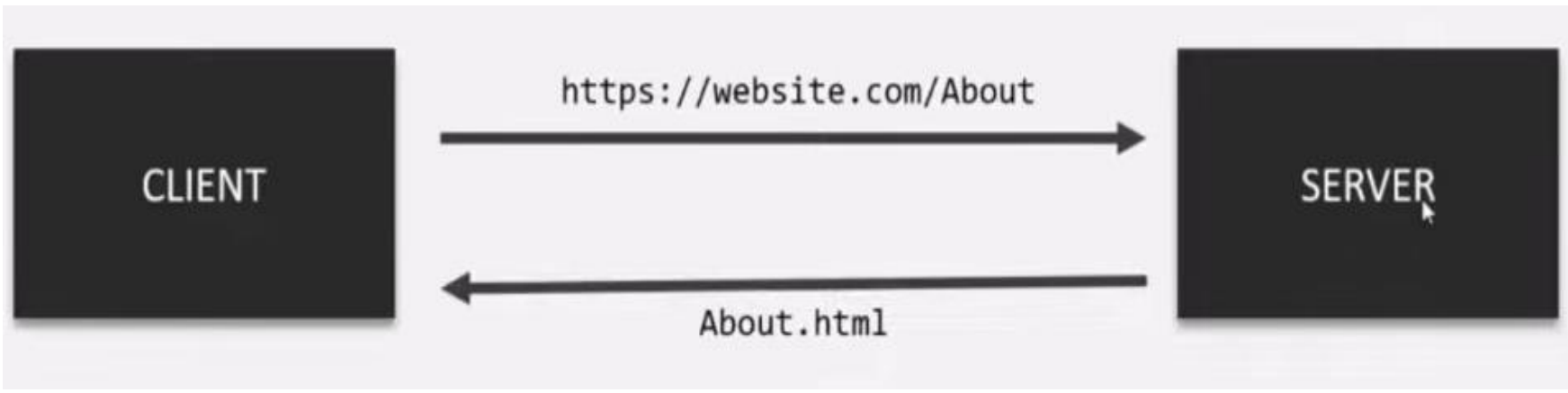
Single Page Application

Single Page Application (SPA)

An SPA is an application with a single HTML page where the content dynamically updates without reloading the entire page.

Example: **Gmail** and **Netflix** are SPAs.

- The URL changes, but only the content within the page changes, making it appear as if new pages are being loaded.
- The page content is updated **dynamically using JavaScript** without sending repeated requests to the server.



Advantages of Single Page Applications (SPA)

1. **Faster Loading**: Since JavaScript changes the page content dynamically, the application does not need to reload the entire HTML document. This improves the speed of the application.
2. **Smooth Navigation**: SPAs provide a fast, app-like experience where data can be loaded in the background, similar to a mobile application.
3. **Improved User Experience**: The lack of page refreshes and smooth transitions make the application feel more responsive and interactive.

Advantage of SPA

Since we are using JavaScript to change the content of the page, it is much faster. Here we are not reaching out to the server to request a new piece of HTML data, every time we navigate to a different URL.

This allows us to create an application which is fast and reactive.

Why Choose Angular?

Why Use Angular?

While you can build applications with **plain JavaScript or jQuery**, as the application grows in complexity, the code becomes harder to maintain. Angular solves this problem by providing:

1. **Loose Coupling**: Angular helps in structuring the application in a way that is easy to maintain.
2. **Reusable Utility Code**: Predefined classes and methods can handle tasks like user navigation and browser history.
3. **Testability**: Applications built with Angular are easier to test using automated tools.

Limitations of JavaScript / jQuery

1. Vanilla JavaScript or jQuery code becomes hard to maintain and we will need a way to properly structure our application.
2. A lot of applications built using vanilla JavaScript / jQuery is hard to test.
3. There are some functionalities which we will have to write from scratch when using JavaScript / jQuery

Advantages of Using Angular

Advantage of using Angular

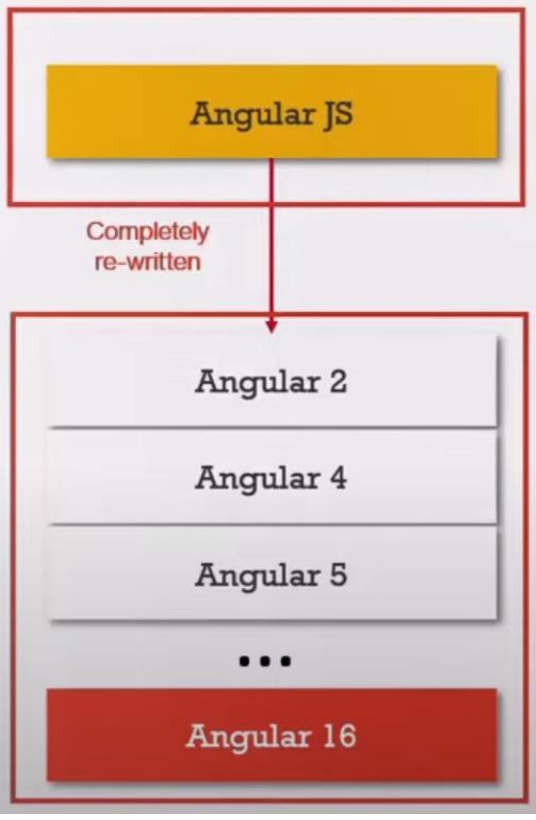
1. Angular gives our application a clean and loosely coupled structure that is easy to understand & maintain.
2. It brings a lot of utility code which can be re-used in lot of applications. Especially, when dealing with user navigation & browser history.
3. Applications built with Angular are more testable.

Angular Versions

- **AngularJS**: The original version, released in 2010. It was a popular JavaScript framework but was complex and difficult to maintain as applications grew.
- **Angular 2 and Beyond**: Released in 2016, Angular 2 was a complete rewrite of AngularJS using TypeScript, making it easier to build modern applications. Angular 2 and later versions are simply referred to as **Angular**.

  Since Angular 2, new versions like Angular 4, 5, 6, etc., are released every 6 months with minor improvements and new features. The latest version, as of this course, is **Angular 16**.

Versions of Angular



1. Angular JS was not designed with the need of todays applications.
2. Angular 2 was completely re-written from ground up.
3. It fixed all the issues which Angular 1 had.
4. There had not been major changes after the release of Angular 2.

Summary of Key Angular Concepts

- **Angular is a JavaScript framework** for creating SPAs.
- SPAs are fast and reactive, as the page content is changed dynamically using JavaScript, without the need to reload the entire page.
- Angular provides **predefined methods and classes** to make development faster and easier.
- **Angular 2 and later versions** are based on **TypeScript** and are a complete rewrite of the original AngularJS.

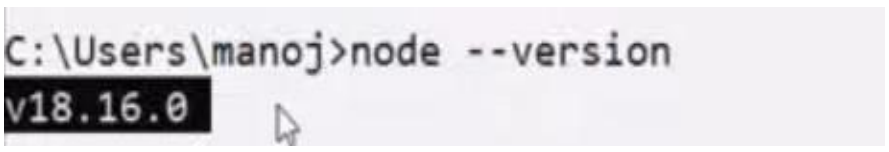
Creating a new Angular project

Notes: Setting Up an Angular Project

1. Install NODE JS on development machine.
2. Install angular CLI globally.
3. Create an Angular project.
4. Compile & run Angular project.

1. **Overview:**
   - This guide explains how to set up the development environment for creating and running an Angular project.
   - Steps involved:
     1. Install Node.js
     2. Install Angular CLI globally
     3. Create a new Angular project
     4. Compile and run the Angular project

2. **Step 1: Installing Node.js**:
    - Node.js is required for executing JavaScript outside the browser and for accessing npm (Node Package Manager).
    - Go to the official website nodejs.org and download either the latest version or the long-term support (LTS) version.
    - After downloading, follow the installation wizard, accept the license agreement, and keep default settings.
    - After installation, verify by opening the command prompt (CMD) and typing `node --version`. If successful, the installed Node.js version will display.



3. **Step 2: Installing Angular CLI**:
    - Angular CLI is a command-line tool used for creating Angular projects and generating boilerplate code.
    - To install Angular CLI globally, open a command prompt or terminal and run:

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

    - Mac users may need to prefix the command with `sudo` for elevated permissions.

After installation, verify it by typing `ng version` to see Angular and Node.js versions.

```
C:\Users\manoj>npm install -global @angular/cli@latest
npm WARN deprecated @npmcli/move-file@2.0.1: This functionality has

added 256 packages in 37s

37 packages are looking for funding
  run `npm fund` for details

C:\Users\manoj>ng version
```

Create Angular Project

Create Angular Project

To create a new angular project, move to the folder where you wand to create the project using command prompt / terminal and type following command.

```
ng new project-name
```

4. **Step 3: Creating a New Angular Project**:
    - Create a folder for the project (e.g., `AngularCompleteCourse`).
    - Navigate to the desired folder in CMD or terminal and use the following command to create a new Angular project:

```arduino
ng new project-name
```

    - Example: `ng new angular-ecart`

- You'll be prompted with a few questions:
    - Enable Angular routing? (Select Yes or No based on your project needs)
    - Choose the stylesheet format (CSS, SCSS, SASS, etc.). For simplicity, select CSS.
- Angular CLI will now create the project and download necessary packages from npm.

```
C:\Users\manoj>cd C:\Users\manoj\OneDrive\Documents\angular-complete-course

C:\Users\manoj\OneDrive\Documents\angular-complete-course>ng new angular-ekart
? Would you like to add Angular routing? No
? Which stylesheet format would you like to use?
> CSS
  SCSS   [ https://sass-lang.com/documentation/syntax#scss                ]
  Sass   [ https://sass-lang.com/documentation/syntax#the-indented-syntax ]
  Less   [ http://lesscss.org                                             ]
```

```
/ Installing packages (npm)...
```

Compile & Run

To run an Angular project, move to the project folder using command prompt / terminal and type following command.

```
ng serve
```

5. **Step 4: Running the Angular Project**:
    - Navigate into the project folder (`cd angular-ecart`).
    - Use the command `ng serve` to start the project:
        - This command compiles the project, creates JavaScript and CSS bundles, and starts a live development server.
        - The project will run at `localhost:4200` .
    - Open a browser, navigate to `http://localhost:4200` , and you should see your Angular project running.
6. **Next Steps**:
    - You'll need a code editor to modify the Angular code. A popular choice is Visual Studio Code, which will be covered in the next lecture.

This concludes the setup and initial run of an Angular project.

Editing the first Angular project

Angular Project Setup and Basics

1. **Project Structure & Setup**
   - After creating an Angular project, it contains essential files, folders, and dependencies. The project is accessible on `localhost:4200` using the command `ng serve`.
   - Example:

```bash
ng serve
```

2. **Development Server**
   - Running `ng serve` starts a development server that allows accessing the Angular app on `localhost:4200`. To stop this, press `Ctrl + C`. Without the server running, the page cannot be accessed.

3. **Restarting Server**
   - To restart the server, navigate to the project folder and rerun `ng serve`.
   - Example:

```bash
cd <project-folder>
ng serve
```

4. **VS Code Setup**
   - VS Code is recommended for editing Angular code. Download it from code.visualstudio.com, and install it. Open the Angular project folder in VS Code to begin editing.

5. **HTML Rendering**
   - The UI is initially rendered from the `index.html` file, found in the `src/` folder. This file includes injected script bundles created during the build process.

6. **Scripts Injection**
    - Angular automatically injects script files into `index.html` when running `ng serve`. These files include `runtime.js`, `polyfills.js`, `styles.js`, `vendor.js`, and `main.js`.
7. **App Component**
    - The UI is rendered by a component linked to the `<app-root>` element. The `app-root` is defined in the `app.component.ts` file.
    - Example of `app-root` in `index.html`:

```html
<app-root></app-root>
```

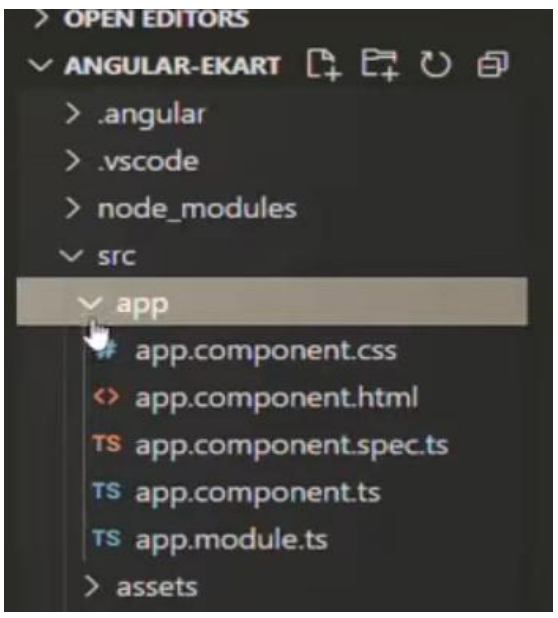
8. **Component Structure**
    - A component has four key files:
        - `.ts` (TypeScript logic),
        - `.html` (HTML template),
        - `.css` (CSS for styling),
        - `.spec.ts` (test file).
    - The main logic resides in `app.component.ts`, which has a class decorated with `@Component`.
9. **Component Template**
    - The `@Component` decorator in `app.component.ts` links the component to an HTML file through the `templateUrl`.
    - Example:

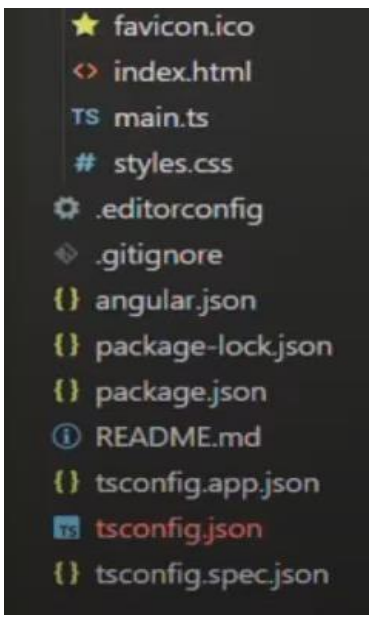
```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent { }
```

10. **Rendering Dynamic Content**

    - Angular allows dynamic content generation using its component-based architecture. Instead of static HTML, you can dynamically generate content, which is a key feature of Angular's framework.

Angular Project Files & Folder Structure





Angular Project Structure Overview

When an Angular project is created using Angular CLI, several folders and files are automatically generated to manage the structure of the project. Here's a breakdown of the important folders and files, along with their use:

1. **Node Modules Folder**

- **Purpose**: Contains all the third-party libraries that the Angular project depends on, downloaded via npm.
- **Usage**: When you run `npm install`, this folder is populated with all required libraries based on the `package.json` file.
- **Note**: This folder is not checked into the repository or deployed to production servers.

2. **Package.json File**

- **Purpose**: Contains the configuration for the Angular project and lists all the npm dependencies.
- **Key Elements**:
  - **Name and Version**: Project name and version.
  - **Scripts**: Commands such as `ng serve` to run the project.
  - **Dependencies**: Lists required libraries for the project.
  - **Dev Dependencies**: Lists libraries only needed for development.

3. **Package-lock.json File**

- **Purpose**: Records the exact versions of installed dependencies, ensuring consistency across environments (development, production, etc.).
- **Note**: It ensures that every team member has the same version of dependencies installed when running `npm install`.

4. **EditorConfig File**

- **Purpose**: Ensures coding consistency across the team by enforcing coding standards (e.g., indentation, character length).
- **Access**: Managed by team leads or managers, not developers.

5. **.gitignore File**

- **Purpose**: Specifies files and folders that should not be included in version control (e.g., `node_modules`, `temp` folders).
- **Example**: You can add a custom folder like `/temp` to avoid committing unnecessary files.

6. **Angular.json File**

- **Purpose**: The main configuration file for the Angular project.
- **Details**:
  - Contains information about styles, scripts, and the main entry points (e.g., `index.html`, `main.ts`).
  - Defines the project structure, like the application's name, project type, etc.

7. **tsconfig.json File**

- **Purpose**: Configures the TypeScript compiler, specifying how TypeScript code should be compiled to JavaScript.
- **Note**: Developers rarely need to modify this file during day-to-day development.

8. **Source Folder**

- **Purpose**: The most important folder containing all the source code of the application.
- **Subfolders**:
  - **App Folder**: Contains the main application code including components, services, and modules.
    - **App Component**: The default component generated when the project is created.
    - **App Module (app.module.ts)**: Every application must have at least one module.
  - **Assets Folder**: Contains static files like images and icons. These files are publicly accessible.

Key Concepts

- **Dependencies vs. Dev Dependencies**:
  - **Dependencies**: Libraries required for the project to run.
  - **Dev Dependencies**: Libraries required only during development (e.g., testing libraries, linters).

Commands to Remember

- **ng new**: Creates a new Angular project.
- **npm install**: Installs all the dependencies listed in `package.json`.

These files and folders form the core structure of an Angular project, ensuring smooth collaboration and consistent project setup across different environments.

Bootstrapping Angular Application

Notes: Bootstrapping an Angular Application

Definition: Bootstrapping

- The process of initializing or loading an Angular application.
- The Angular app starts by loading the `index.html` file, which serves as the entry point in the browser.

Angular CLI and Commands

- `ng serve`:
  - Builds the application, compiles the code in memory, and starts the development server.
  - Automatically recompiles if changes are made.
- `ng build`:
  - Builds the application and generates the output files (e.g., JavaScript, CSS) in a `dist` folder, including `index.html`.

`index.html` and `approot` Tag

- Initially, the `index.html` file contains the `<app-root>` tag but no direct references to JavaScript or CSS files.
- After running `ng serve` or `ng build`, Angular injects JavaScript bundles (e.g., `runtime.js`, `polyfills.js`, `main.js`) into the `index.html`.

Script Files in Angular

- `runtime.js`: Webpack runtime for module loading.
- `polyfills.js`: Ensures compatibility with older browsers.
- `main.js`: Contains compiled JavaScript code of the application.
- `vendor.js`: Includes Angular code libraries and third-party dependencies.
- `styles.js`: Bundles CSS files into JavaScript and injects them into the app.

Entry Point of an Angular Application

- `angular.json` specifies the entry point of the application.
- The `"main"` option in `angular.json` points to `main.ts`, which is the first TypeScript file executed.

`main.ts` File

- `platformBrowserDynamic()`: Loads the application in the browser.
- `AppModule`: Root module of the application is bootstrapped (loaded) in the browser.

Modules in Angular

- Angular applications are organized into **modules**.
- The root module (in this case, `AppModule`) is the main module that gets loaded first.

AppModule and AppComponent

- `AppModule`:
  - `@NgModule` decorator defines metadata:
    - `declarations`: Components, directives, pipes.
    - `imports`: External modules.
    - `providers`: Services.
    - `bootstrap`: Components to load first (in this case, `AppComponent`).

- `AppComponent`:
  - `@Component` decorator includes metadata for rendering the UI:
    - `selector`: Specifies the HTML element to use (`<app-root>`).
    - `templateUrl`: Specifies the view template file (`app.component.html`).

Rendering the UI

- The selector (e.g., `<app-root>`) is replaced by the view template (`app.component.html`) content.
- The UI is displayed in the browser once Angular knows what to render.

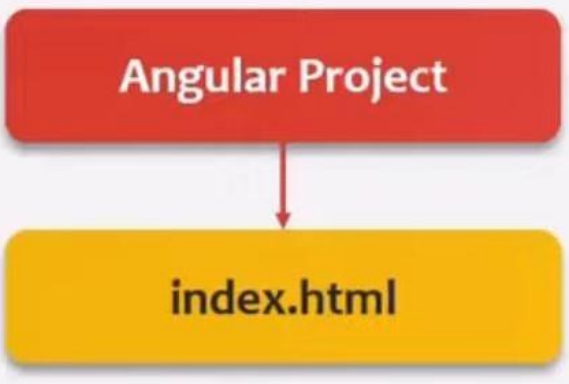
Example Code

`main.ts`

```ts
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { AppModule } from './app/app.module';

platformBrowserDynamic().bootstrapModule(AppModule)
  .catch(err => console.error(err));
```

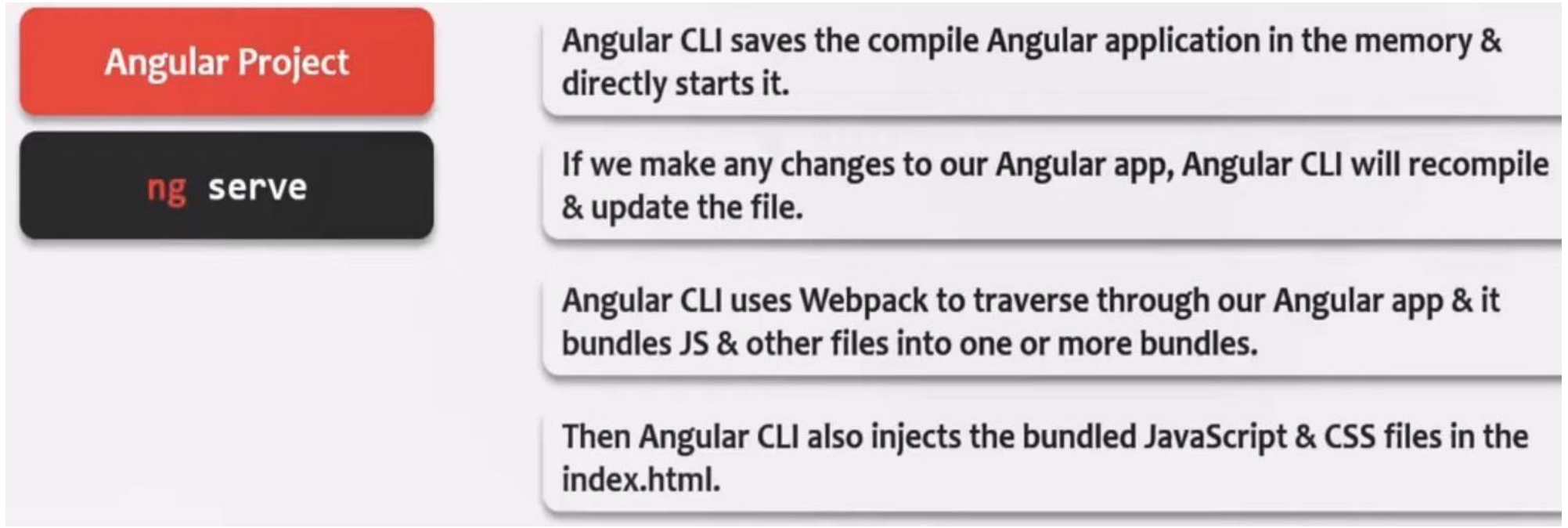
```ts
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'my-app';
}
```

Bootstrapping is the process of initializing or loading the Angular application.

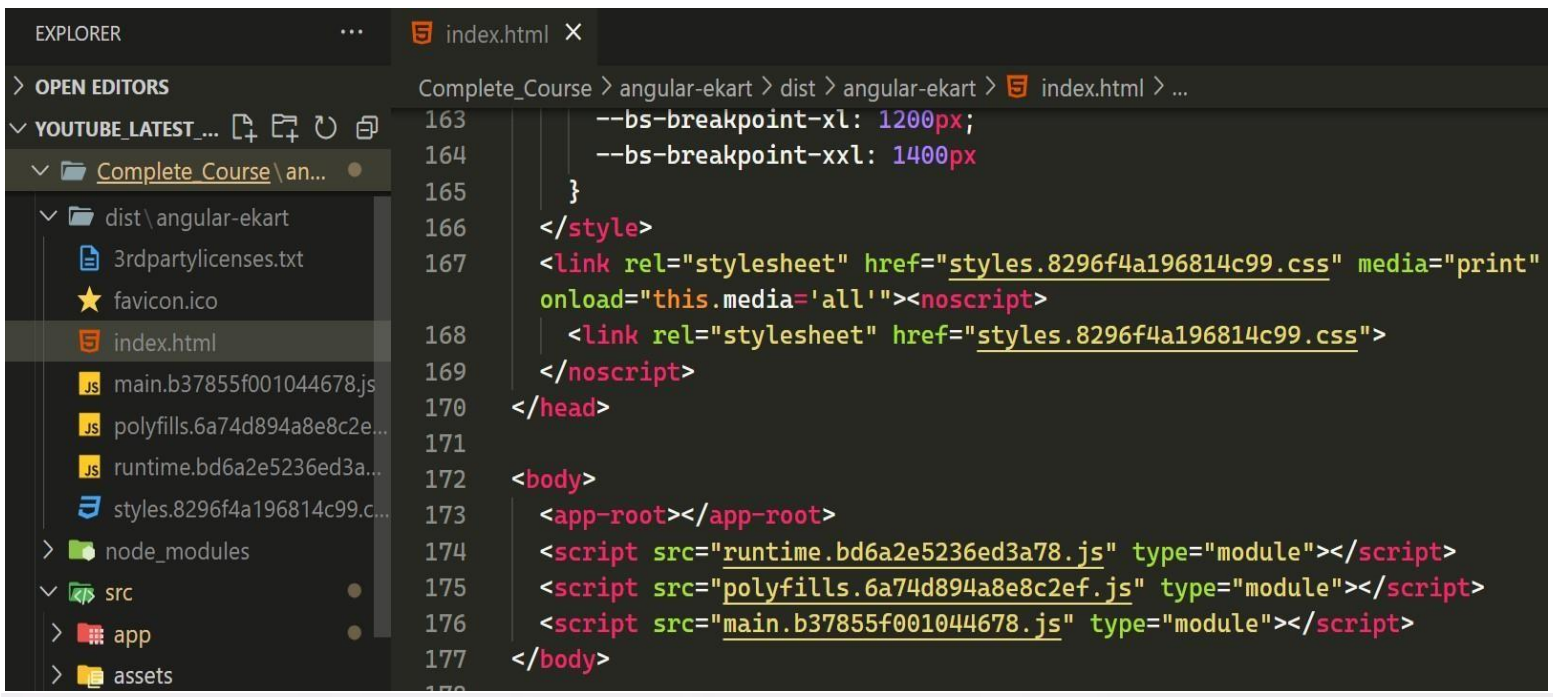


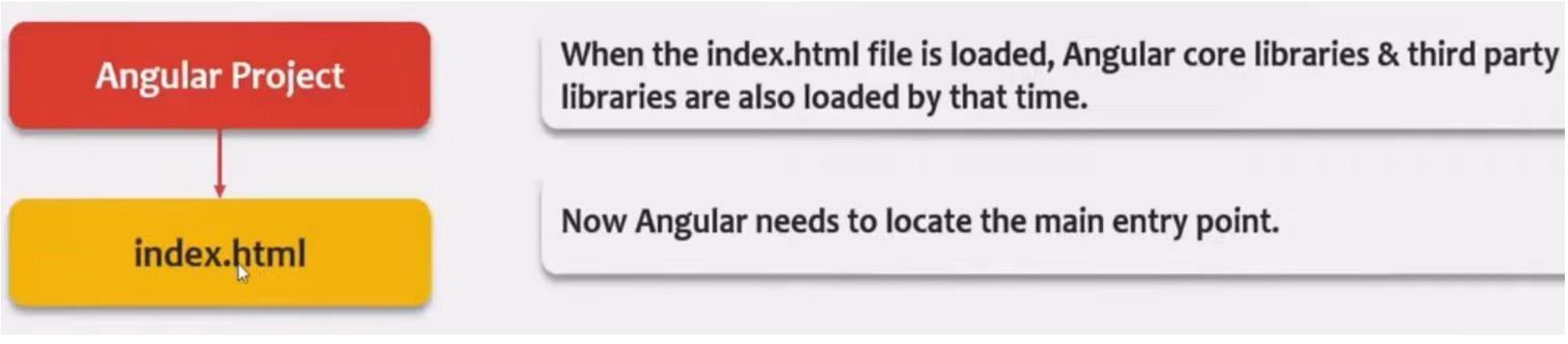
index.html

```html
<!doctype html>
<html lang="en">
<head>
  <title>AngularBasics</title>
  <link rel="icon" href="favicon.ico">
</head>
<body>
  <app-root></app-root>
</body>
</html>
```

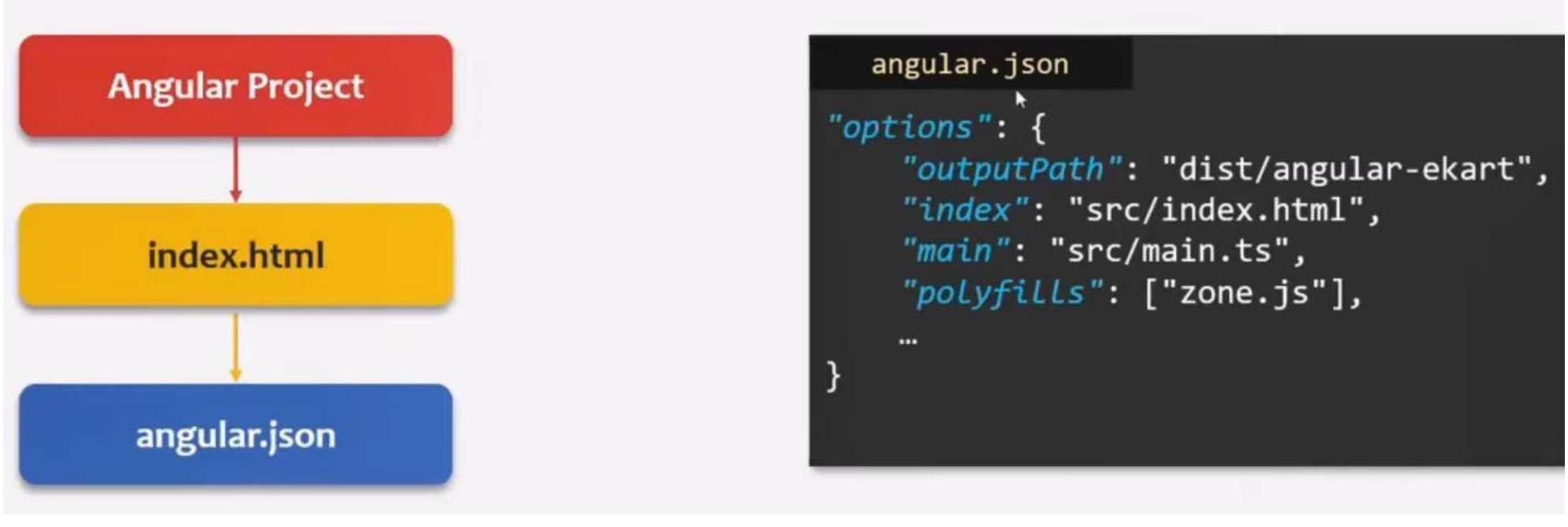



```
C:\Users\manoj\OneDrive\Documents\angular-complete-course\angular-ekart>ng build
√ Browser application bundle generation complete.
√ Copying assets complete.
√ Index html generation complete.

Initial Chunk Files           | Names         |  Raw Size | Estimated Transfer Size
main.38e3fe80cdbc685a.js      | main          |  92.45 kB |               27.75 kB
polyfills.230054cd2c60f2ca.js | polyfills     |  33.00 kB |               10.68 kB
runtime.424b4efe23da8770.js   | runtime       | 904 bytes |              510 bytes
styles.ef46db3751d8e999.css   | styles        |   0 bytes |                      -

                              | Initial Total | 126.33 kB |               38.93 kB

Build at:                       - Hash:                   - Time:      ms
```

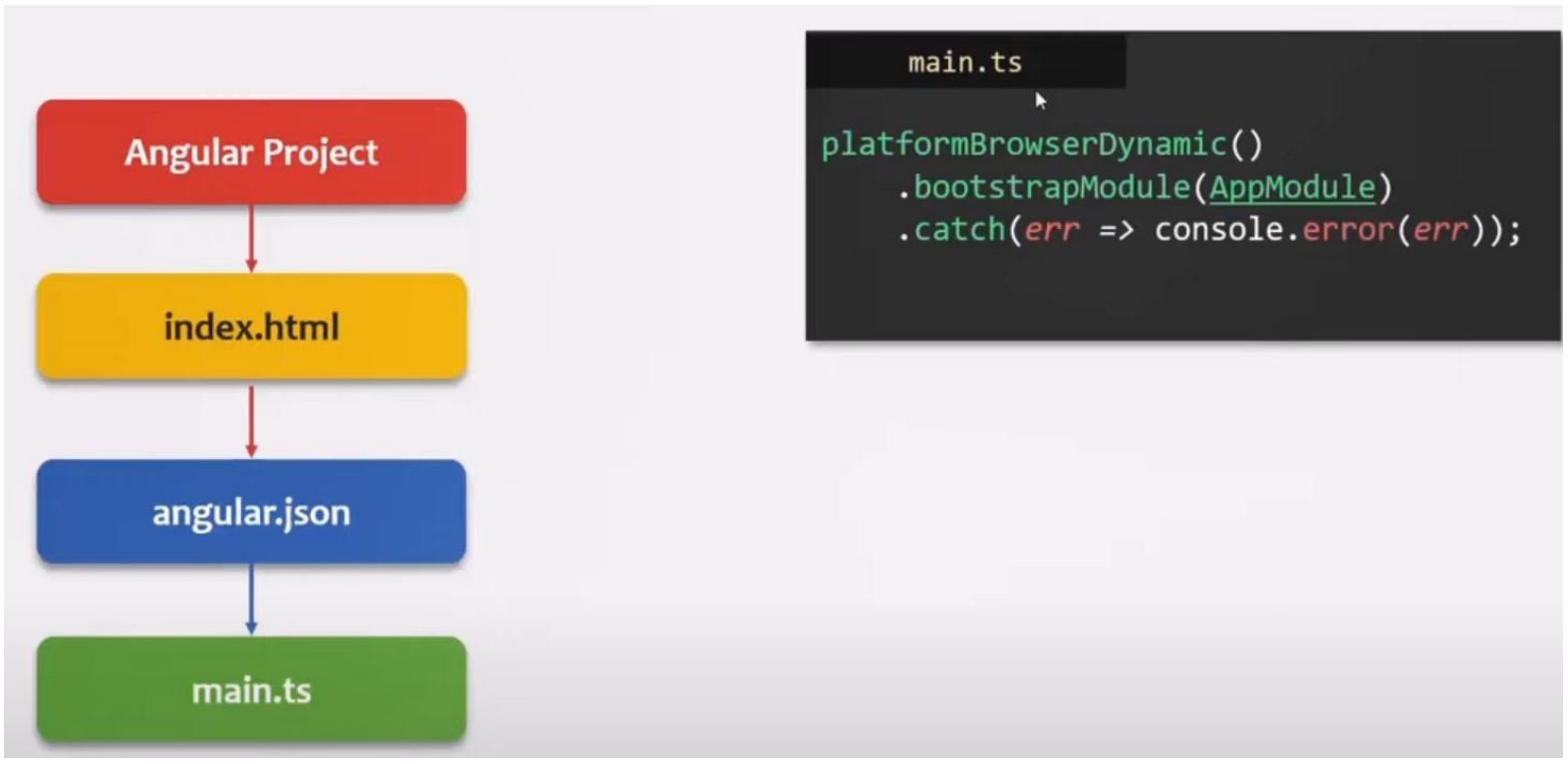


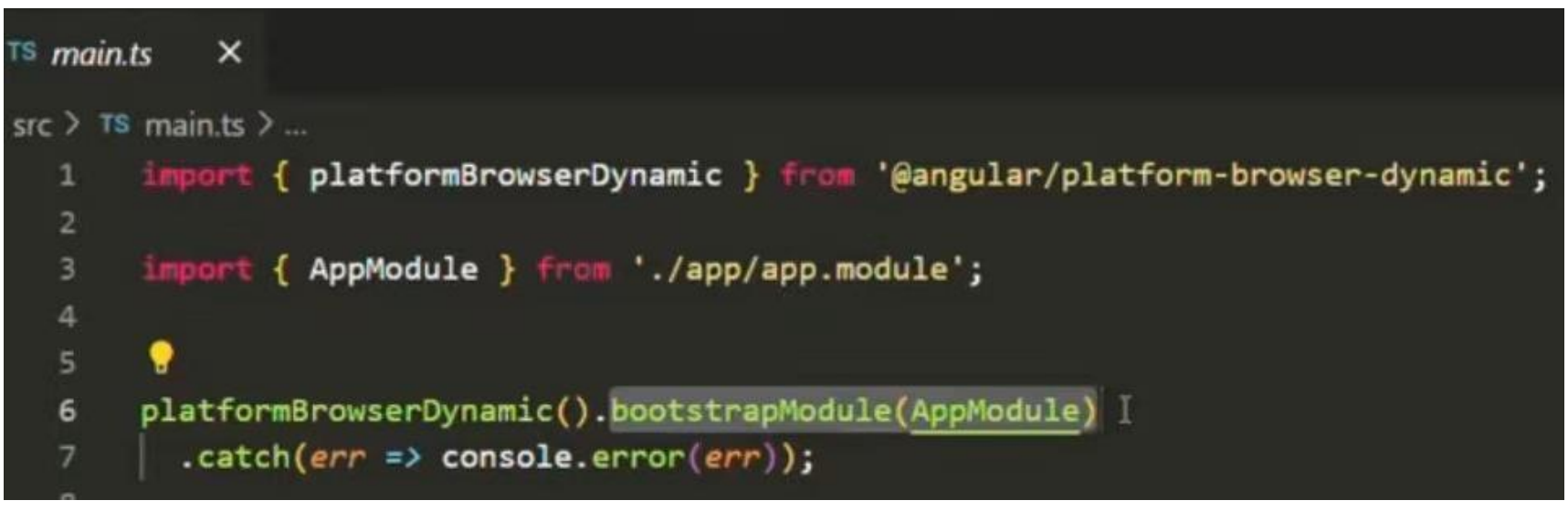


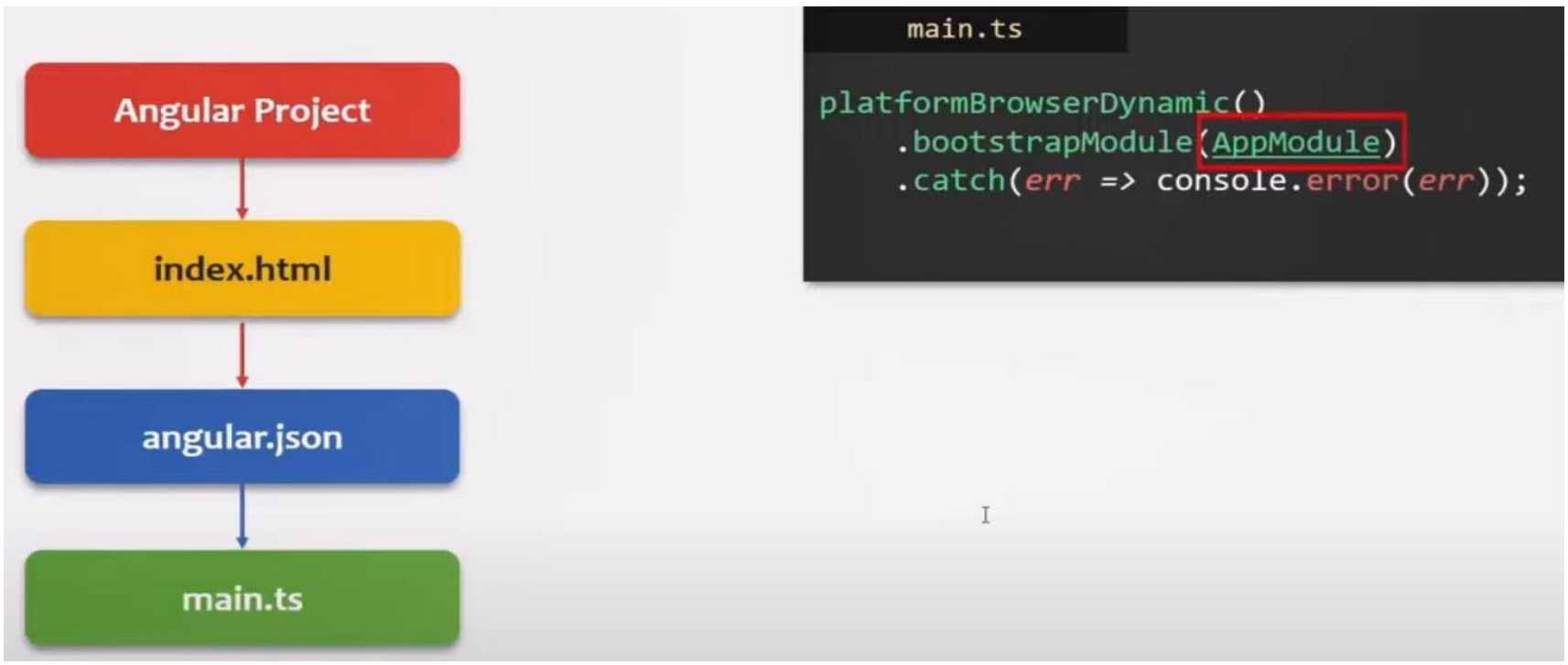
Angular searches angular.json file for main entry point file

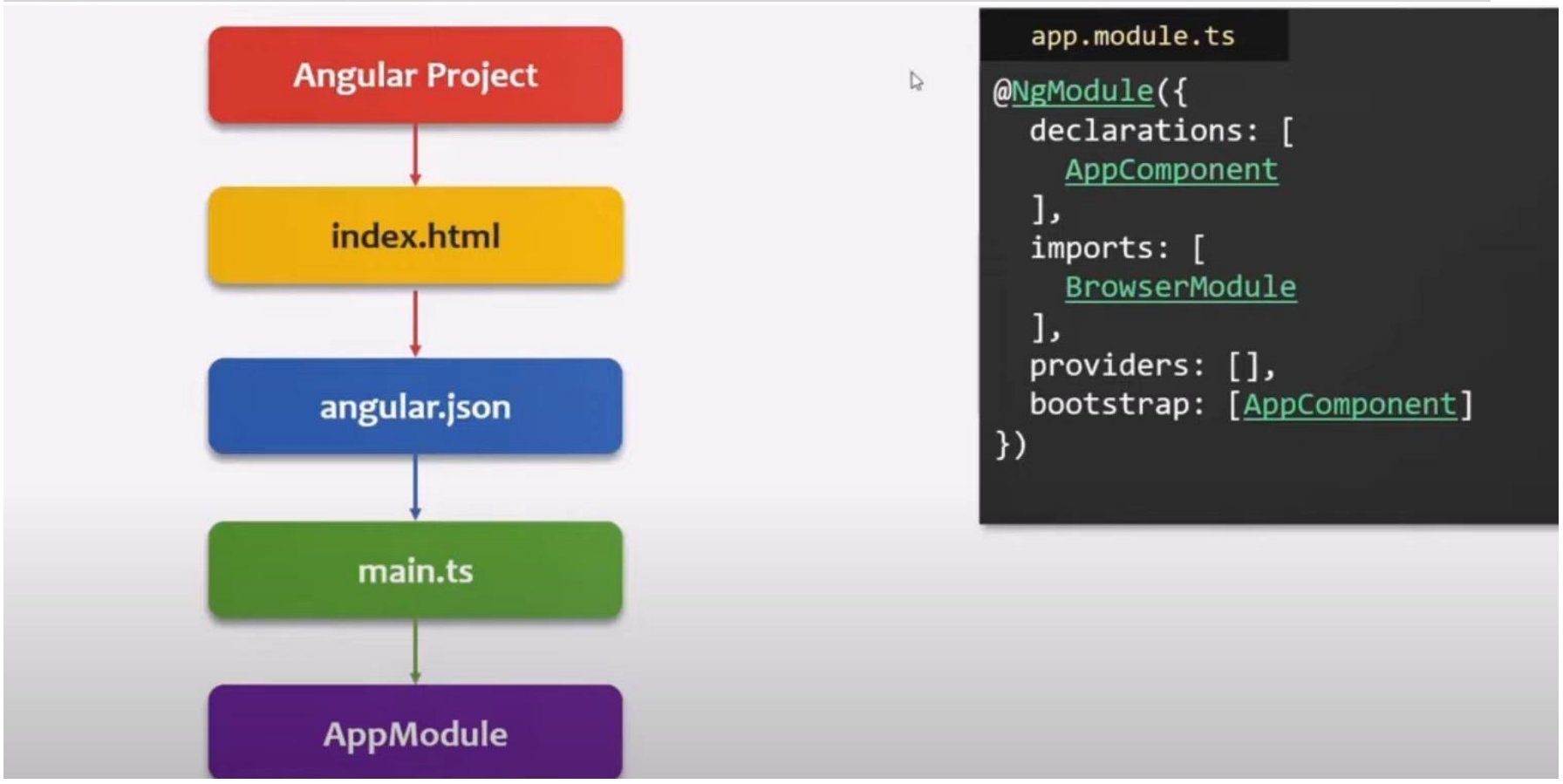


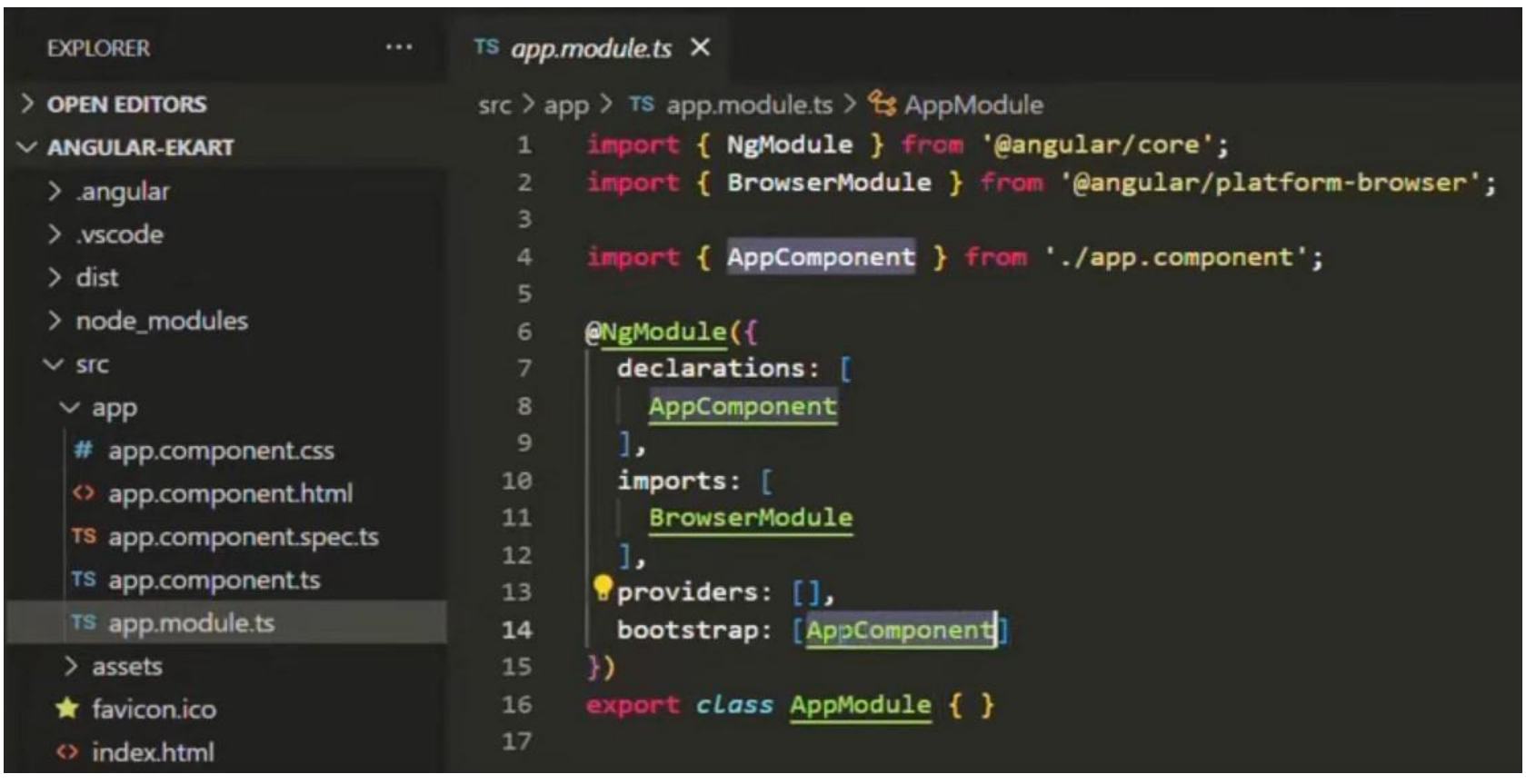
now angular comes to know where the entry point is it looks into that typescript file main . ts

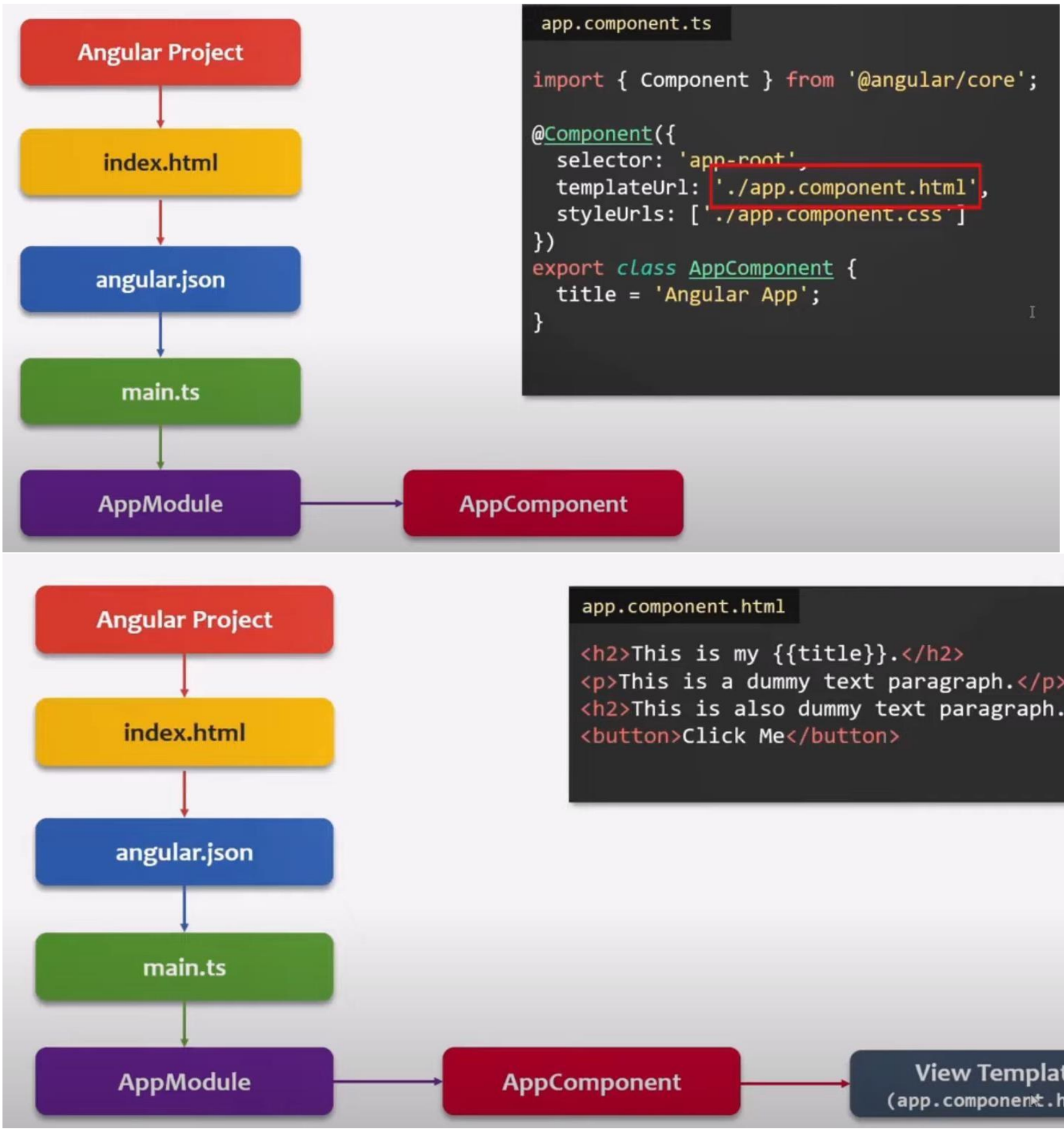




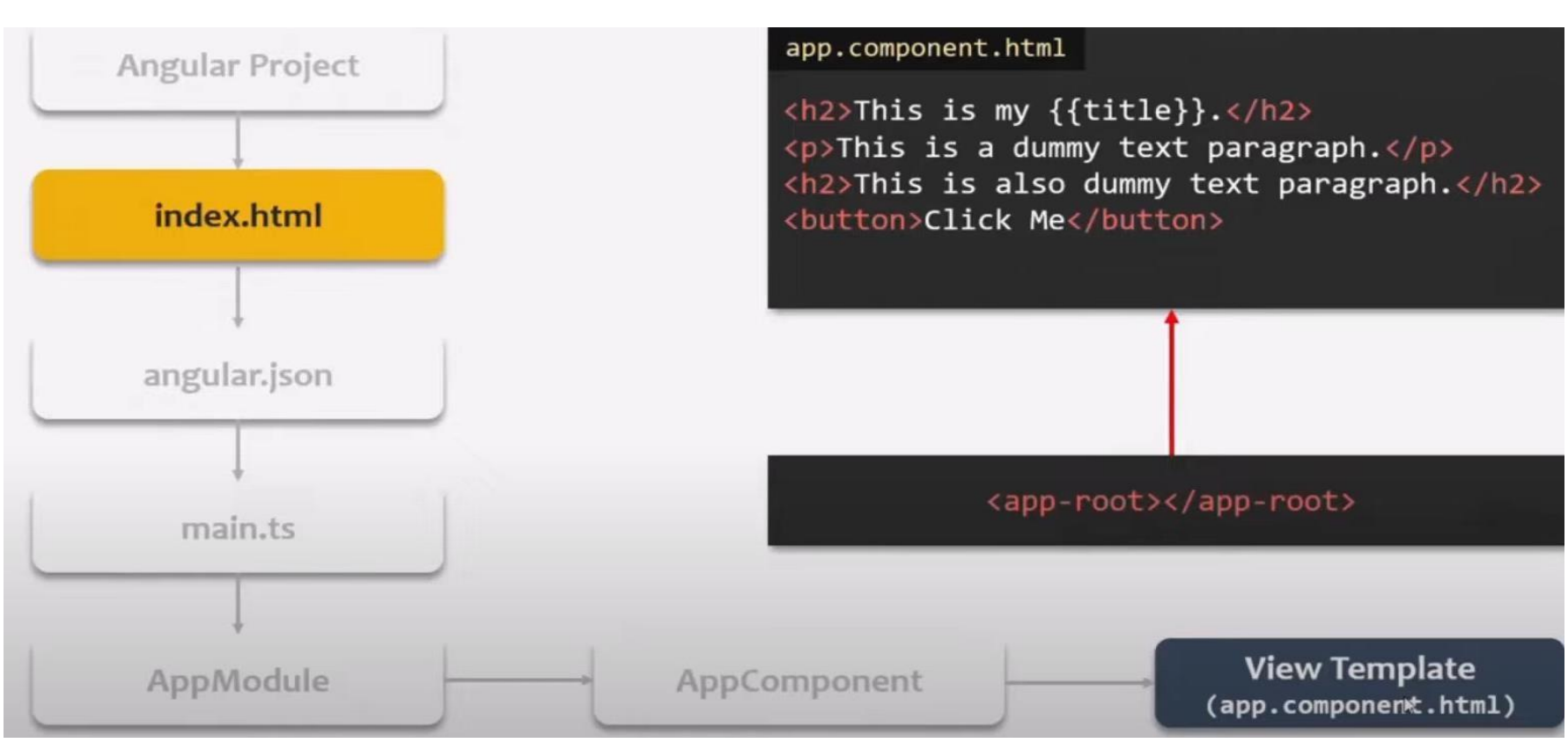

Angular Project
index.html
angular.json
main.ts
main.ts
platformBrowserDynamic()
.bootstrapModule(AppModule)
.catch(err => console.error(err));


Angular Project
index.html
angular.json
main.ts
AppModule
app.module.ts
@NgModule({
declarations: [
AppComponent
],
imports: [
BrowserModule
],
providers: [],
bootstrap: [AppComponent]
})


EXPLORER
OPEN EDITORS
ANGULAR-EKART
.angular
.vscode
dist
node_modules
src
app
app.component.css
app.component.html
app.component.spec.ts
app.component.ts
app.module.ts
assets
favicon.ico
index.html
app.module.ts
src > app > app.module.ts > AppModule
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent } from './app.component';
@NgModule({
declarations: [
AppComponent
],
imports: [
BrowserModule
],
providers: [],
bootstrap: [AppComponent]
})
export class AppModule { }


Angular Project
index.html
angular.json
main.ts
AppModule
AppComponent
app.component.ts
import { Component } from '@angular/core';
@Component({
selector: 'app-root',
templateUrl: './app.component.html',
styleUrls: ['./app.component.css']
})
export class AppComponent {
title = 'Angular App';
}

Angular Project
index.html
angular.json
main.ts
AppModule
AppComponent
View Template
(app.component.html)
app.component.html
<h2>This is my {{title}}.</h2>
This is a dummy text paragraph.
<h2>This is also dummy text paragraph.</h2>
<button>Click Me</button>


Angular Project
index.html
angular.json
main.ts
AppModule
AppComponent
View Template
(app.component.html)
app.component.html
<h2>This is my {{title}}.</h2>
This is a dummy text paragraph.
<h2>This is also dummy text paragraph.</h2>
<button>Click Me</button>
<app-root></app-root>

What is TypeScript

TypeScript is a free & open-source programming language developed by Microsoft.

Advantage of TypeScript

1. TypeScript is a superset of JavaScript.
2. TypeScript has additional features, which do not exist in current version of JavaScript Supported by most browsers.
3. TypeScript is strongly typed. But JavaScript is dynamically typed

JavaScript

```
let name = 'John';
let age = 28;
let isMarried = false;
```

TypeScript

```
let name: string = 'John';
let age: number = 28;
let isMarried: boolean = false;
```

4. TypeScript has some object-oriented features that we do not have in JavaScript yet.
5. With TypeScript, we can catch errors at compile time.

Understanding Component

In Angular, a **decorator** is a special type of function that is attached to a class, method, or property to add metadata, modify behavior, or enhance functionality. Angular uses decorators to define and configure components, services, directives, and more.

For example, the `@Component` decorator is used to define an Angular component:

Angular Components

Definition:

A **component** in Angular is a small, reusable piece of the user interface (UI). It forms the building block of Angular applications, which are created by combining multiple components. Every Angular app has at least one root component (often named `AppComponent` by convention), and this component organizes all other child components.

Key Points:

- Angular is a component-based JavaScript framework.
- Components represent small UI sections, and multiple components are combined to build complex UIs.
- Every Angular app has at least one root component (commonly named `AppComponent`).
- Components can be parent or child components, forming a hierarchical structure in the app.

Example:

Let's say you want to create a **Header Component**:

1. **Create a TypeScript Class**: A component starts with a TypeScript class. The class is exported so that it can be used in other files.

typescript Copy code

```typescript
export class HeaderComponent { }
```

2. **Add `@Component` Decorator**: To make this class a component, you decorate it with the `@Component` decorator. This decorator requires a selector and a template.

typescript

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

@Component({
  selector: 'app-header',
  template: `<h3>Ecart</h3>`
})
export class HeaderComponent { }
```

- `selector`: Defines the custom HTML tag (`<app-header></app-header>`) that Angular recognizes as this component.
- `template`: This defines the view, or UI, that the component will render. You can write inline HTML here or refer to an external file.

3. **Declare the Component in Module**: To inform Angular about this component, it needs to be declared in the module's `declarations` array.

```typescript
import { HeaderComponent } from './header/header.component';

@NgModule({
  declarations: [
    AppComponent,
    HeaderComponent
  ],
})
export class AppModule { }
```

4. **Use the Component in Another Template**: Once declared, the component can be used in other components' templates by referencing its selector.

```html
<!-- In app.component.html -->
<app-header></app-header>
```

In this case, the `HeaderComponent` with its template (`<h3>Ecart</h3>`) will render where the `<app-header></app-header>` tag is placed in the root component.

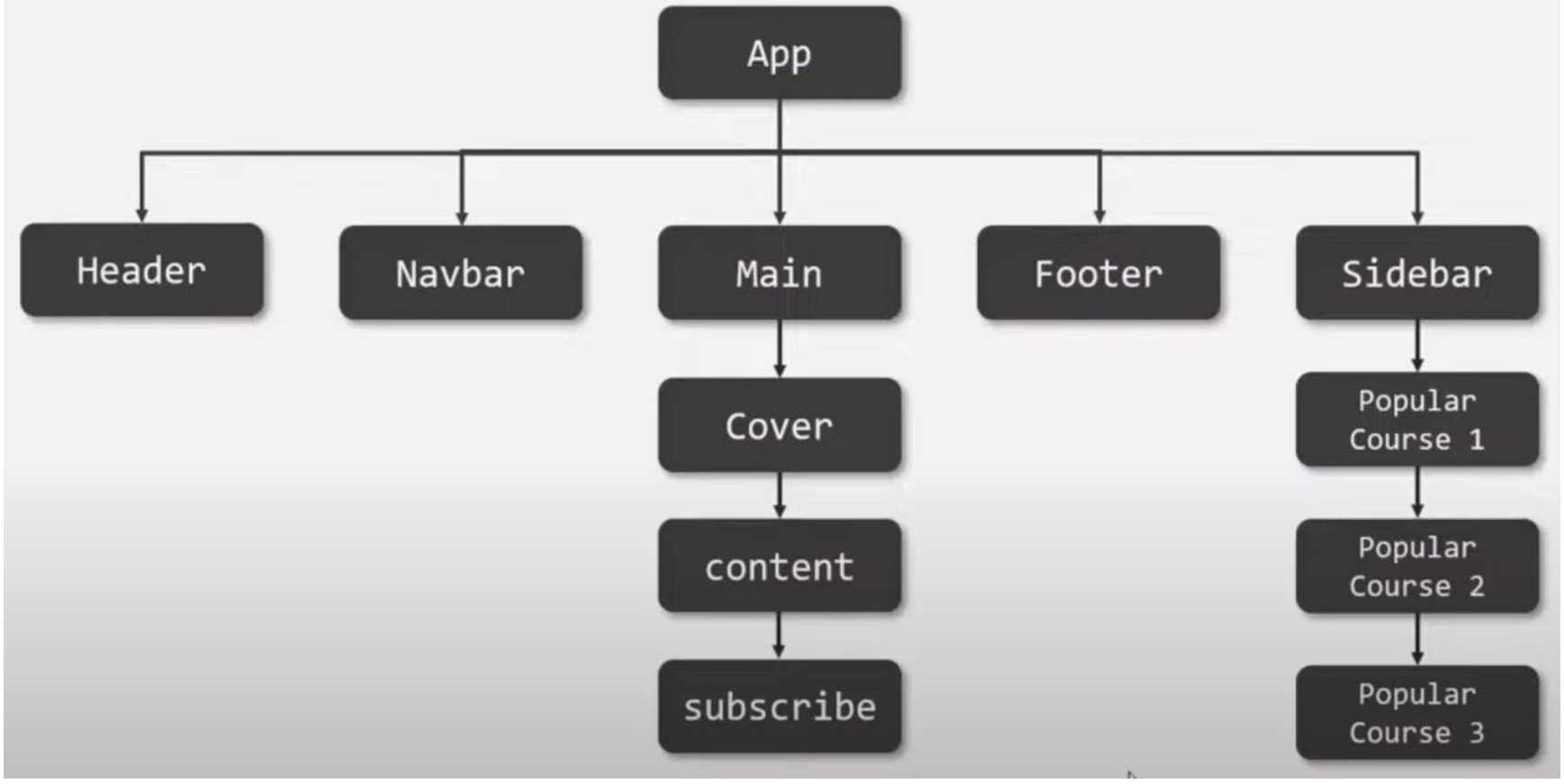
Three Steps to Create and Use a Component:

1. **Create a TypeScript Class** and export it.
2. **Decorate the class with** `@Component` and provide necessary metadata (selector and template).
3. **Declare the component** in the app module (or relevant module).

These steps will enable Angular to recognize, compile, and use the component in the application.

Angular is a component-based JavaScript framework for building client-side application.

1. A component is a piece of user interface.
2. Every Angular application has at least one component
3. An Angular application is essentially a tree of component.
4. Combining all these components together makes an Angular UI



Create a Component

1. Create a TypeScript class & export it
2. Decorate the class with @Component decorator
3. Declare the class in main module file

View Template of Component

The view template of a component is a form of HTML that tells Angular how to render a component.

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
```

Disadvantage of using template property

1. It mixes the HTML & TypeScript code which makes the code less maintainable.
2. Since HTML is written as a string, if there is some error, we will not know about it during compile time.
3. If the number of lines of HTML code is huge, it will be messy & not maintainable.

View Template in Angular

- **Definition**: A *view template* is the HTML content that Angular renders when a component is used in the application. It defines the UI of the component.

Ways to Define a View Template

There are two ways to define a view template for a component:

1. **Using the `template` Property**
    - The `template` property is used to define a small amount of HTML directly as a string in the component's TypeScript file.

Example:

typescript

```typescript
@Component({
  selector: 'app-header',
  template: `<h3>Header Content</h3>`
})
export class HeaderComponent { }
```

- **Advantages**: Simple to use for small amounts of HTML (1-2 lines).
- **Disadvantages**:
  - Mixing HTML with TypeScript code reduces maintainability.
  - HTML errors (e.g., missing closing tags) are only detected at runtime.
  - Messy when handling large amounts of HTML.

2. **Using the `templateUrl` Property**
   - The `templateUrl` property is used to specify the path to an external HTML file. This is preferred when there is a large amount of HTML content.
   - Example:

```typescript
@Component({
  selector: 'app-header',
  templateUrl: './header.component.html'
})
export class HeaderComponent { }
```

- **Advantages**:
  - HTML is kept separate from TypeScript, improving code organization and maintainability.
  - Errors in HTML are easier to detect and manage.
  - Suitable for large amounts of HTML.

Example Workflow

1. Define the selector for the component in the `@Component` decorator.
2. If using the `template` property, directly assign HTML as a string.
3. If using the `templateUrl` property, create a separate HTML file, and link to it using the path in the component.

Conclusion

- Use the `template` property for very small amounts of HTML.
- Use the `templateUrl` property for larger, more maintainable code, even if the HTML is just a few lines.
- Keeping HTML and TypeScript separate leads to cleaner and more maintainable code.

Key Points:

- Mixing HTML and TypeScript using the `template` property can make the code less maintainable.
- Errors in HTML strings are harder to detect when using the `template` property.
- **Best practice**: Prefer using `templateUrl` for better separation of concerns.

Styling View Template

Notes on Styling in Angular Components

View Template

- The HTML file associated with a component that defines its structure.
- Example: `header.component.html` is the view template for the header component.

Ways to Style View Template in Angular

1. `styles` **Property**
    - An inline way to apply CSS directly within the component.
    - You define an array in the component's metadata and write CSS as strings.
    - Example:

```
@Component({
  selector: 'app-header',
  templateUrl: './header.component.html',
  styles: [
    'a { text-decoration: none; margin: 0 10px; }',
    'button { padding: 10px 20px; }',
    '.header-class { width: 100%; height: 70px; }'
  ]
})
```

- Pros: Quick and easy for small CSS.
- Cons:
    - Mixing TypeScript and CSS in the same file is not a good practice.
    - Errors are only caught at runtime.
    - Becomes unmanageable with large CSS code.

2. `styleUrls` **Property**
    - A better way to handle styles by separating them into external `.css` files.
    - Example:

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

- Pros:
  - Keeps the styles in a separate file for better maintainability.
  - Errors in the CSS are easier to manage.
- Example of referencing CSS file:

typescript

```typescript
styleUrls: ['./header.component.css']
```

Advantages of Using `styleUrls` Over `styles`

- Separation of concerns: Keeps CSS and TypeScript separate.
- Easier debugging and maintainability.
- More scalable for large CSS.

Scope of Styles in Angular

- Styles in `styleUrls` or `styles` are scoped only to the component they are defined for.
- Styles do not apply globally or affect child components unless explicitly shared.
- Example:
  - If you style a button in the parent component, it won't affect buttons in the child component unless both use the same CSS file.

Examples of CSS

1. Styling an anchor tag:

```css
a {
  text-decoration: none;
  margin: 0 10px;
}
```

2. Styling a button:

```css
button {
  padding: 10px 20px;
}
```

3. Styling an element with a class:

```css
.header-class {
  width: 100%;
  height: 70px;
}
```

4. Styling an element by ID:

```css
#element-id {
  background-color: red;
}
```

Conclusion

- Use `styleUrls` for better structure and maintainability.
- Understand the scoped nature of Angular component styles—styles are only applied to the specific component unless shared.

```
@Component({
    selector:'app-header',
    templateUrl: './header.component.html',
    styles: ['a{text-decoration: none; margin: 0 10px;}', 'button{padding: 10px 20px;}']
})
```

style the classes and id using styles

```
'button{padding: 10px 20px;}', '.ekart--header{width: 100%; height: 70px}', '#id{}']
```

Disadvantage of using styles property

1. It mixes the CSS & TypeScript code which makes the code less maintainable.
2. Since CSS is written as a string, if there is some error, we will not know about it during compile time.
3. If the number of lines of CSS code is huge, it will be messy & not maintainable.

```ts
@Component({
    selector:'app-header',
    templateUrl: './header.component.html',
    styleUrls: ['./header.component.css']
})
export class HeaderComponent{
```

Adding CSS Styles Globally

TS header.component.ts | # styles.css | # header.component.css

src > # styles.css > ...

```css
/* You can add global styles to this file, and also import other style files
@import url('https://fonts.googleapis.com/css2?family=Concert+One&family=Mont
@import url('https://fonts.googleapis.com/css2?family=Concert+One&family=Mont
@import url('https://cdnjs.cloudflare.com/ajax/libs/font-awesome/4.6.3/css/fo
*{
    margin: 0px;
    padding: 0px;
    box-sizing: border-box;
}
body{
    font-family: 'Montserrat', sans-serif;
}
```

Notes: Applying Global CSS Styles in Angular

1. **Global CSS in Angular**

- **Definition**: In Angular, global CSS styles are applied to all components and HTML elements by defining them in the `style.css` file.
- **Example**:

```css
p {
  font-size: 16px;
}
```

This sets the font size for all `<p>` elements in the Angular app to 16px.

2. **Applying Global Reset Styles**

- **Definition**: Reset styles are applied globally to remove default margins and paddings for all HTML elements.
- **Example**:

```
* {
  margin: 0;
  padding: 0;
  box-sizing: border-box;
}
```

This resets the margin and padding to 0 for all elements and sets `box-sizing` to `border-box`.

3. **Using External Fonts**

- **Definition**: Google Fonts can be imported and used globally across an Angular application by adding an import statement in `style.css`.
- **Steps**:
  1. Visit Google Fonts, select a font.
  2. Copy the `@import` URL.
  3. Paste it at the top of `style.css`.

- **Example**:

```css
@import url('https://fonts.googleapis.com/css2?family=Montserrat&display=swap');

body {
  font-family: 'Montserrat', sans-serif;
}
```

4. Multiple Fonts for Different Elements

- **Definition**: Different fonts can be used for specific parts of the application.
- **Example**:

```css
@import url('https://fonts.googleapis.com/css2?family=Concert+One&display=swap');

.app-logo {
  font-family: 'Concert One', cursive;
}
```

5. Adding Font Awesome Icons

- **Definition**: Font Awesome icons can be integrated into an Angular app globally by adding the Font Awesome CDN link to `style.css`.
- **Steps**:
  1. Get the CDN link from Font Awesome.
  2. Add it to `style.css`.
- **Example**:

```
@import url('https://cdnjs.cloudflare.com/ajax/libs/font-awesome/4.7.0/css/font-awesome.min.css
```

6. **Usage in HTML**

- **Definition**: After importing Font Awesome, use the icon classes directly in HTML.
- **Example**:

```html
<i class="fa fa-home"></i> <!-- Displays a home icon -->
```

Summary:

- Use `style.css` for global styles.
- Import external fonts and icons (Google Fonts, Font Awesome) for use across the application.
- Apply reset styles and global font families in `style.css`.

Using Bootstrap for Styling

Notes on Using Bootstrap in an Angular Project

1. **Bootstrap Overview**

- **Definition**: Bootstrap is a responsive, mobile-first CSS framework for building web applications. It's open-source and comes with features like:

    - **Sass variables**
    - **Mixins**
    - **Responsive grid system**
    - **Pre-built components**
    - **JavaScript plugins**
- **Files**: It consists of CSS and JS files. Some components (like carousels, dropdowns) require JavaScript functionality, and Bootstrap JS has dependencies on **jQuery** and **Popper.js**.

2. Installing Bootstrap in an Angular Project

Method 1: NPM Installation

- Open the terminal in the Angular project directory.
- Run the command:

```bash
npm install bootstrap --save
```

- This will install Bootstrap and store it in the `node_modules` folder.

Checking Installation

- After installation, the `bootstrap` folder will appear in `node_modules`, containing:
    - **CSS folder**: Contains Bootstrap CSS files.
    - **JS folder**: Contains Bootstrap JavaScript files.

3. Using Bootstrap CSS in Angular

Option 1: Modify `angular.json`

- Open the `angular.json` file.
- In the `styles` array, add the path to the `bootstrap.min.css` file:

```
"styles": [
  "node_modules/bootstrap/dist/css/bootstrap.min.css",
  "src/styles.css"
]
```

Option 2: Import in `styles.css`

- Open `src/styles.css`.
- Add an import statement for the Bootstrap CSS file:

```css
@import 'node_modules/bootstrap/dist/css/bootstrap.min.css';
```

4. Using Bootstrap Components

Example: Adding a Button

- To test if Bootstrap is working, add a button to `app.component.html`:

```html
<button class="btn btn-primary">Bootstrap Button</button>
```

- **Rebuild** the Angular app using `ng serve` to apply Bootstrap styles.

Check: After rebuilding, the button should display with Bootstrap styling.

5. Uninstalling Bootstrap

- To uninstall Bootstrap, run:

```bash
npm uninstall bootstrap
```

- Check the `node_modules` folder to confirm that Bootstrap is removed.

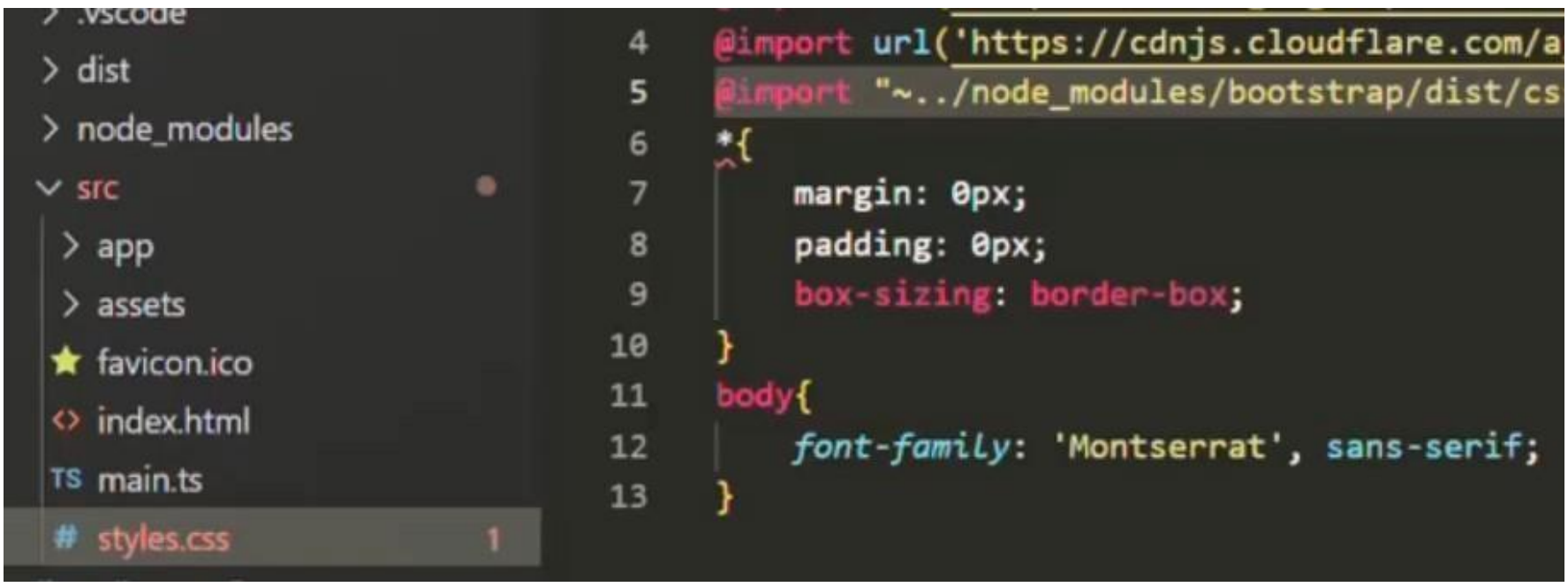
- Check the `node_modules` folder to confirm that Bootstrap is removed.

6. **Common Bootstrap Components** (Refer to Bootstrap Documentation):

- **Buttons**: `<button class="btn btn-primary">Button</button>`
- **Forms**: Pre-built form layouts.
- **Grid system**: A responsive grid for layout control.

```
],
"styles": [
  "node_modules/bootstrap/dist/css/bootstrap.min.css",
  "src/styles.css"
]
```

OR



Create Component using Angular CLI

```
ng generate component component-name
```

1. Creates a component class decorated with @Component decorator
2. Generates the view template & stylesheet for that component
3. Registers the component class in the main module.

Angular Component Creation Overview

- **Manual Component Creation Recap**:
  - Header component created manually with `@Component` decorator.
  - Component registered in `appmodule.ts` under `declarations` array.

- **Angular CLI for Component Creation**:
  - Command: `ng generate component <component-name>` or `ng g c <component-name>`.
  - CLI automates:
    - Creation of component class with `@Component` decorator.
    - Creation of view template and stylesheet.
    - Automatic registration of component in the main module (`app.module.ts`).

Fixing Angular Compilation Error (Bootstrap Removal)

- **Issue**: Uninstalled Bootstrap, but still referencing `bootstrap.min.css` in `angular.json` caused compilation failure.
- **Solution**:
  - Remove Bootstrap reference from the `styles` array in `angular.json`.
  - After fixing, restart the server (`Ctrl + C` → `ng serve`).

Creating a Component Using Angular CLI (Top Header Example)

1. **Command**: `ng generate component top-header` or `ng g c top-header`.
2. **Result**:
   - A new folder `top-header` is created with component-related files.
   - Class `TopHeaderComponent` created with `@Component` decorator.
   - Selector `app-top-header` generated (customized to `top-header`).

3. **HTML Structure**:
    - Update `top-header.component.html` with the desired HTML structure.
    - Replace placeholder `<p>` with custom HTML.
4. **CSS Styling**:
    - Add styles in `top-header.component.css` as required.
5. **Using the Component**:
    - Add `<top-header></top-header>` in `app.component.html` to render the new component.
6. **No Manual Registration Needed**:
    - Angular CLI automatically declares the new component in `app.module.ts`.

Child Component Creation and Styling (Top Menu Example)

- **Scenario**: Top menu as a child component of header.
- **Steps**:
    1. Navigate to the header component folder: `cd src/app/header`.
    2. Create `top-menu` component: `ng g c top-menu`.
    3. Move related HTML and CSS for the top menu into `top-menu.component.html` and `top-menu.component.css`.
    4. Add the `top-menu` selector in `header.component.html` to render it.
- **Handling CSS Isolation**:
    - Each component has isolated CSS.
    - Copy relevant styles from `header.component.css` to `top-menu.component.css` to apply them.
- **Global Styles**:
    - If common styles (e.g., for anchor tags) are needed globally, add them to `styles.css` for global scope.

Summary of Key Actions

1. Fix Bootstrap error by removing reference from `angular.json`.
2. Create components using Angular CLI.
3. Move component-specific HTML and CSS to respective component files.
4. Register components automatically using Angular CLI.
5. Use child components in parent component templates.
6. Apply isolated or global styles as needed.

Component Selectors in Angular

1. Selector as an HTML Tag

- **Definition**: A component selector is used like an HTML tag. Wherever this tag is used, the component’s view template will be rendered.
- **Example**:

```
<top-header></top-header>
```

- In this case, the `top-header` component will be rendered as an HTML tag.

2. **Selector as an HTML Attribute**

- **Definition**: A component selector can also be used as an HTML attribute by wrapping the selector name in square brackets. The component will no longer work as an HTML tag.
- **Example**:

```
// In component.ts
@Component({
  selector: '[top-header]',
  templateUrl: './top-header.component.html'
})
```

```
<!-- In app.component.html -->
<div top-header></div>
```

- Here, `top-header` is used as an attribute on a `div`.

3. **Selector as a CSS Class**

- **Definition**: By prefixing the selector name with a dot (`.`), the component selector can be used as a CSS class.
- **Example**:

```
// In component.ts
@Component({
  selector: '.top-header',
  templateUrl: './top-header.component.html'
})
```

```
<!-- In app.component.html -->
<div class="top-header"></div>
```

- The component is now applied using a CSS class on the `div`.

4. Selector as an ID

- **Definition**: A component selector can also be used as an ID by using a hash (`#`) symbol before the selector name.
- **Example**:

```
// In component.ts
@Component({
  selector: '#top-header',
  templateUrl: './top-header.component.html'
})
```

```
<!-- In app.component.html -->
<div id="top-header"></div>
```

- The component will now be rendered as an ID.

5. Common Use Case: Angular Directives

- **Definition**: Although you can use component selectors as attributes, classes, or IDs, this is less common. Angular generally uses attribute selectors for **directives** rather than components.
- **Example**: Attribute selectors are mainly used in custom directives. When working with components, it's more common to use the selector as an HTML tag.

6. Conclusion

- While Angular supports various ways to use selectors (HTML tag, attribute, class, ID), the most common approach for components is to use them as HTML tags. Attribute selectors are primarily used for Angular directives.

What is Data Binding

Data Binding in Angular

Overview

- **Definition**: Data binding in Angular allows communication between a component class and its view template. It facilitates the flow of data between these two parts.
- **Purpose**: To synchronize data between the component class (UI logic) and the view template (HTML).

Types of Data Binding

1. **One-Way Data Binding**
    - **Definition**: Data flows in a single direction, either from the component to the view template or from the view template to the component.
    - **From Component to View Template**
        - **Method**: String interpolation or property binding.
        - **Example**:

```
<!-- String Interpolation -->
<p>{{ title }}</p>
<!-- Property Binding -->
<input [value]="title">
```

- **Explanation**: The value of `title` from the component class is displayed in the view template.

- **From View Template to Component Class**
  - **Method**: Event binding.
  - **Example**:

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

  - **Explanation**: The `onClick` method in the component class is triggered when the button is clicked.

2. **Two-Way Data Binding**
   - **Definition**: Data flows in both directions simultaneously; changes in the component class reflect in the view and vice versa.
   - **Method**: `ngModel`
   - **Example**:

```html
<input [(ngModel)]="title">
```

   - **Explanation**: Changes to the `title` input field will update the component class and any changes to `title` in the component will update the input field.

Key Concepts

- **String Interpolation**: Used for inserting component data into the view template using double curly braces (`{{ }}`).
- **Property Binding**: Used to bind data to an HTML property using square brackets (`[ ]`).
- **Event Binding**: Used to listen to events (like clicks) and execute component methods using parentheses (`( )`).
- **Two-Way Binding**: Achieved with `ngModel`, synchronizing data between the view and the component.

Conclusion

- Data binding is crucial for dynamic communication between components and templates in Angular.
- **One-Way Data Binding** allows unidirectional data flow, while **Two-Way Data Binding** supports bidirectional data synchronization.

Next Steps

- **String Interpolation**: To be covered in detail for achieving one-way data binding from component to view template.

String Interpolation

String Interpolation

- **Definition**: String interpolation is a way to display data from the component class within the HTML template. It's a form of one-way data binding that inserts data directly into the template.
- **Syntax**: Use `{{ expression }}` syntax to include expressions in HTML.
- **Usage**: Primarily for displaying static or computed data in the template (e.g., displaying the title of a product).

```
export class ProductListComponent {
  productTitle: string = 'iPhone 13';
}
```

html

```
<!-- In product-list.component.html -->
<h1>{{ productTitle }}</h1> <!-- Displays: iPhone 13 -->
```

Summary

- **String Interpolation**: `{{ expression }}` is used to dynamically display data in the view.
- **Expressions**: Any TypeScript expression can be placed inside `{{ }}`, including property values, arithmetic operations, function calls, and conditional expressions.
- **Use Case**: Ideal for passing data from the component to the view template in a one-way, readable-only format.

Name: iPhone
Price: $999
Color: Matte black
Discounted Price: $798

Types of Data Binding

1. One-Way Data Binding

- **Definition**: Allows data to be passed from the component class to its view template but not vice versa.

- **Methods**:
  - **String Interpolation**: Uses double curly braces `{{ }}` to display values from the component in the view.
  - **Property Binding**: Binds HTML element properties directly to component properties.

2. Two-Way Data Binding

- **Not covered in this text**: Two-way binding allows for bidirectional data flow, where changes in the view update the component, and vice versa.

Examples of String Interpolation

Example 1: Displaying Properties

- **Code**:

```
<p>Name: {{ product.name }}</p>
<p>Price: {{ product.price }}</p>
<p>Color: {{ product.color }}</p>
```

- **Explanation**: Displays values of properties `name`, `price`, and `color` of an object `product` in the HTML view.

Example 2: Concatenating Strings

- **Code:**

```html
<p>Price: {{ '$' + product.price }}</p>
```

- **Explanation**: Uses concatenation to display the dollar symbol (`$`) before the `price` value.

Example 3: Arithmetic Operations

- **Code:**

```html
<p>Discounted Price: {{ product.price - (product.price * product.discount / 100) }}</p>
```

- **Explanation**: Calculates the discounted price using an arithmetic expression and displays it.

Example 4: Using Functions

- **Code in Component:**

```typescript
getDiscountedPrice(): number {
    return this.product.price - (this.product.price * this.product.discount / 100);
}
```

```
export class ProductListComponent {
  product = {
    name: 'iPhone X',
    price: 789,
    color: 'Black',
    discount: 8.5,
    inStock: 0
  }

  getDiscountedPrice() {
    return this.product.price - (this.product.price * this.product.discount / 100)
  }
}
```

```
<p>Name: {{ product.name }}</p>
<p>Price: {{ '$' + product.price }}</p>
<p>Color: {{product.color}}</p>
<p>Discounted Price: {{ getDiscountedPrice().toFixed(2) }}</p>
<p>{{product.inStock > 0 ? 'Only' + product.inStock +' items left': 'Not in Stock'}}</p>
```

- **Code in Template:**

  html — Copy code

  ```
  <p>Discounted Price: {{ getDiscountedPrice() }}</p>
  ```

- **Explanation:** Calls a function in the component to compute and display the discounted price in the view.

Example 5: Formatting Numbers

- **Code:**

  html

  ```
  <p>Discounted Price: {{ getDiscountedPrice().toFixed(2) }}</p>
  ```

- **Explanation:** Formats the discounted price to display only two decimal places using the `.toFixed()` method.

Example 6: Conditional (Ternary) Operator

- **Code:**

  html — Copy code

  ```
  <p>{{ product.inStock > 0 ? 'Only ' + product.inStock + ' items left' : 'Not in stock'
  ```

- **Explanation:** Uses a ternary operator to check stock availability, displaying either the number of items left or a "Not in stock" message.

Property Binding

Interpolation is used to just display a piece of data in HTML, such as displaying a title or a name.

Property binding lets us bind a property of a DOM object, for example the hidden property, to some data value. This can let us show or hide a DOM element, or manipulate the DOM in some other way.

Property Binding

- **Definition**: Property binding also achieves one-way data binding by binding a property of a DOM element to a component's property.
- **Syntax**: Use `[property]="expression"` syntax to bind a DOM element's property to a dynamic value in the component class.
- **Usage**: For binding properties of HTML elements, particularly for attributes like `src`, `disabled`, `hidden`, etc., that may need to change based on component data.

Example:

typescript

```typescript
// In product-list.component.ts
export class ProductListComponent {
  productImage: string = 'assets/images/iphone.png';
  isAvailable: boolean = true;
}
```

```html
<!-- In product-list.component.html -->
<img [src]="productImage" alt="Product Image"> <!-- Binds productImage to src -->

<button [disabled]="!isAvailable">Buy Now</button> <!-- Binds isAvailable to disabled -->
```

Key Differences Between String Interpolation and Property Binding

- **String Interpolation**: Best for directly displaying data in text form within HTML.
- **Property Binding**: Necessary for binding properties of HTML elements that do not support string interpolation, like `disabled`, `hidden`, and `checked`.

Advanced Binding Techniques

Attribute Binding

- **Definition**: When binding HTML attributes (e.g., `aria-*`, `data-*`, or `colspan`), property binding may not work as they are not standard properties. Use `attr.` prefix to bind these.
- **Syntax**: Use `[attr.attributeName]="expression"` to bind the attribute dynamically.

Example:

typescript

```
<!-- In product-list.component.html -->
<input [attr.aria-label]="productTitle" />
```

Alternative Syntax: `bind-` Prefix

- **Explanation**: Angular also provides an alternative syntax for property binding using `bind-propertyName="expression"`.

Example:

```
<!-- Alternative syntax for property binding -->
<img bind-src="productImage" alt="Product Image">
```

Component Code:

typescript

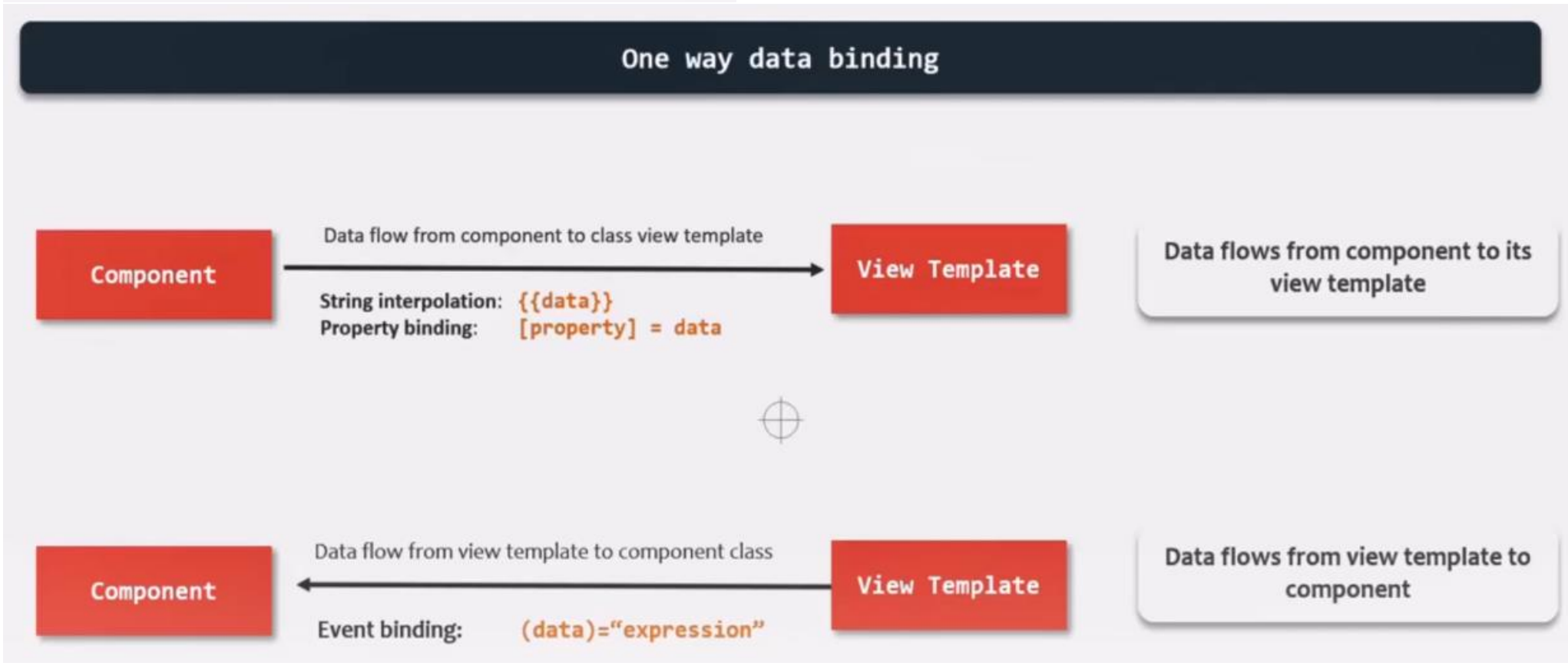
```
// product-list.component.ts
export class ProductListComponent {
  productTitle: string = 'iPhone 13';
  productImage: string = 'assets/images/iphone.png';
  isAvailable: boolean = true;
  name: string = 'John Doe';
}
```

Template Code:

```html
<!-- product-list.component.html -->
<h1>{{ productTitle }}</h1>
<img [src]="productImage" alt="Product Image">

<button [disabled]="!isAvailable">Buy Now</button>

<input [attr.aria-label]="productTitle" bind-value="name">
```

These concepts and examples cover the usage of string interpolation, property binding, attribute binding, and alternative syntax options.

Event Binding



Notes on Event Binding in Angular

Event Binding Overview:

- Event binding in Angular allows data to flow from the view (template) to the component class. It is a one-way data binding approach that enables the application to respond to user actions, such as clicks, key presses, and input changes.
- Event binding syntax: `(eventName)="expressionOrMethod"`

Key Concept:

- When an event occurs in the template (e.g., a button click or text input), Angular captures the event and can pass data to the component or trigger a function in the component class.

Example 1: Button Click Event Binding

html

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

Component Class:

typescript

```typescript
onButtonClick(): void {
  console.log('Button was clicked!');
}
```

Explanation: When the button is clicked, the `onButtonClick()` method is triggered, and the message 'Button was clicked!' is logged to the console.

Example 2: Handling Input Events

html

```html
<input (input)="onInputChange($event)" placeholder="Type something" />
<p>Typed Value: {{ inputValue }}</p>
```

```typescript
inputValue: string = '';

onInputChange(event: Event): void {
  const target = event.target as HTMLInputElement;
  this.inputValue = target.value;
}
```

Explanation: Each time a user types in the input field, the `onInputChange()` method is called. The method extracts the value from the event object and updates the `inputValue` property, which is displayed in the paragraph below.

Example 3: Passing Parameters with Event Binding

html

```
<button (click)="onAction('Save')">Save</button>
<button (click)="onAction('Delete')">Delete</button>
```

Component Class:

typescript

```
onAction(action: string): void {
  console.log(`${action} button clicked`);
}
```

Explanation: Clicking on either button triggers the `onAction()` method with a specific parameter (e.g., 'Save' or 'Delete'), and the action is logged to the console.

Understanding the `$event` Object:

- `$event` is a variable that contains the event object emitted by the DOM. It provides details about the event, such as the type of event, target element, and more.
- Common use cases include handling input changes or getting information from event objects.

Example 4: Accessing Event Properties

html

```
<input (keyup)="logKey($event)" />
```

Component Class:

typescript

```
logKey(event: KeyboardEvent): void {
  console.log(`Key pressed: ${event.key}`);
}
```

Explanation: Each time a key is pressed in the input field, the `logKey()` method logs the key pressed to the console using the `event.key` property.

Tips for Using Event Binding:

- Ensure you only pass functions or expressions that handle logic appropriately within your component class.
- Use the `$event` object wisely to access properties when necessary.
- Combine event binding with property binding to achieve two-way data binding using `[(ngModel)]` for form controls.

Two Way Data Binding

Data Binding in Angular

1. Overview

- Data binding in Angular connects the component class and the view template.
- Types of data binding include:
  - **One-way data binding**: Data flows in a single direction (either from the component to the view or from the view to the component).
  - **Two-way data binding**: Data flows in both directions (component to view and view to component).

2. One-Way Data Binding

- **From Component to View:**
  - Achieved using **string interpolation** (`{{property}}`) or **property binding** (`[property]="expression"`).
- **From View to Component:**
  - Achieved using **event binding** (`(event)="handlerFunction()"`).

3. Event Binding in Angular

- Event binding transfers data from the template to the component when an event occurs (e.g., user input, clicks).
- Syntax: `(eventName)="componentMethod($event)"`

Example:

```ts
// search.component.ts
export class SearchComponent {
  searchText: string = 'Initial Value';

  updateSearchText(event: any) {
    this.searchText = event.target.value;
  }
}
```

```html
<!-- search.component.html -->
<input (input)="updateSearchText($event)" [value]="searchText">
<p>Search result: {{ searchText }}</p>
```

- In this example, the input field is bound to the `updateSearchText()` method, and the `searchText` property is updated whenever the user types.

4. Two-Way Data Binding

- Combines property binding and event binding.
- Simplified using Angular's built-in `ngModel` directive.
- **Syntax:** `[(ngModel)]="property"`

Example with `ngModel`:

```ts
// search.component.ts
import { Component } from '@angular/core';

@Component({
  selector: 'app-search',
  templateUrl: './search.component.html'
})
export class SearchComponent {
  searchText: string = 'Initial Value';
}
```

```html
<!-- search.component.html -->
<input [(ngModel)]="searchText">
<p>Search result: {{ searchText }}</p>
```

- For `ngModel` to work, ensure the `FormsModule` is imported in `app.module.ts`:

```ts
// app.module.ts
import { FormsModule } from '@angular/forms';

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

Key Points:

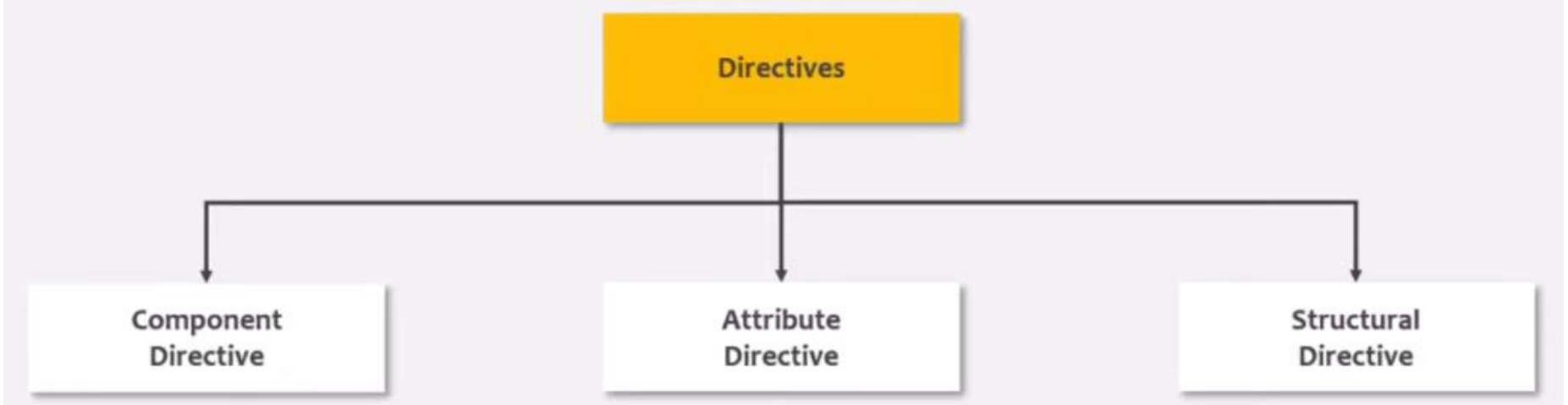
- **Event Binding**: Used when you want to respond to user interactions.
- **Two-Way Binding** (`ngModel`): Makes it easy to keep the data in sync between the view and component.
- Import `FormsModule` from `@angular/forms` for `ngModel` to function correctly.

Conclusion: Event binding and two-way binding provide powerful ways to interact with user inputs and keep the UI and data synchronized seamlessly in Angular applications.

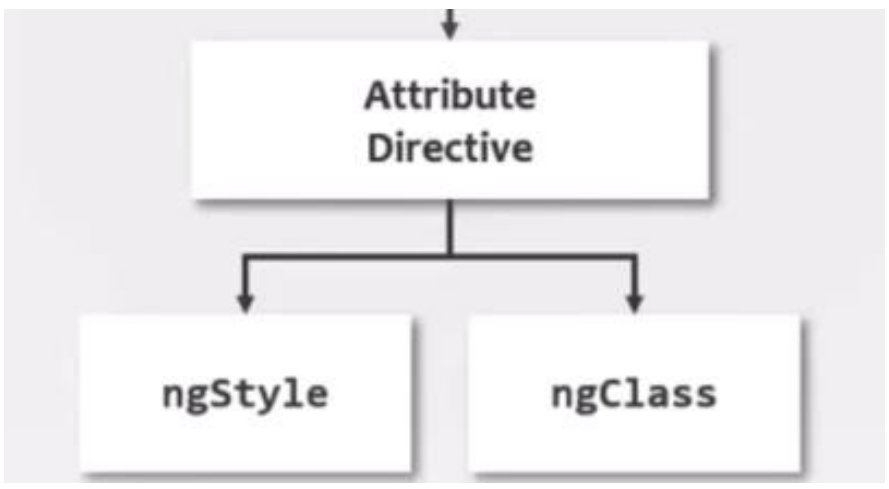
Understanding Directives

A Directive is an instructions to the DOM

- Manipulate DOM
- Change Behavior
- Add / Remove DOM Elements

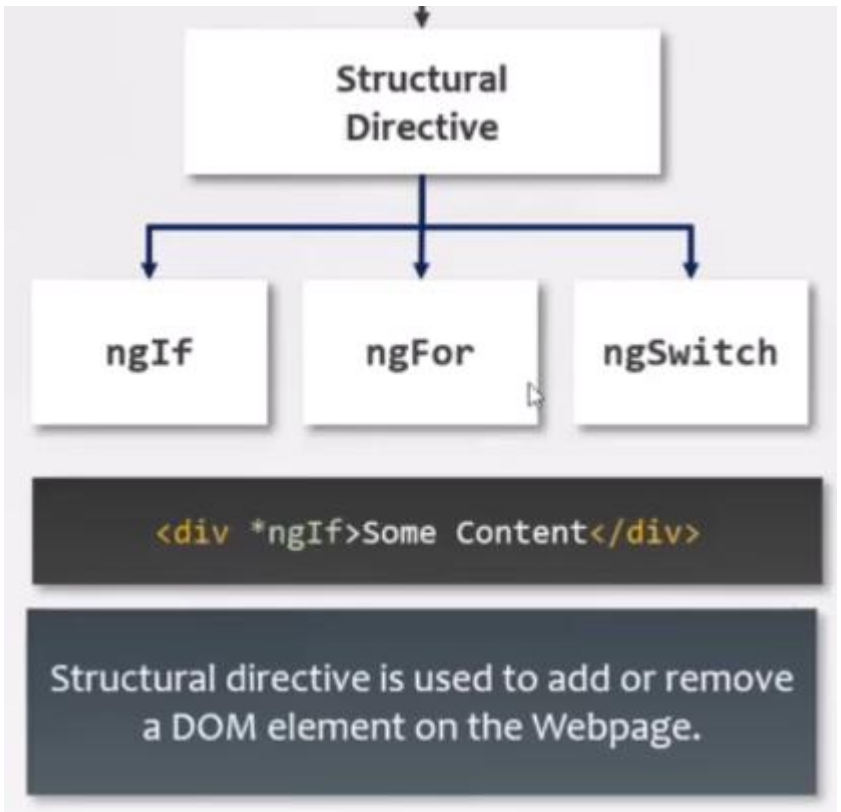


Component directive is the angular component. It is a directive with a template.




```
<div changetoGreen>Some Content</div>
```

Attribute directive is used to change the appearance or behavior of a DOM element.




```
<div changeToGreen>Some Content</div>
```

```
@Directive({
    selector: '[changeToGreen]'
})
export class ChangeToGreen{

}
```

Angular Directives Overview

Definition: Directives in Angular are instructions given to the DOM (Document Object Model) that allow developers to modify the behavior, appearance, or layout of elements on a web page. They extend HTML by adding custom behavior.

Types of Directives

1. **Component Directives**
    - **Definition**: A component directive is an Angular component itself. It has a template and is used to render specific content and implement business logic.
    - **Example**:

```
@Component({
  selector: 'app-example',
  template: `<h1>Hello, World!</h1>`
})
export class ExampleComponent {}
```

- **Explanation**: When the `app-example` selector is used in the template, it instructs Angular to render the component's view and logic at that location.

2. **Attribute Directives**

- **Definition:** Attribute directives modify the appearance or behavior of an element. Unlike component directives, they do not have templates and do not render or remove elements.
- **Example:**

```
@Directive({
  selector: '[changeToGreen]'
})
export class ChangeToGreenDirective {
  constructor(private el: ElementRef) {
    this.el.nativeElement.style.backgroundColor = 'green';
  }
```

- **Explanation:** This directive changes the background color of an element to green when applied as an attribute, e.g., `<div changeToGreen></div>`.
- **Built-in Examples:**
  - `ngClass`: Dynamically adds or removes classes.
  - `ngStyle`: Applies inline styles based on an expression.

3. **Structural Directives**

- **Definition:** Structural directives modify the DOM layout by adding or removing elements.
- **Examples:**

```
<div *ngIf="isVisible">This is conditionally displayed.</div>
<div *ngFor="let item of items">{{ item }}</div>
```

- **Built-in Structural Directives:**
  - `*ngIf`: Adds/removes elements based on a condition.
  - `*ngFor`: Repeats an element for each item in a list.
  - `*ngSwitch`: Displays one element from a set based on a condition.
- **Note:** Structural directives are prefixed with an asterisk (`*`) to indicate they change the structure of the DOM.

Creating a Custom Directive

Example: Creating an attribute directive to change an element's behavior.

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

Usage:

```
html

<p appHighlight>This text will be highlighted in blue.</p>
```

Explanation: Custom directives extend the functionality of Angular applications by allowing you to create reusable features.

ngFor Directives

Angular ngFor directive iterates over a collection of data like an array, list, etc. and creates an HTML element for each of the items from an HTML template.

1. The ngFor directive is used to repeat a portion of HTML template once per each item for an itterable list
2. The ngFor directive is a structural directive. It manipulates the DOM by adding / removing elements from the DOM.

Angular Directives: `*ngFor`

Definition

The `*ngFor` directive is a structural directive in Angular used to iterate over a collection, such as an array or list, and create a corresponding HTML element for each item. It dynamically manipulates the DOM by adding or removing elements based on the iteration.

Key Points

- **Structural Directive**: A type of directive that changes the structure of the DOM (e.g., adding or removing elements).
- **Syntax**: The directive is prefixed with an asterisk (`*`) to indicate it manipulates the DOM.
- **Iterating**: It uses the `let` keyword to create a variable that holds the current item during iteration, followed by `of` to specify the array or collection.

Example 1: Basic Iteration

HTML Template

```html
<div *ngFor="let item of items">
  <p>Current element of array is: {{ item }}</p>
</div>
```

Explanation

- `*ngFor="let item of items"` : Loops over each element in the `items` array.
- `{{ item }}` : Displays the current element in each iteration within a `<p>` tag.

Result

If `items = [2, 4, 6]`, the output will be:

```sql
Current element of array is: 2
Current element of array is: 4
Current element of array is: 6
```

Example 2: Using a Property from a Component

TypeScript Component (`product-list.component.ts`)

```typescript
export class ProductListComponent {
  listOfStrings: string[] = ['Apple', 'Banana', 'Cherry'];
}
```

HTML Template (`product-list.component.html`)

```html
<div *ngFor="let fruit of listOfStrings">
  <p>{{ fruit }}</p>
</div>
```

Result

The above code will render a paragraph for each string in the `listOfStrings` array.

Example 3: Dynamic Menu Generation

TypeScript Component (`main-menu.component.ts`)

```typescript
export class MainMenuComponent {
  mainMenuItems: string[] = ['Home', 'Products', 'Sale', 'New Arrival', 'Contact',
}
```

HTML Template (`main-menu.component.html`)

```html
<a *ngFor="let menuItem of mainMenuItems" href="#">
  {{ menuItem }}
</a>
```

Explanation

- The `*ngFor` directive repeats the `<a>` (anchor) element for each item in `mainMenuItems`.
- The variable `menuItem` holds the current menu item, which is displayed as the anchor text.

Result

This will create anchor links like:

```sql
Home | Products | Sale | New Arrival | Contact | Services
```

Common Errors and Troubleshooting

- **Error**: "Can't bind to `*ngFor` since it isn't a known property."
  - **Solution**: Ensure the array or list is defined in the component and is accessible.
- **Error**: Using `in` instead of `of`.
  - **Correction**: The correct syntax for iteration in Angular is `let item of array`, not `in`.

These notes outline how the `*ngFor` directive functions, complete with example code and explanations for each part of the process.

Rendering List Of Complex Type

Using `*ngFor` Directive to Loop Over Complex Types

Definition: `*ngFor` is an Angular structural directive that allows iteration over a collection and dynamically renders an HTML template for each item. It is useful for displaying lists or sets of data in a component template.

Example Use Case: When working with a list of complex types (e.g., an array of product objects), `*ngFor` can be used to display the details of each product on a webpage.

Scenario Overview:

- You have an array of product objects, each containing various properties like `id`, `name`, `description`, `brand`, `price`, etc.
- You want to display the product details (e.g., name, price, image) using the `*ngFor` directive.

Example Code:

1. **Product List Array in Component:**

```
// product-list.component.ts
import { Component } from '@angular/core';

@Component({
  selector: 'app-product-list',
  templateUrl: './product-list.component.html',
  styleUrls: ['./product-list.component.css']
})
export class ProductListComponent {
```

```
products = [
  {
    id: 1,
    name: 'Product A',
    description: 'Description of Product A',
    brand: 'Brand A',
    price: 100,
    imageUrl: 'path/to/imageA.jpg'
  },
  {
    id: 2,
    name: 'Product B',
    description: 'Description of Product B',
    brand: 'Brand B',
    price: 200,
    imageUrl: 'path/to/imageB.jpg'
  },
  // Additional product objects...
];
```

```
<!-- product-list.component.html -->
<div class="product-container">
  <div *ngFor="let product of products" class="product-item">
    <img [src]="product.imageUrl" alt="{{ product.name }}">
    <h3>{{ product.name }}</h3>
    <p>Price: ${{ product.price }}</p>
    <p>Brand: {{ product.brand }}</p>
    <p>{{ product.description }}</p>
  </div>
</div>
```

3. **CSS Styling**:

```css
/* product-list.component.css */
.product-container {
  display: flex;
  flex-wrap: wrap;
  gap: 20px;
}
.product-item {
  border: 1px solid #ddd;
  padding: 10px;
  width: 200px;
}
.product-item img {
  max-width: 100%;
  height: auto;
}
```

Explanation:

- The `*ngFor` directive is used on the `<div>` element to repeat the block for each product in the `products` array.
- The `let product of products` syntax means that for each iteration, `product` refers to the current item.
- Property binding (e.g., `[src]="product.imageUrl"`) is used to dynamically set the `src` attribute for the image.
- String interpolation (`{{ product.name }}`) is used to display the product's properties.

Steps to Create a New Component:

- Use the Angular CLI to generate a new component:

```bash
ng generate component product-list
```

- Move to the appropriate directory if needed and register the new component in `app.module.ts` by adding it to the `declarations` array.

ngIf Directives

Angular ngIf directive is a structural directive that is used to completely add or remove a DOM element from the webpage based on a given condition.

Angular `*ngIf` Directive Notes

Definition

The `*ngIf` directive in Angular is a structural directive used to conditionally add or remove elements from the DOM based on a given Boolean condition. When the condition evaluates to `true`, the element is added to the DOM; when it evaluates to `false`, the element is removed.

Example 1: Basic Usage

Consider a search component with a text box where user input is displayed in a paragraph element:

html Copy code

```
<input type="text" [(ngModel)]="searchText" placeholder="Type here...">
<p *ngIf="searchText !== ''">Search result for: {{ searchText }}</p>
```

Explanation:

- The paragraph (`<p>`) is only displayed if `searchText` is not an empty string (`''`).
- If `searchText` is empty, the paragraph is removed from the DOM.

Code Details:

- The condition `searchText !== ''` is a TypeScript expression that returns `true` when there is input in the text box.
- Initially, `searchText` can be set to an empty string (`searchText = ''`), ensuring the paragraph is hidden until the user types something.

Example 2: Using `*ngIf` with Product Listings

Suppose we have a product list where only items with a discount should display a "percent off" label:

html Copy code

```
<div *ngFor="let product of products">
  <div *ngIf="product.discountPrice">
    {{ 100 - (product.discountPrice / product.price * 100).toFixed(0) }}% off
  </div>
</div>
```

Explanation:

- The `*ngFor` directive iterates over `products`.
- The `*ngIf` directive ensures the label is only shown if `product.discountPrice` exists.
- The calculation `(100 - (product.discountPrice / product.price * 100).toFixed(0))` determines the percentage discount.

Summary

- The `*ngIf` directive is used to manage the presence of elements in the DOM.
- It is particularly useful for showing or hiding UI elements based on conditions.
- When combined with other directives like `*ngFor`, it can dynamically control element rendering based on data properties.

ngStyle Directives

The ngStyle directive is an attribute directive which allows us to set many inline style of an HTML element using expression.

Angular `ngStyle` Directive Notes

Definition

The `ngStyle` directive is an attribute directive in Angular that allows developers to apply and control multiple inline styles on an HTML element using TypeScript expressions. This enables dynamic styling based on conditions or expressions evaluated at runtime.

Example Explanation

Imagine you have a product card where you want to show a label indicating whether a product is in stock or not. The text should appear in bold and change color depending on the product's availability.

Code Example:

```
<div class="product-card">
  <div [ngStyle]="{'font-weight': 'bold', 'color': product.isInInventory ? 'green' : 'red'}">
    {{ product.isInInventory ? 'Available in stock' : 'Not available in stock' }}
  </div>
</div>
```

Explanation:

- `ngStyle` is used here as an attribute directive enclosed within square brackets.
- The inline styles are defined using curly braces (`{}`), where each property is assigned dynamically:
  - `'font-weight': 'bold'` ensures the text is bold.
  - `'color': product.isInInventory ? 'green' : 'red'` sets the text color to green if `product.isInInventory` is `true` , otherwise to red if `false` .
- The `product.isInInventory` condition is evaluated as a TypeScript expression and determines the color of the text.

Dynamic Styling:

- Unlike using static `style` attributes, `ngStyle` allows for dynamic changes based on data-bound properties, making it a powerful tool for responsive UI behavior in Angular applications.

Example 1: Applying Multiple Styles Conditionally

```
<div [ngStyle]="{
  'font-size': product.isPremium ? '20px' : '14px',
  'background-color': product.isInStock ? 'lightgreen' : 'lightcoral',
  'border': product.isHighlighted ? '2px solid blue' : '1px solid gray'
}">
  {{ product.name }}
</div>
```

Explanation:

- `font-size` is set to `20px` if the product is premium, otherwise `14px`.
- `background-color` is `lightgreen` if the product is in stock, otherwise `lightcoral`.
- `border` changes depending on whether the product is highlighted.

Example 2: Styling Based on Component Property

```
// product.component.ts
productStyles = {
  color: 'darkblue',
  'font-weight': 'bold'
};
```

html

```
<!-- product.component.html -->
<div [ngStyle]="productStyles">
  Special Product Offer!
</div>
```

Explanation:

- Styles are pre-defined in the component and applied to the element through `ngStyle`.

Example 3: Conditional Style Using a Method

```
// product.component.ts
getStockStyle(isInStock: boolean) {
  return {
    color: isInStock ? 'green' : 'red',
    'text-decoration': isInStock ? 'none' : 'line-through'
  };
}
```

```
<!-- product.component.html -->
<div [ngStyle]="getStockStyle(product.isInStock)">
  {{ product.isInStock ? 'In Stock' : 'Out of Stock' }}
</div>
```

Explanation:

- A method is used to determine the style based on the `isInStock` property. This method returns an object with style properties and values.

Example 4: Styling with Multiple Conditions

```
<div [ngStyle]="{
  'padding': '10px',
  'color': product.rating > 4.5 ? 'gold' : 'black',
  'font-style': product.isNewArrival ? 'italic' : 'normal',
  'opacity': product.isInStock ? 1 : 0.5
}">
  {{ product.name }} - Rating: {{ product.rating }}
</div>
```

Explanation:

- This element is styled with multiple conditions:
  - The `color` changes to gold if the rating is greater than `4.5`.
  - `font-style` is set to `italic` if the product is a new arrival.
  - `opacity` is reduced to `0.5` if the product is not in stock.

These examples illustrate how `ngStyle` can be leveraged to apply complex, conditional styles dynamically in Angular applications.

ngClass Directives

The ngClass Directive is an Attribute Directive, which allows us to add or remove CSS classes to or from an HTML element dynamically, based on some TypeScript expression.

The `ngClass` directive in Angular is a built-in attribute directive that allows you to dynamically add or remove CSS classes from an HTML element based on a TypeScript expression. It enables the conditional application of styles based on the component's state.

Key Points:

1. **Definition:**

   The `ngClass` directive is used to add or remove CSS classes from an HTML element based on a TypeScript expression.

2. **Usage:**

   To apply dynamic classes to an element, use the `ngClass` directive inside square brackets `[ ]` for property binding. Inside the binding, use a TypeScript expression that evaluates to `true` or `false`.

3. **Example:**

```
<button [disabled]="!searchText" [ngClass]="{
    'btn': true,
    'btn-search': searchText.length > 0,
    'btn-disabled': !searchText
}">Search</button>
```

- `btn` is always applied since it's set to `true`.
- `btn-search` is applied if `searchText` has a value.
- `btn-disabled` is applied when `searchText` is empty.

4. **How It Works:**

   - The `ngClass` directive expects an expression inside the square brackets. This expression can be an object, array, or string that evaluates to the CSS classes.
   - If a property evaluates to `true`, the class is added; if `false`, it’s removed.
   - In the example, the `searchText` property is checked for length: if non-empty, a specific CSS class (`btn-search`) is applied.

5. **Binding with Typescript Expression:**
   The classes inside `ngClass` are dynamically added or removed based on the Boolean result of the TypeScript expressions. For example:

```
searchText: string = '';
```

- If `searchText` is an empty string, `ngClass` applies a `btn-disabled` class.
- If `searchText` is non-empty, it applies the `btn-search` class.

6. **Falsy and Truthy Values:**
   - In TypeScript/JavaScript, falsy values include `false`, `null`, `undefined`, `0`, `NaN`, and `''` (empty string). These values evaluate to `false` in conditions.
   - Truthy values include anything other than the falsy values. For example, non-empty strings (like `'iPhone'`) evaluate to `true`.
7. **Property Binding**:
   The `ngClass` directive must be wrapped in square brackets to bind a TypeScript expression. This allows Angular to evaluate the expression dynamically and apply the appropriate class.
8. **No DOM Manipulation**:
   The `ngClass` directive does not manipulate the DOM by adding or removing elements. It only changes the appearance or behavior of an existing HTML element by toggling CSS classes.

Example 1: Dynamic Class Based on Boolean Condition

In this example, the class `highlight` is added to an element only if the condition is `true`.

```
<div [ngClass]="{ 'highlight': isHighlighted }">
  This is a dynamic class example.
</div>
```

In the component:

```typescript
isHighlighted: boolean = true;  // Set to `true` to highlight the div
```

- When `isHighlighted` is `true`, the class `highlight` is applied.
- When `isHighlighted` is `false`, the class `highlight` is removed.

Example 2: Using Multiple Classes with Conditions

Here, multiple classes are applied based on different conditions.

```html
<button [ngClass]="{
  'btn-primary': isPrimary,
  'btn-secondary': !isPrimary,
  'btn-large': isLarge,
  'btn-small': !isLarge
}">
  Submit
</button>
```

In the component:

```typescript
isPrimary: boolean = true;  // Determines primary or secondary style
isLarge: boolean = false;   // Determines large or small button size
```

- If `isPrimary` is `true`, the button will get the `btn-primary` class, otherwise the `btn-secondary` class.
- If `isLarge` is `true`, the button will get the `btn-large` class, otherwise the `btn-small` class.

Example 3: Toggling Multiple Classes with an Array

Instead of using an object, you can also use an array to add multiple classes conditionally.

```
<div [ngClass]="[isActive ? 'active' : '', isSpecial ? 'special' : '']">
  Toggling multiple classes
</div>
```

In the component:

typescript

```
isActive: boolean = true;  // If true, apply 'active'
isSpecial: boolean = false; // If true, apply 'special'
```

- If `isActive` is `true`, the class `active` is applied.
- If `isSpecial` is `true`, the class `special` is applied.

Example 4: Applying Class Based on Array Length

In this example, the class `has-items` is added if an array has items, otherwise, `empty-items` is applied.

```
<div [ngClass]="{ 'has-items': items.length > 0, 'empty-items': items.length === 0 }">
  List of items
</div>
```

In the component:

typescript

```
items: string[] = ['item1', 'item2'];  // Change this array to test
```

- If `items.length` is greater than 0, the class `has-items` is applied.
- If `items.length` is 0, the class `empty-items` is applied.

Example 5: Combining Static and Dynamic Classes

You can combine static classes with dynamic ones.

html Copy

```
<div class="container" [ngClass]="{ 'highlight': isHighlighted, 'error': isError }">
  This is a combined class example.
</div>
```

In the component:

typescript

```
isHighlighted: boolean = true;
isError: boolean = false;
```

- The `container` class is always applied statically.
- The `highlight` class is applied if `isHighlighted` is `true`.
- The `error` class is applied if `isError` is `true`.

Example 6: Using ngClass with Multiple Expressions (Object or Array)

You can mix objects and arrays in `ngClass` for more complex conditions.

html Copy code

```
<div [ngClass]="[ { 'highlight': isHighlighted }, { 'error': hasError }, 'default-class' ]">
  Dynamic classes with mix
</div>
```

In the component:

```typescript
isHighlighted: boolean = true;
hasError: boolean = false;
```

- The `highlight` class is applied if `isHighlighted` is `true`.
- The `error` class is applied if `hasError` is `true`.
- The `default-class` is applied regardless of any condition.

Example 7: Conditional CSS Classes for a List of Items

You can also use `ngClass` to conditionally style a list of items.

```
<ul>
  <li *ngFor="let item of items" [ngClass]="{ 'selected': item.selected, 'highlighted':
    {{ item.name }}
  </li>
</ul>
```

```
[ngClass]="{ 'selected': item.selected, 'highlighted': item.highlighted }">
```

In the component:

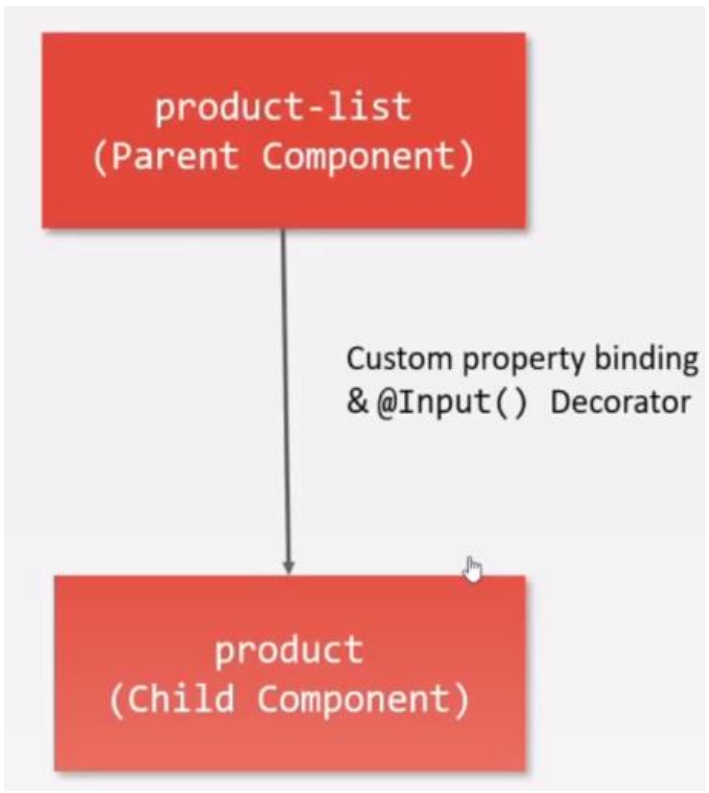
```typescript
items = [
  { name: 'Item 1', selected: true, highlighted: false },
  { name: 'Item 2', selected: false, highlighted: true },
  { name: 'Item 3', selected: false, highlighted: false }
];
```

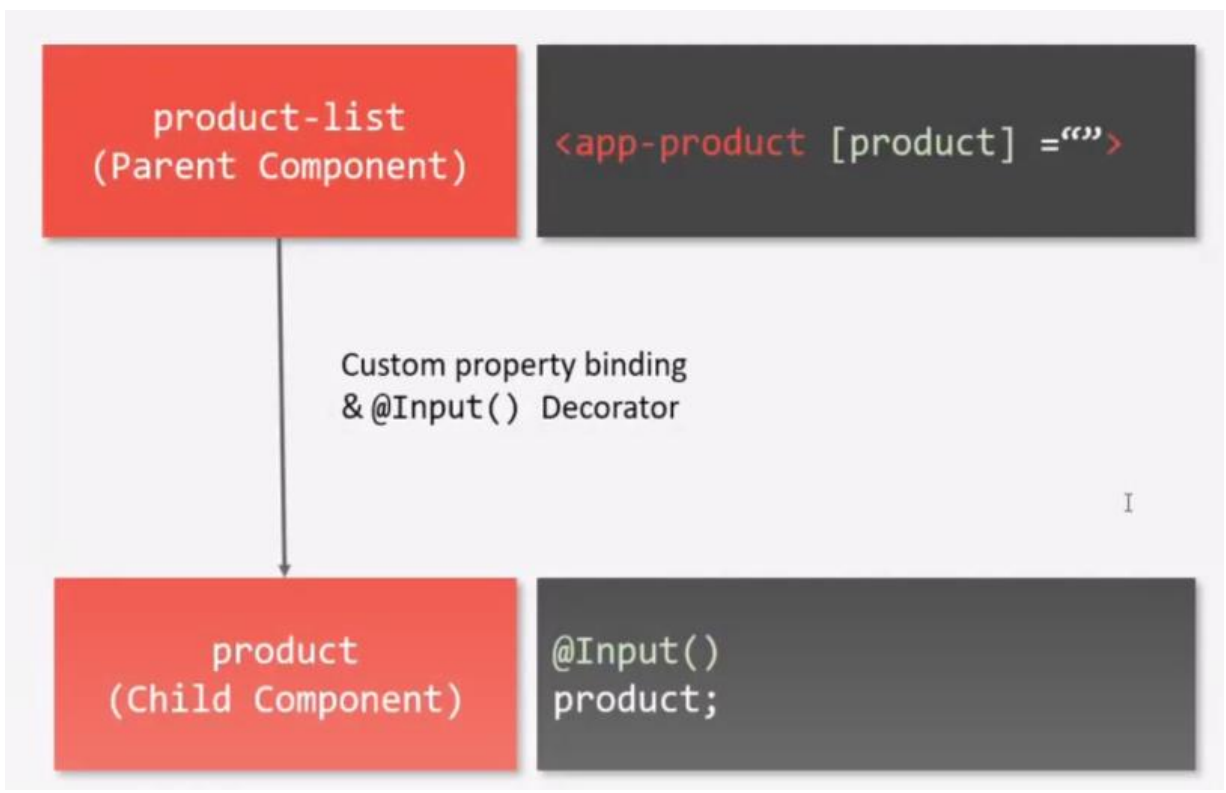
- Each list item has conditional classes applied based on its properties (`selected` , `highlighted`).

@Input & Custom Property Binding

Custom Property Binding Using `@Input` Decorator

Definition: Custom property binding allows passing data from a parent component to a child component using the `@Input` decorator. This enables the parent component to bind a value to a property in the child component, allowing for communication between components with a parent-child relationship.





Steps and Example:

1. **Create a Child Component:**
   - The child component (e.g., `ProductComponent`) will receive data from its parent component.
   - Create a property in the child component (e.g., `product`) to hold the incoming data.
   - Decorate the property with `@Input()` to allow it to accept data from the parent component.

```
@Input() product: any; // Declare property with @Input decorator
```

2. **Bind the Property in the Parent Component:**
   - In the parent component (e.g., `ProductListComponent`), bind the `product` property to the child component using square brackets `[]`.
   - Use the `*ngFor` directive to loop over an array (e.g., `products`) and pass the current element to the child component.

```
<app-product *ngFor="let prod of products" [product]="prod"></app-product>
```

3. **Use the Bound Data in the Child Component:**
   - In the child component's template, access the `product` property to display data.

```
html

<div>{{ product.name }}</div>
```

4. **Define Type for the Bound Property:**
   - Optionally, define the type of the property (e.g., an anonymous object with properties like `id`, `name`, etc.) to ensure proper type checking.

```
typescript                                                    Copy code

@Input() product: { id: number, name: string, price: number };
```

5. **Handling Strict Type Checking (Optional):**
   - If there are strict type checking errors, you can disable the `strict` option in `tsconfig.json` or ensure the property is correctly initialized.

```
json                                                          Copy code

"strict": false
```

Key Concepts:

- **@Input Decorator:** Used to define a custom property in the child component, which can be bound by the parent component.
- **Property Binding:** Achieved using square brackets `[]` to pass data from parent to child.
- `*ngFor` **Directive:** Loops over an array in the parent component and binds each item to the child component.

Summary:

- The parent component passes data to the child component through a custom property binding.
- The child component receives the data using `@Input()` and can display it or manipulate it as needed.

Example Scenario:

Let's say you have a parent component that needs to pass a string to a child component to display it.

Step-by-Step:

1. **Child Component (e.g., `MessageComponent`):**
    - In the child component, we create a property (`message`) to hold the data passed from the parent.
    - We decorate this property with `@Input()` to let Angular know that the property can receive data from the parent.

child.component.ts:

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

@Component({
  selector: 'app-message',
  template: '<p>{{ message }}</p>',
})
export class MessageComponent {
  @Input() message: string = '';  // Receiving data from parent
}
```

child.component.html:

```html
<!-- Displaying the message passed from the parent -->
<p>{{ message }}</p>
```

2. **Parent Component (e.g., `AppComponent`):**
   - In the parent component, you pass the data (e.g., a string) to the `MessageComponent` using property binding with square brackets `[]`.

parent.component.html:

```html
<app-message [message]="parentMessage"></app-message>
<!-- Binding parentMessage to message in child -->
```

parent.component.ts:

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  parentMessage: string = 'Hello from Parent Component!';
}
```

Explanation:

- **Parent to Child Communication:** The parent component passes the string `parentMessage` to the child component using `[message]="parentMessage"`.
- **@Input() Decorator:** The `@Input()` decorator in the child component allows the `message` property to receive the value passed from the parent.
- **Property Binding:** The square brackets `[]` are used for binding the parent property (`parentMessage`) to the child property (`message`).

Result:

When you run the application, the child component will display:

```csharp

Hello from Parent Component!
```

This example is simpler because it involves just one string being passed from the parent to the child component using the `@Input()` decorator and property binding.

Notes on Custom Property Binding and `@Input` Decorator

Custom Property Binding:

- Custom property binding allows you to bind properties of a component class to a TypeScript expression. This enables passing data from a parent component to a child component.

`@Input` Decorator:

- The `@Input()` decorator is used in a child component to receive data from a parent component.
- Properties in the child component need to be decorated with `@Input()` for the parent to pass data to them.

Example Overview:

- In the example, we have two components: `ProductListComponent` (parent) and `FilterComponent` (child).
- The `FilterComponent` contains radio buttons for filtering products by their stock status (in stock, out of stock).

Steps and Explanation:

1. **Creating the Filter Component:**
    - The `FilterComponent` is created using Angular's `ng generate component` command.
    - This component displays radio buttons for filtering products based on stock status.
2. **Using the Filter Component in Parent:**
    - The `FilterComponent` is used inside the `ProductListComponent` template using its selector (`app-filter`).
3. **Passing Data from Parent to Child:**
    - The parent component, `ProductListComponent` , contains an array of products.
    - Data such as the total product count, in-stock count, and out-of-stock count is passed to the child component via custom property binding.
4. **Binding Data Using `@Input()` :**
    - In the `FilterComponent` , properties like `all` , `inStock` , and `outOfStock` are decorated with `@Input()` .
    - These properties are used in the `ProductListComponent` template as attributes with property binding, such as `[all]="totalProdCount"` , where `totalProdCount` is calculated in the parent component.
5. **Calculating Product Counts:**
    - In `ProductListComponent` , properties such as `totalProdCount` , `totalProdInStock` , and `totalProdOutOfStock` are created.
    - These properties are derived from the `products` array. For example, the in-stock count is determined using the `filter` method with a condition (`p.isInInventory`).

6. **Binding Data to Child Component:**
    - The calculated values are passed to the child component's `@Input` properties using property binding.
    - The parent component binds `totalProdCount` to `all`, `totalProdInStock` to `inStock`, and `totalProdOutOfStock` to `outOfStock` in the child component.
7. **Rendering Data in Child Component:**
    - The values received from the parent (e.g., `all`, `inStock`, `outOfStock`) are displayed in the `FilterComponent` template using string interpolation.

Key Concepts:

- **Custom Property Binding**: A method for binding a component's property to a value from a TypeScript expression, typically from the parent to the child.
- `@Input` **Decorator**: This decorator is used in the child component to receive data from the parent component.
- **Property Binding Syntax**: The `[property]="expression"` syntax is used to pass data from the parent to the child.

Conclusion:

- Custom property binding enables dynamic interaction between parent and child components.
- The `@Input` decorator is essential for making properties in a child component receptive to data passed from a parent.

`@Output` & Custom Event Binding

Notes on `@Output` Decorator

The `@Output` decorator in Angular is used to allow data to flow from a **child component** to a **parent component**. It enables the child component to send information or trigger actions in the parent component.

Key Concepts:

- `@Output` **Decorator**: It is used in the child component to emit an event to the parent component. The parent listens to this event to react or handle the data.
- **EventEmitter**: The `@Output` decorator is typically used with an instance of `EventEmitter`, which allows the child component to emit an event to the parent component.

Syntax:

```ts
@Output() eventName = new EventEmitter<Type>();
```

- `eventName` : The name of the event the child component will emit.
- `EventEmitter` : A class that is used to emit events to the parent component.
- `Type` : The type of data being passed (optional).

Example Workflow:

- The child component emits an event when a certain action occurs (like a button click).
- The parent component listens for that event and reacts accordingly.

Example:

1. Child Component (`child.component.ts`):

In the child component, we create an `EventEmitter` property decorated with `@Output` .

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

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

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

- Here, `messageEvent` is the output property that emits a string message when the button is clicked.
- The `sendMessage()` method triggers the `emit()` method on the `EventEmitter` to send the message to the parent.

2. Parent Component (`parent.component.ts`):

The parent component listens for the event emitted by the child component.

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

@Component({
  selector: 'app-parent',
  template: `<app-child (messageEvent)="receiveMessage($event)"></app-child>
             <p>{{ receivedMessage }}</p>`
})
export class ParentComponent {
  receivedMessage: string;

  receiveMessage(message: string) {
    this.receivedMessage = message;
  }
```

- In the parent component, we use the child component's selector `<app-child>`.
- The parent listens for the `messageEvent` using the `(messageEvent)` syntax.
- When the event is emitted, the `receiveMessage()` method is called, and the message is stored in the `receivedMessage` property.
- The message is displayed in the parent component's template.

3. **How it Works:**

- The child component emits an event when the button is clicked.
- The parent component listens for this event and reacts by updating its own property (`receivedMessage`), which is displayed in the template.

Example with Data Type (e.g., Number):

1. **Child Component**:

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

@Component({
  selector: 'app-child',
  template: `<button (click)="sendNumber()">Send Number</button>`
})
```

```
export class ChildComponent {
  @Output() numberEvent = new EventEmitter<number>();

  sendNumber() {
    this.numberEvent.emit(42);  // Sending a number to the parent
  }
}
```

2. Parent Component:

ts

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

@Component({
  selector: 'app-parent',
  template: `<app-child (numberEvent)="receiveNumber($event)"></app-child>
             <p>{{ receivedNumber }}</p>`
})
export class ParentComponent {
  receivedNumber: number;

  receiveNumber(number: number) {
    this.receivedNumber = number;
  }
}
```

- The child emits a number `42`, and the parent component listens for it. Once received, it updates the `receivedNumber` property.

Conclusion:

- The `@Output` decorator is crucial for enabling communication from the child component to the parent component.
- It uses the `EventEmitter` class to emit events, and the parent component listens for those events to react or update its state.

Non-Related Component Communication

Notes on Passing Data Between Non-Related (Sibling) Components in Angular

When two components are not related (i.e., they are siblings or have no direct parent-child relationship), Angular provides ways to communicate between them using a combination of **custom property binding** and **custom event binding**. Here's a step-by-step guide to achieve data passing between sibling or non-related components:

1. Communication Using Custom Event Binding and Property Binding

- **Custom Event Binding** is used to emit events from the child component.
- **Custom Property Binding** allows data to be passed from the parent to the child component.

In the case of non-related components, these concepts can be combined through the **container component**, which acts as an intermediary between the sibling components.

Scenario:

In the example provided, there are two sibling components:

- **Search Component**: Takes user input (search text).
- **Product List Component**: Displays products based on the search text.

The goal is to filter the products dynamically based on the search text input from the Search Component, without these components having a parent-child relationship.

Steps to Achieve Communication:

2. Search Component (Child) - Emit Event Using `@Output`

- In the **Search Component**, when the user types in the search box, the search text needs to be passed to the **Product List Component**.
- To pass data to the parent component (Container Component), the **Search Component** emits an event using `@Output` and `EventEmitter`.

Code in Search Component (`search.component.ts`):

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

@Component({
  selector: 'app-search',
  templateUrl: './search.component.html',
})
export class SearchComponent {
  @Output() searchTextChanged = new EventEmitter<string>();
  searchText: string = '';

  onSearchTextChanged() {
    this.searchTextChanged.emit(this.searchText); // Emit the search text
  }
```

- `@Output()` : The `searchTextChanged` event is emitted when the user types in the search box.
- **EventEmitter**: It emits the value of `searchText` to the parent.

HTML for Search Component (`search.component.html`):

```
<input type="text" [(ngModel)]="searchText" (input)="onSearchTextChanged()" />
```

- The input field is bound to the `searchText` property using two-way data binding `[(ngModel)]` .
- The `input` event triggers the `onSearchTextChanged()` method to emit the value.

3. **Container Component (Parent) - Listen to the Event and Pass Data**

- The **Container Component** listens for the emitted event and receives the search text value.
- It stores the received value in a property (`searchText`) and passes it to the **Product List Component** via **Property Binding**.

Code in Container Component (`container.component.ts`):

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

@Component({
  selector: 'app-container',
  templateUrl: './container.component.html',
})
export class ContainerComponent {
  searchText: string = '';

  onSearchTextChanged(value: string) {
    this.searchText = value;  // Store the search text received from child
  }
}
```

HTML for Container Component (`container.component.html`):

```html
<app-search (searchTextChanged)="onSearchTextChanged($event)"></app-search>
<app-product-list [searchText]="searchText"></app-product-list>
```

- The `(searchTextChanged)` event is bound to the `onSearchTextChanged($event)` method.
- The `[searchText]` property binding passes the received value (`searchText`) to the **Product List Component**.

4. Product List Component (Sibling) - Receive Data Using `@Input`

- The **Product List Component** receives the filtered `searchText` value via `@Input()` .

Code in Product List Component (`product-list.component.ts`):

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

@Component({
  selector: 'app-product-list',
  templateUrl: './product-list.component.html',
})
```

```ts
export class ProductListComponent {
  @Input() searchText: string = '';
  products = [
    { name: 'Nike Air Max' },
    { name: 'Adidas Running Shoes' },
    { name: 'Nike Pro' },
    { name: 'Apple Watch' },
  ];
```

```ts
 get filteredProducts() {
   if (this.searchText.trim() === '') {
     return this.products;  // Show all products if search text is empty
   }
   return this.products.filter(product =>
     product.name.toLowerCase().includes(this.searchText.toLowerCase())
   );
 }
```

- The `searchText` property is decorated with `@Input()`, allowing it to receive the value passed from the **Container Component**.

HTML for Product List Component (`product-list.component.html`):

html Copy co

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

- The `*ngFor` directive is used to display the list of filtered products based on the search text.

5. Summary of the Process:

- **Search Component** emits an event (`searchTextChanged`) with the typed text using `@Output()` .
- **Container Component** listens to the event and passes the data to the **Product List Component** via property binding (`[searchText]`).
- **Product List Component** filters and displays products based on the received search text.

Conclusion:

By combining **custom event binding** (using `@Output` and `EventEmitter`) and **custom property binding** (using `@Input`), we can pass data between sibling components or components with no direct relationship. In this scenario, the **Container Component** acts as an intermediary to facilitate communication between the **Search Component** and the **Product List Component**.

Template Reference Variable

A template reference variable is a variable which stores a reference to a DOM element, Component or Directive on which it is used.

Angular - Template Reference Variable

Definition:

A **template reference variable** in Angular is a variable that stores a reference to a DOM element, a component, or a directive on which it is declared. It can be used to access properties and methods of the associated element or component.

Example:

1. **Without Template Reference Variable (default data binding)**:
    - In a search input scenario, the search functionality triggers on every keystroke via two-way data binding. Each character typed updates the search immediately.
2. **Implementing Template Reference Variable**:
    - Now, we want the search to happen only when the user clicks a search button.
3. **Steps to Use Template Reference Variable**:
    - **Step 1**: Remove two-way data binding and input event from the search input box.
    - **Step 2**: Add a template reference variable to the search input box like so:

```html
<input #searchInput type="text">
```

- **Step 3**: Bind the search button's `click` event to a method that handles the search:

```html
<button (click)="updateSearchText(searchInput)">Search</button>
```

- **Step 4**: In the component's method, use the template reference variable to access the input box's value:

```typescript
updateSearchText(inputEl: HTMLInputElement) {
  this.searchText = inputEl.value;
}
```

4. **How it Works**:
    - The template reference variable `#searchInput` holds a reference to the input element. When the search button is clicked, the method `updateSearchText()` is called with the reference to the input, allowing us to access the input value and update the component's `searchText` property.

Key Concepts:

- **Creating a Template Reference Variable**: Use `#` followed by a variable name (e.g., `#searchInput`).
- **Using the Reference**: Pass the reference to methods or use it directly in the template for operations on the DOM element.
- **DOM Manipulation**: The template reference allows you to interact directly with elements in the DOM, like accessing input values or controlling other HTML elements.

Reference Variable on Components

Component Template Reference Variable in Angular

Definition:

A **template reference variable** is a variable in Angular templates that allows access to a DOM element, component, or directive. It enables interaction with the properties and methods of a component instance directly in the template without needing explicit data binding. Template reference variables are declared using the `#` symbol followed by a variable name.

Key Points:

- Template reference variables are typically used to access a DOM element or a component instance.
- They are defined in the template, and the value of the variable is a reference to the DOM element or component.
- You can use these variables to call methods or access properties of the component instance directly.

Example: Accessing a DOM Element

```
<!-- HTML Template -->
<input #myInput type="text" placeholder="Enter text here">
<button (click)="logInput(myInput.value)">Log Input</button>

<!-- TypeScript -->
logInput(value: string) {
  console.log(value);
}
```

Here, `#myInput` is a template reference variable. When the button is clicked, the `logInput()` method is called with the value from the input field, accessed through the reference variable.

Example: Accessing a Component Instance

```
<!-- product-list.component.html -->
<app-product-list #productList></app-product-list>
<button (click)="productList.showDetails()">Show Details</button>
```

In this example, `#productList` is a template reference variable that refers to an instance of the `ProductListComponent`. The button click directly calls the `showDetails()` method of that component.

Example: Conditional Rendering using Template Reference

Example: Conditional Rendering using Template Reference

html Copy code

```
<app-product-list #productList></app-product-list>
<app-product-detail *ngIf="productList.selectedProduct" [product]="productList.selectedPro
```

Here, `#productList` is used to access the `selectedProduct` property of `ProductListComponent` and pass it to the `ProductDetailComponent` for conditional rendering.

Conclusion:

Template reference variables are a powerful way in Angular to access DOM elements or component instances directly in the template. They simplify interaction between the template and the component, making it easier to manage component states and DOM elements.

Angular Template Reference Variables - Definition and Example

Definition:

- A **template reference variable** in Angular is a variable that holds a reference to a DOM element, a component, or a directive. It is declared using the `#` symbol followed by a name, and it can be used to access that element or component directly within the template.

Key Points:

1. **Usage**: Template reference variables can be used to reference HTML elements, Angular components, or directives.
2. **Scope**: They are only available within the template where they are declared.
3. **Example Use Cases**:
    - Manipulating DOM elements directly.
    - Accessing public properties or methods of a component.

Example: Accessing a Component Using Template Reference Variable

In this example, we will use a template reference variable to select a product when a user clicks on it and display its details.

1. **Creating a Product Class**:

```
// product.ts
export class Product {
  id: number;
  name: string;
  price: number;
}
```

2. **Creating `ProductListComponent`**:
    - The `ProductListComponent` will allow selecting a product, and we will store the selected product in the `selectedProduct` property.

```ts
// product-list.component.ts
import { Product } from '../models/product';

export class ProductListComponent {
  selectedProduct: Product;

  products = [
    { id: 1, name: 'Product 1', price: 100 },
    { id: 2, name: 'Product 2', price: 200 },
  ];

  selectProduct(product: Product) {
    this.selectedProduct = product;
  }
}
```

3. **Using Template Reference Variable in HTML:**

```html
<!-- product-list.component.html -->
<div *ngFor="let prod of products" (click)="selectProduct(prod)">
  {{ prod.name }}
</div>
```

4. **Creating `ProductDetailComponent`:**

   - This component will display details of the selected product.

```ts
// product-detail.component.ts
import { Product } from '../models/product';

export class ProductDetailComponent {
  product: Product;
}
```

5. **Using Template Reference Variable in the Container:**

   - Accessing the selected product from `ProductListComponent` and passing it to `ProductDetailComponent`.

```html
<!-- container.component.html -->
<app-product-list #productListComponent></app-product-list>
<app-product-detail *ngIf="productListComponent.selectedProduct"
                    [product]="productListComponent.selectedProduct">
</app-product-detail>
```

Summary:

- **Template reference variables** allow easy access to components and their properties within templates. By binding it with a component, we can reference component instances and access properties like `selectedProduct` for rendering in another component (e.g., product details).

1. `ViewChild` Decorator (Angular):

- **Definition**: The `ViewChild` decorator in Angular allows you to access and manipulate elements or child components in your component's template. It is used to fetch references to DOM elements or Angular components and work with them within your TypeScript code.
- **Use Case**: It is typically used when you need to programmatically control or access a DOM element, its properties, or methods of child components after the view initialization.

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

@Component({
  selector: 'app-example',
  template: `
    <input #txtName type="text" value="Initial Value" />
  `,
})
```

```
export class ExampleComponent {
  @ViewChild('txtName') txtName!: ElementRef;

  ngAfterViewInit() {
    console.log(this.txtName.nativeElement.value); // Logs "Initial Value"
    this.txtName.nativeElement.value = 'Updated Value'; // Updates the value to "Updated
```

2. `ElementRef` (Angular):

- **Definition**: `ElementRef` is a wrapper around a native DOM element in Angular. It gives direct access to the native element in the view. It is commonly used with `ViewChild` to interact with native HTML elements in the component's template.
- **Use Case**: Modify the DOM element's properties, styles, or even bind events programmatically.

```
@ViewChild('txtName') txtName!: ElementRef;


ngAfterViewInit() {
  this.txtName.nativeElement.style.color = 'blue'; // Changes text color to blue
}
```

3. **Lifecycle Hook:** `ngAfterViewInit`:

- **Definition**: The `ngAfterViewInit` lifecycle hook in Angular is triggered once the view of the component and its child views are fully initialized. It's used for operations that need the view to be rendered before accessing or interacting with elements.
- **Use Case**: Access DOM elements or child components once the view has been initialized.

Example:

```
export class ExampleComponent implements AfterViewInit {
  @ViewChild('txtName') txtName!: ElementRef;

  ngAfterViewInit() {
    // View is initialized, safe to access DOM elements or child components
    console.log(this.txtName.nativeElement.value);
```

4. **Accessing Child Components with** `ViewChild`:

- **Definition**: Using `ViewChild` with components allows you to interact with child components' methods and properties from a parent component.
- **Use Case**: Invoke methods or access properties from child components programmatically.

Example:

```typescript
@Component({
  selector: 'app-child',
  template: `<p>Child Component</p>`
})
export class ChildComponent {
  sayHello() {
    console.log('Hello from Child Component');
  }
}
```

```typescript
@Component({
  selector: 'app-parent',
  template: `<app-child></app-child>`
})
export class ParentComponent {
  @ViewChild(ChildComponent) child!: ChildComponent;

  ngAfterViewInit() {
    this.child.sayHello(); // Calls the sayHello() method in ChildComponent
  }
}
```

5. Using `Document.getElementById()` vs `ViewChild`:

- **Definition**: `Document.getElementById()` is a standard JavaScript method used to access a DOM element by its ID, while `ViewChild` is an Angular feature for accessing template elements or child components directly.
- **Difference**: In Angular, it is recommended to use `ViewChild` instead of `getElementById()` because it integrates better with Angular's change detection and lifecycle.

Example with `getElementById()`:

typescript

```typescript
const element = document.getElementById('txtName');
if (element) {
  element.value = 'New Value';
}
```

Example with `ViewChild`:

typescript

```typescript
@ViewChild('txtName') txtName!: ElementRef;

ngAfterViewInit() {
  this.txtName.nativeElement.value = 'New Value';
}
```

6. Difference Between HTML Elements and Components in `ViewChild`:

- **Definition**: `ViewChild` can be used both with HTML elements and Angular components. When used with HTML elements, it accesses the native DOM element. When used with Angular components, it accesses the component instance.
- **Use Case**: Use the native DOM element to change styles or values, and use the component instance to call methods or access component properties.

Example with HTML Element:

typescript Copy code

```typescript
@ViewChild('txtName') txtName!: ElementRef;

ngAfterViewInit() {
  this.txtName.nativeElement.style.backgroundColor = 'yellow'; // Changes background color
}
```

Example with Angular Component:

typescript

```typescript
@ViewChild(ChildComponent) child!: ChildComponent;

ngAfterViewInit() {
  this.child.someMethod(); // Calls a method in ChildComponent
```

The ViewChild decorator is used to query and get a reference of the DOM element in the component. It returns the first matching element.

```
Property decorator that configures a view query. The change detector looks for the first
element or the directive matching the selector in the view DOM. If the view DOM
changes, and a new child matches the selector, the property is updated.

View queries are set before the ngAfterViewInit callback is called.

Metadata Properties:
@ViewChild('inputRef') searchInput: any;
```

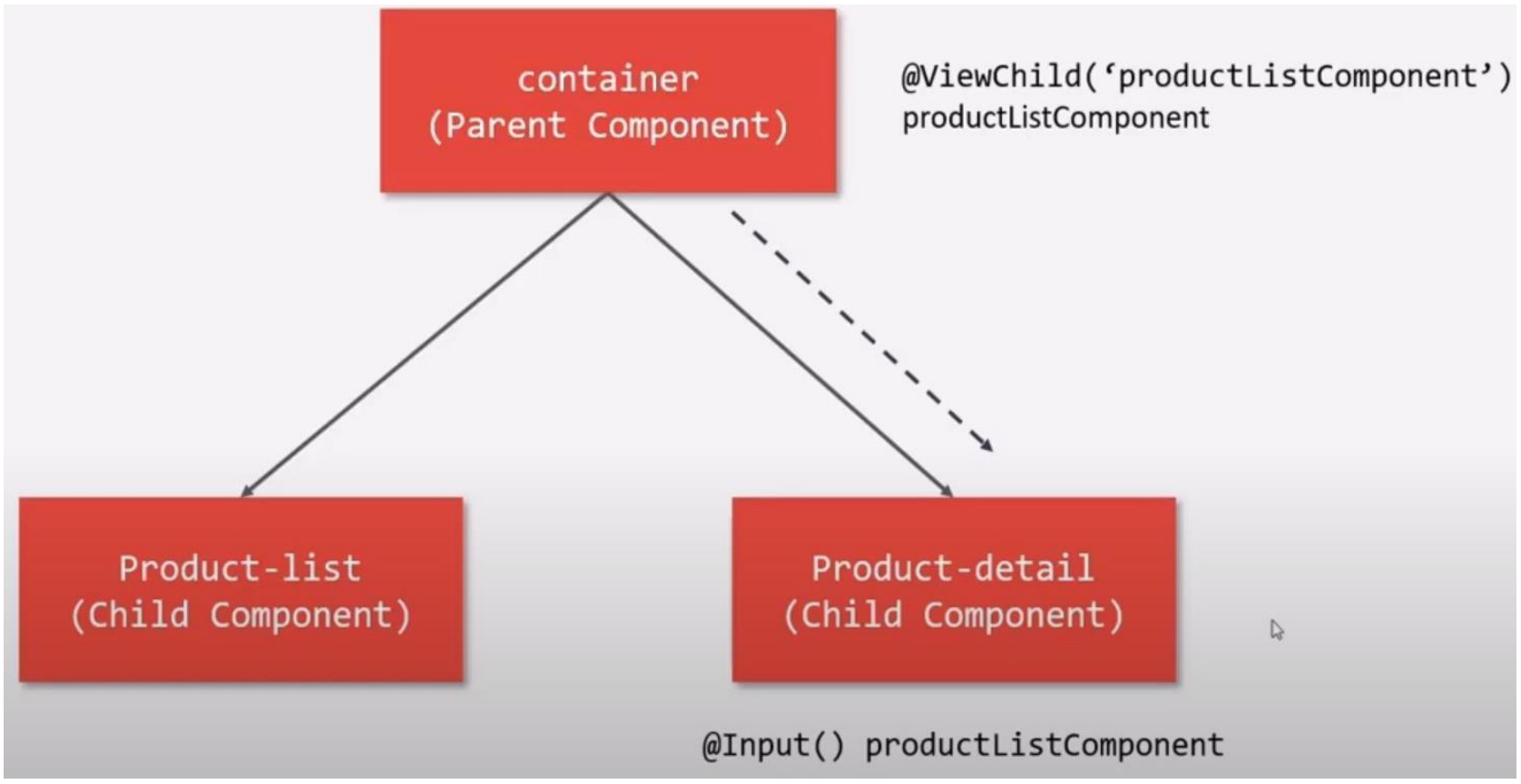
```
import { Component, ElementRef, ViewChild } from '@angular/core';

@Component({
  selector: 'app-view-child',
  templateUrl: './view-child.component.html',
  styleUrls: ['./view-child.component.css']
})
export class ViewChildComponent {
  searchText = '';

  @ViewChild('inputRef') searchInput!: ElementRef;

  searchedResult() {
    this.searchText = this.searchInput.nativeElement.value;
  }
}
```

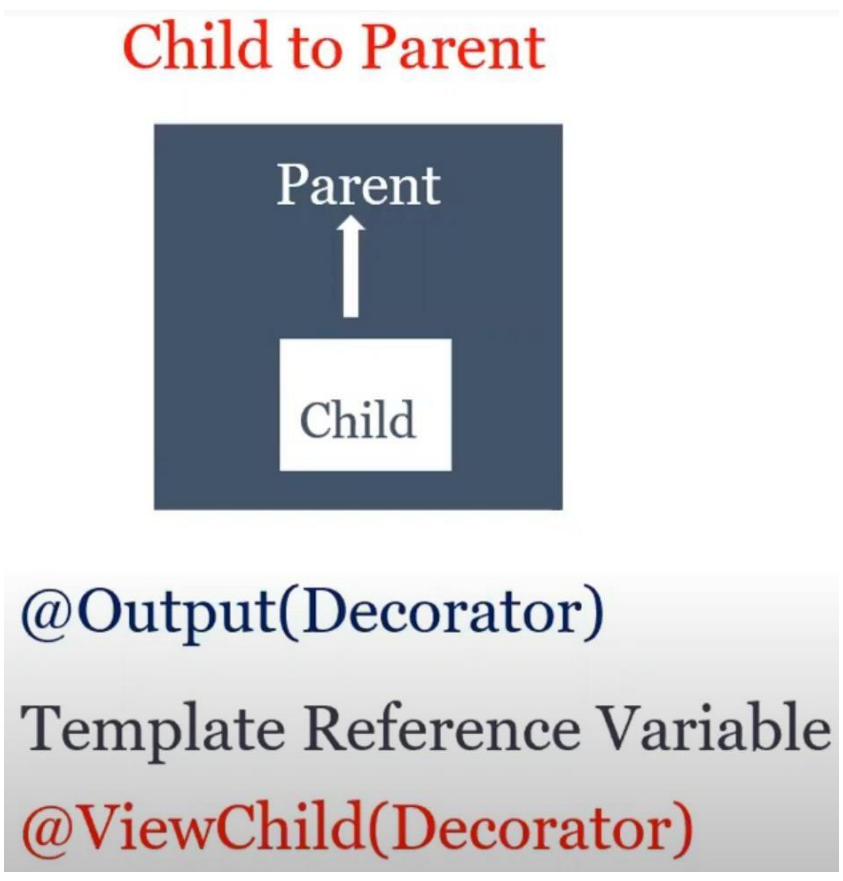
view-child.component.html U

mplete_Course > angular-ekart > src > app > view-child > view-child.component

```
Go to component
<div class="container">
  <input type="text" #inputRef>
  <button (click)="searchedResult()">Click me</button>
</div>

<div>
  <p>Search results {{searchText}}</p>
</div>
```

```typescript
//Optional 2nd argument of @ViewChild()
//1. read: Use it to read the different token from the queried elements.
//2. static: Determines when the query is resolved.
//        True is when the view is initialized (before the first change detection) for the first time.
//        False if you want it to be resolved after every change detection
@ViewChild('searchInput', {static: true}) searchInputEl: ElementRef;
```





@view child decorator

- `import { ViewChild } from '@angular/core';`
- `import{ChildComponent} from './child/child.component'`
- `@ViewChild(ChildComponent) child`

Header component

```
export class HeaderComponent {

  data = 'data of view child';

  passToParent() {
    return "Hi i am Child of view";
  }

}
```

I should have the child components selector in html page of parent.

parent.component.html U

Complete_Course > angular-ekart > src > app > view-child > parent

Go to component

```
1  <!-- child component -->
2
3  <app-header #val></app-header>
```

The view child reference on component works after ngAfterViewInit

```typescript
export class ParentComponent implements AfterViewInit {

  @ViewChild('val') value?: HeaderComponent;


  ngAfterViewInit() {
    // Safely access the child component's properties and methods
    console.log(this.value?.data);
    // Output the 'data' from the child component
    console.log(this.value?.passToParent());
    // Call the 'passToParent' method from the child component
  }
```

```typescript
import { Component, ViewChild, ElementRef } from '@angular/core';

@Component({
  selector: 'app-example',
  template: `
    <input #txtName type="text" value="Initial Value" />
  `,
})
```

@ViewChildren() in Angular

The ViewChildren decorator is used to get a reference to the list of DOM element from the view template in the component class. It returns all the matching element.

1. ViewChild Decorator

- **Definition**: `ViewChild` allows access to a reference of a single DOM element, component, or directive within the template. It only returns the first matching element.
- **Usage**: This is commonly used when you need to access an element directly in the component class.
- **Example**:

```typescript
@ViewChild('inputEl') inputElement: ElementRef;
```

This example selects the first DOM element with the template reference variable `#inputEl`.

- **Important**: It will only select the **first** element, even if there are multiple elements with the same selector.

2. ViewChildren Decorator

- **Definition**: `ViewChildren` is similar to `ViewChild`, but it is used when you want to retrieve **multiple** elements that match the selector.
- **Usage**: It returns a list of matching DOM elements, components, or directives in the form of a `QueryList`.
- **Example**:

typescript Copy code

```typescript
@ViewChildren('inputEl') inputElements: QueryList<ElementRef>;
```

This example will return a `QueryList` containing all DOM elements with the template reference `#inputEl`.

- **Difference from ViewChild**:
  - `ViewChild` returns a **single** DOM element.
  - `ViewChildren` returns a **list** of DOM elements as a `QueryList`.

```html
<div class="container">
  <input type="text" placeholder="Enter first name" #inputEl>
  <input type="text" placeholder="Enter middle name" #inputEl>
  <input type="text" placeholder="Enter last name" #inputEl>

  <button (click)="displayName()">Show</button>

  <p>{{fullName}}</p>
</div>
```

```typescript
export class ViewChildrenComponent {

  fullName = '';

  @ViewChildren('inputEl') personName?: QueryList<ElementRef>;

  displayName() {
    this.personName?.forEach((val) => {
      this.fullName += val.nativeElement.value + ' ';
    })
    this.fullName.trim();
  }
}
```

3. ElementRef

- **Definition**: `ElementRef` is a wrapper around a native DOM element in Angular. It allows direct interaction with the underlying HTML element.
- **Usage**: It provides access to the native DOM element via the `nativeElement` property.
- **Example**:

typescript — Copy code

```typescript
console.log(this.inputElement.nativeElement.value);
```

This accesses the value of an input element.

4. Template Reference Variable

- **Definition**: A template reference variable is declared in an Angular template and provides a reference to a DOM element or a component.
- **Example**:

html — Copy code

```html
<input #inputEl>
```

The `#inputEl` is a reference to the input element that can be used in the component class via `ViewChild` or `ViewChildren`.

5. QueryList

- **Definition**: A `QueryList` is a class that stores a list of elements retrieved by `ViewChildren`. It has methods like `forEach` to iterate over the elements.
- **Usage**: This is often used to loop over multiple DOM elements.
- **Example**:

```
this.inputElements.forEach(input => {
  console.log(input.nativeElement.value);
});
```

6. ngOnInit Lifecycle Hook

- **Definition**: `ngOnInit` is a lifecycle hook in Angular that is called once the component is initialized. It's commonly used for initialization logic after component properties are set.
- **Usage**: When you need to access or initialize data after the component's properties are set.
- **Important**: The `ViewChildren` properties are not available in `ngOnInit`, because they are resolved **after** the view initialization and change detection.

7. Change Detection Cycle

- **Definition**: In Angular, change detection is the mechanism that checks if the view needs to be updated with any new data and re-renders accordingly.
- **Impact on ViewChildren**: Properties decorated with `ViewChildren` are not resolved until after the change detection cycle is run. Therefore, attempting to access them in `ngOnInit` will result in an error.

8. Event Binding

- **Definition**: In Angular, event binding allows you to listen for user events such as clicks.
- **Example**:

```html
<button (click)="show()">Show</button>
```

The above binds the `click` event of the button to the `show` method in the component class.

9. String Manipulation with `trim()`

- **Definition**: The `trim()` method removes any whitespace from the beginning and end of a string.
- **Example**:

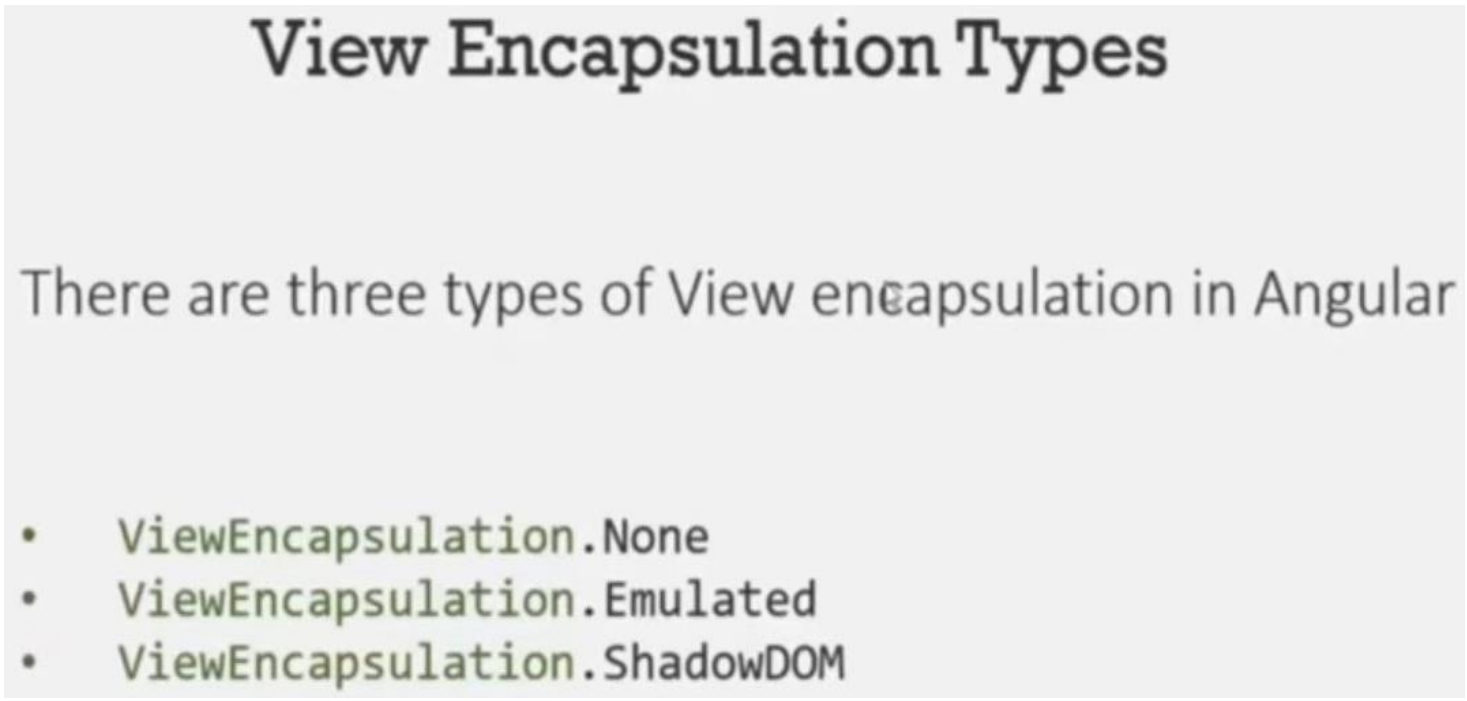
```typescript
let name = "John ";
name = name.trim(); // "John"
```

What is View Encapsulation?

Encapsulation means hiding data and behavior from outside world.

View Encapsulation in Angular

Definition: View encapsulation in Angular ensures that the CSS styles applied to one component do not affect other components. It allows each component to have its own isolated styles, preventing global CSS leaks.

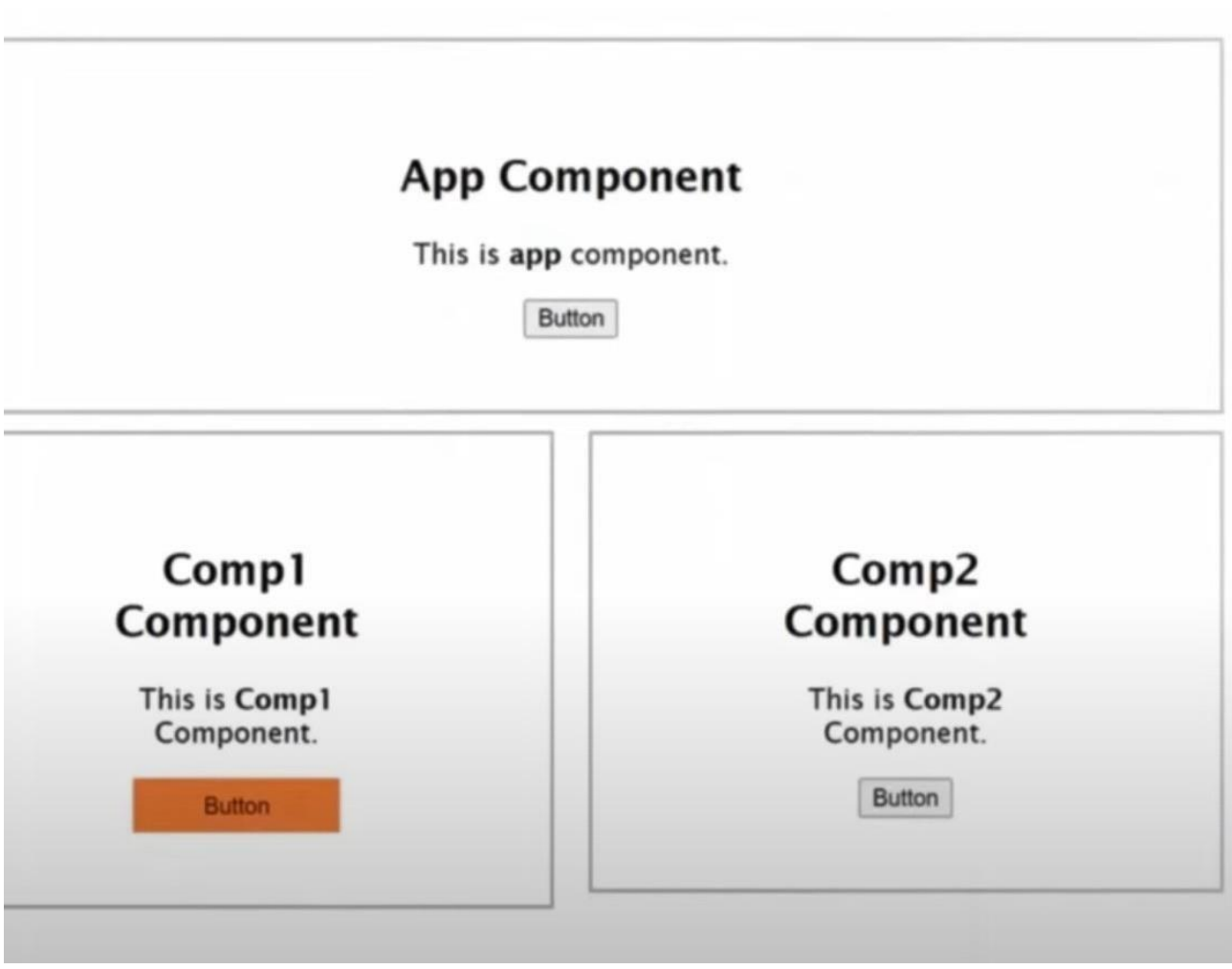


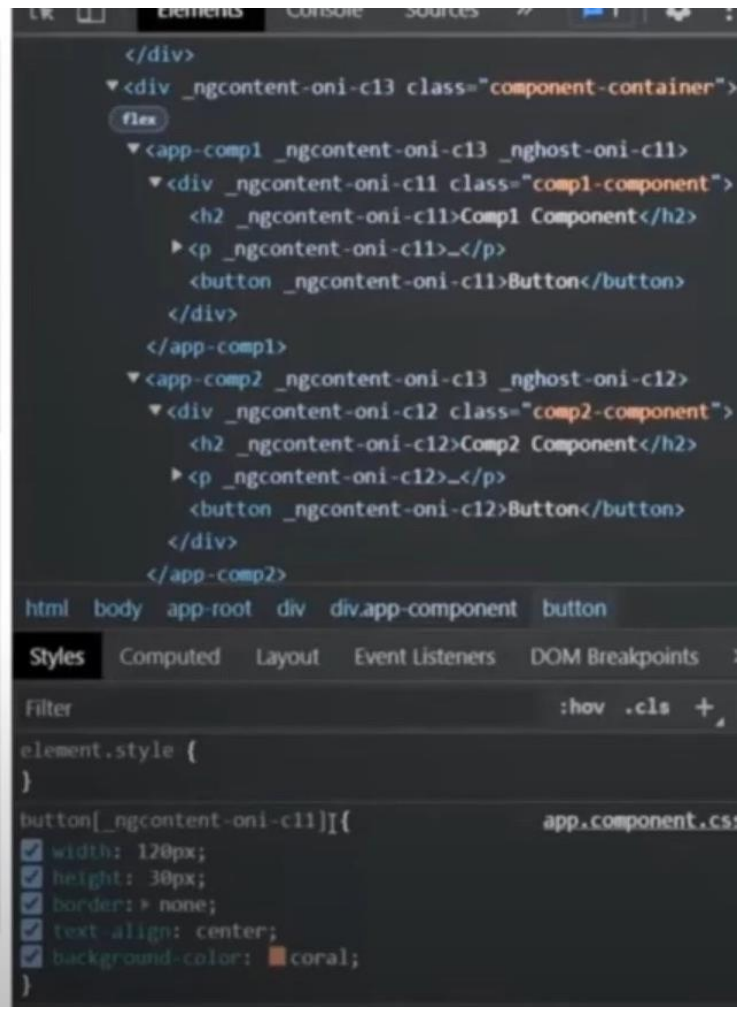
Types of View Encapsulation:

1. **Emulated (Default):**
   - **Behavior:** Angular adds unique attributes to each component's HTML elements and styles, ensuring styles only affect that component.
   - **Example:** If a button style is defined in one component's CSS, it only applies to that component's button.
   - **How it works:** Attributes like `_ngcontent-*` are added to both the component's elements and styles to isolate styles within the component.

View Encapsulation :

Emulated ⇒ some hidden properties will be applied to dom.



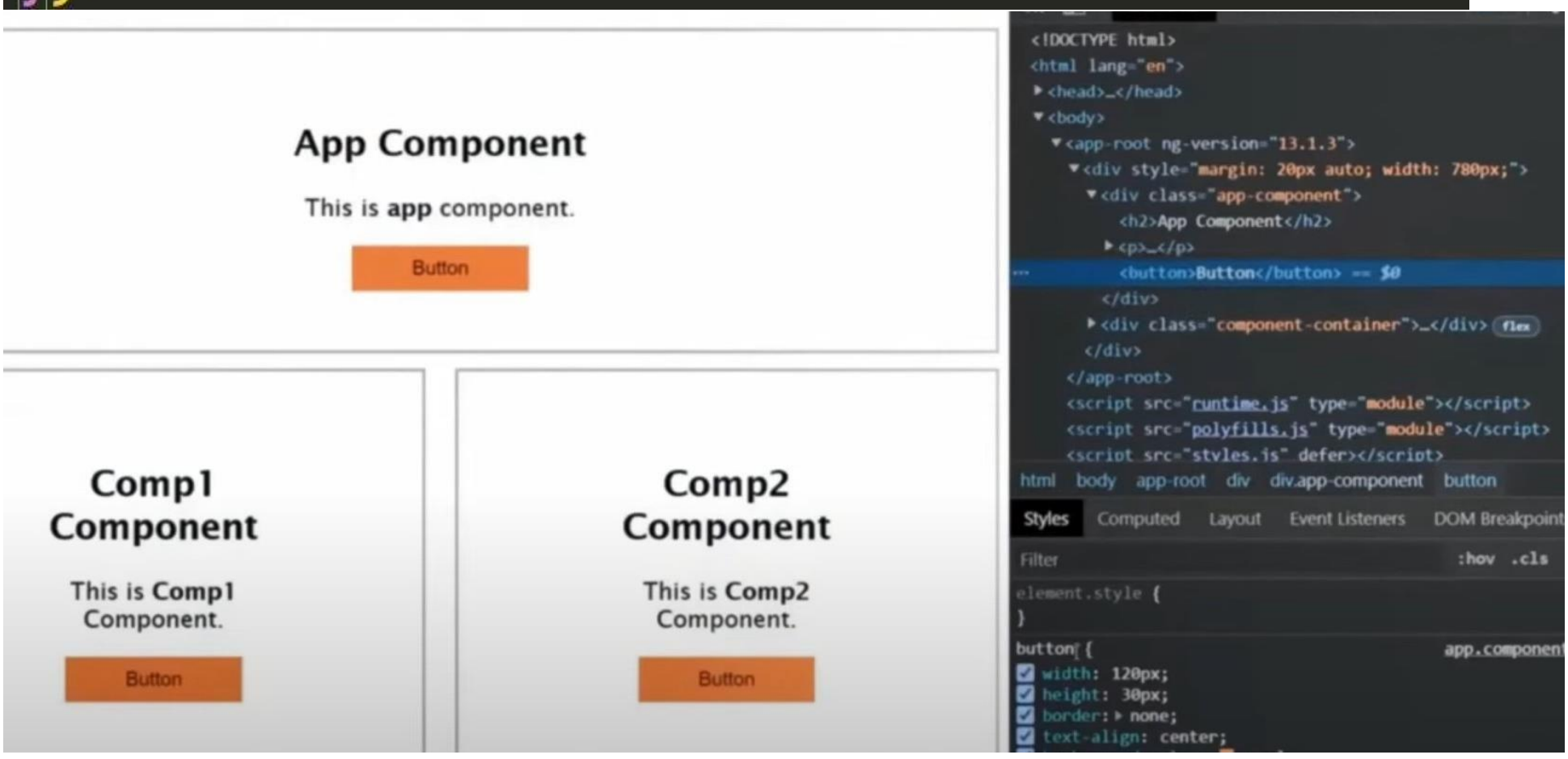


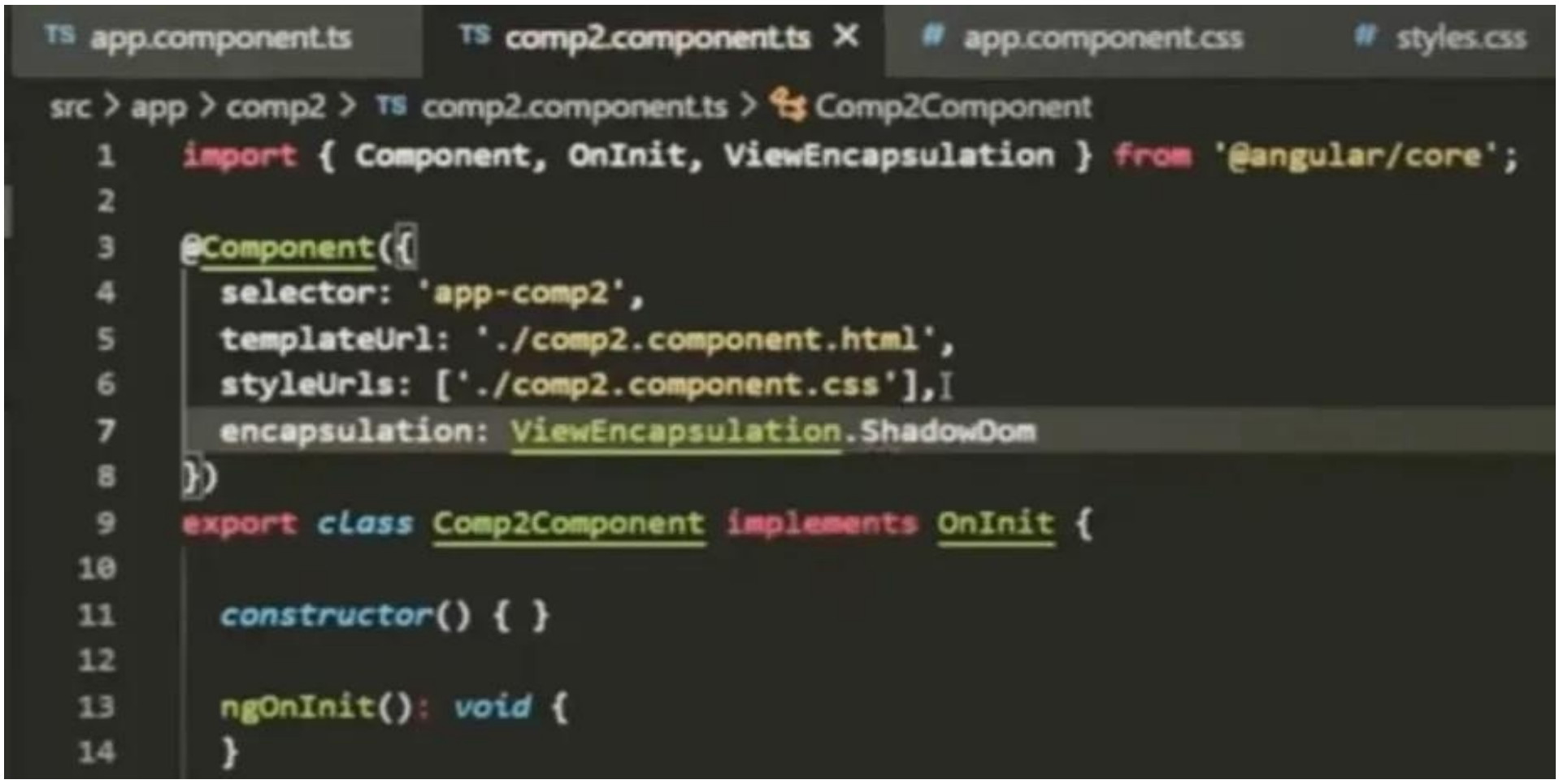
2. **None:**
   - **Behavior:** No encapsulation is applied. Styles are applied globally to all components.
   - **Example:** Button styles defined in a component will affect all button elements in child components or across the app.
   - **Usage:** `encapsulation: ViewEncapsulation.None`

View Encapsulation :

None ⇒ no properties will be applied to these in dom The usage should be added to @ Component Metadata.

```
@Component({
  selector: 'app-view-children',
  templateUrl: './view-children.component.html',
  styleUrls: ['./view-children.component.css'],
  encapsulation: ViewEncapsulation.None
})
```



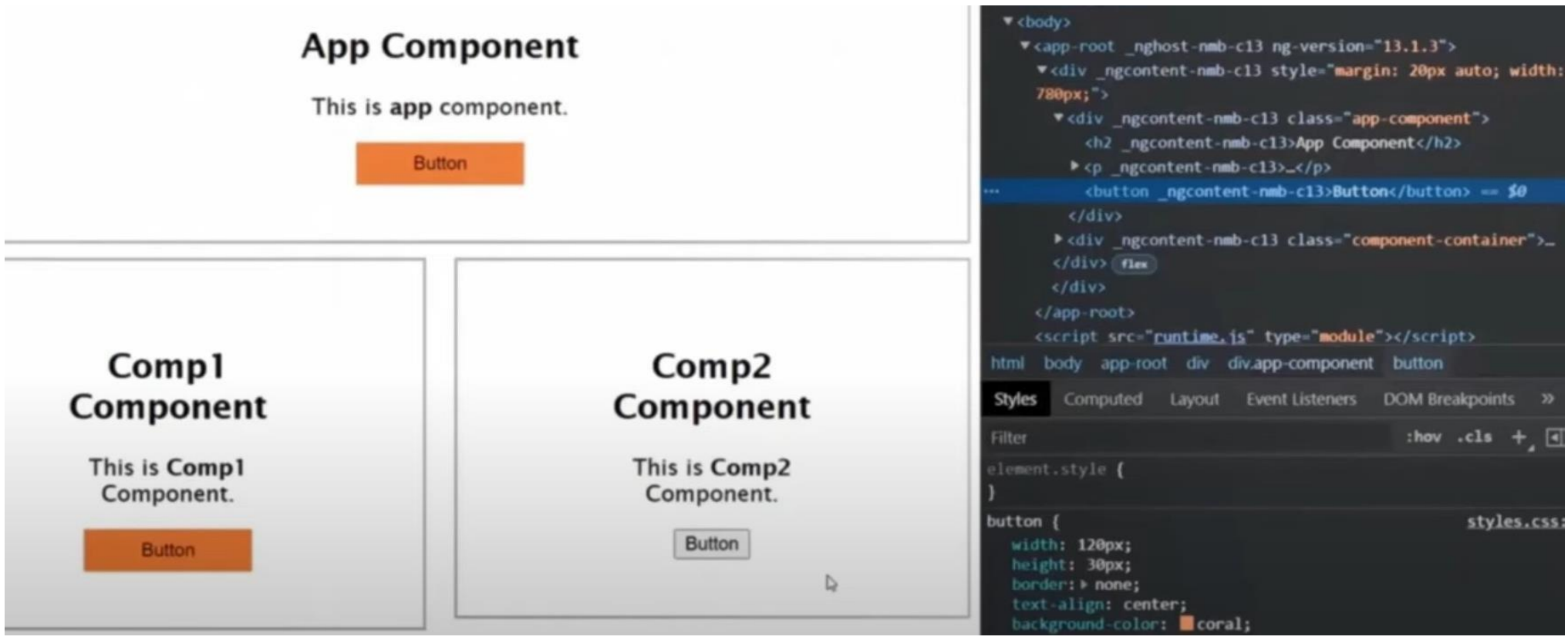


3. **Shadow DOM:**
    - **Behavior:** The component creates its own isolated DOM tree (Shadow DOM), making styles completely private to that component.
    - **Example:** Styles applied to a component with Shadow DOM do not affect the main DOM or other components.
    - **Usage:** `encapsulation: ViewEncapsulation.ShadowDom`

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

View Encapsulation :

shadowDom ⇒ No properties will be applied to the component which has this view encapsulation in dom.



Practical Example:

- **Scenario:** Suppose there are three components: `AppComponent`, `Comp1`, and `Comp2`, each with a `<button>` element.

- With **Emulated encapsulation**, button styles defined in `AppComponent`'s CSS will not apply to buttons in `Comp1` or `Comp2`.
- With **None encapsulation**, button styles in `AppComponent`'s CSS will apply to buttons in `Comp1` and `Comp2`.
- With **Shadow DOM encapsulation** in `Comp2`, the styles in `AppComponent` will not affect the buttons in `Comp2`.

Conclusion:

View encapsulation in Angular helps maintain style encapsulation by controlling how CSS is applied to components, ensuring that styles don't interfere across components.