02. Angular Essentials - Components - Templates - Services & More

# Module Introduction

**Lesson 8: What You’ll Learn in the Angular Essentials Section**

**Overview of This Section**

In this section, we’ll begin exploring the **crucial Angular Essentials**—the core concepts every Angular developer must understand.

And we’ll do so by **building a complete demo application** from the ground up.

This hands-on approach ensures you not only learn theory but also apply it immediately in a real-world context.

**What You Will Learn**

As we build the demo application together, you will:

**🔹 Understand Angular Project Structure**

* Learn how Angular projects are organized
* Explore the purpose of various **folders** and **configuration files**
* Understand how the **CLI scaffolds** a complete project

**🔹 Work with Components**

* Deepen your knowledge of Angular **components**, a fundamental building block
* Learn how to **create**, **use**, and **structure** components
* Understand how to **compose UIs** using multiple reusable components

**🔹 Master Core Angular Concepts**

You’ll get hands-on experience with:

* **TypeScript**: Understand its role in Angular and how to use it effectively
* **Declarative Code**: Learn how Angular helps you write expressive, intention-driven markup
* **Templates** and **bindings** for dynamic UI creation

**🔹 Handle User Interactions**

* Learn how to respond to **user events**, such as button clicks
* Bind methods in your component logic to actions in your template
* Update the UI based on **user input and interaction**

**✅ By the End of This Section**

You will be able to:

* Build **dynamic, interactive Angular applications**
* Understand and navigate a complete Angular project
* Use Angular’s key features confidently in real development scenarios

This is a foundational section that will **prepare you for all the deeper topics** coming later in the course.

# A new starting project & analysing the project structure

**Lesson 9: Exploring the Angular Project Structure**

**Starting with a Shared Code Base**

To begin building our **demo application** and dive into Angular's essential concepts, we’ll use a **starting project** that has already been set up for you.

* This project was created using the **Angular CLI**
* A download link is provided in the course so we all start from the **same baseline** [**https://github.com/mschwarzmueller/angular-complete-guide-course-resources/blob/main/attachments/02-essentials/01-starting-project.zip**](https://github.com/mschwarzmueller/angular-complete-guide-course-resources/blob/main/attachments/02-essentials/01-starting-project.zip)

Using this common starting point helps avoid differences caused by varying Angular CLI versions.

**Note on CLI Differences**

Depending on which **Angular CLI version** you use, you might see slight differences in the project structure.

For example:

* Some versions include a public/ folder for assets like the favicon
* In our provided project, the favicon is in the src/ folder

These are minor structural differences. The Angular code we write remains the **same** regardless.

**Project Structure Walkthrough**

Let’s take a closer look at the files and folders:

**📁 Root-Level Files**

These are primarily **configuration files**:

* **tsconfig.json and related files**  
  Control how **TypeScript** is compiled into JavaScript

⚠️ Don’t modify these unless you know what you're doing

* **package.json**  
  Lists **project dependencies**, including Angular packages
* **angular.json**  
  Configuration file for Angular CLI, including build settings and project structure
* **.editorconfig**  
  Defines formatting rules for consistent code styling across IDEs
* **.gitignore**  
  Tells Git which files/folders to exclude from version control (e.g., node\_modules)

**📁 src/ – The Heart of Your Angular Project**

This is where all your Angular development happens.

**Key contents:**

* **index.html**  
  The root HTML file that loads when users visit your app

Angular dynamically injects your components here

* **main.ts**  
  The **entry point** for your Angular application
  + Bootstraps (starts) your app
  + Loads the root component (e.g., AppComponent)

This is the **first TypeScript file executed** when your app starts

* **styles.css**  
  Global styles that apply across all Angular components
* **assets/**  
  A folder for static resources like images, logos, and fonts
* **app/**  
  The main workspace for developers
  + You’ll build most of your app here
  + Contains the root component and other custom components

**Component File Naming Differences**

Angular file naming conventions have evolved:

* Traditional format: app.component.ts, app.component.html, etc.
* Newer Angular 20+ projects may use: app.ts, app.html, etc.

These differences **do not affect functionality**. You can name files as you prefer, though the .component. convention is still widely used.

**Installing Dependencies**

After downloading the starting project, you'll likely see **errors in main.ts** or other files. This happens because required **node packages are not yet installed**.

To fix this:

1. Open your terminal
2. Navigate into the project folder
3. Run the following command:

npm install

This command installs all dependencies listed in package.json.

✅ **Important**: You only need to run this once after downloading the project—not every time you work on it.

⚠️ Ignore any warnings unless the process ends with an error.

**Running the Angular Development Server**

Once dependencies are installed, start the local dev server:

npm start

This command internally runs:

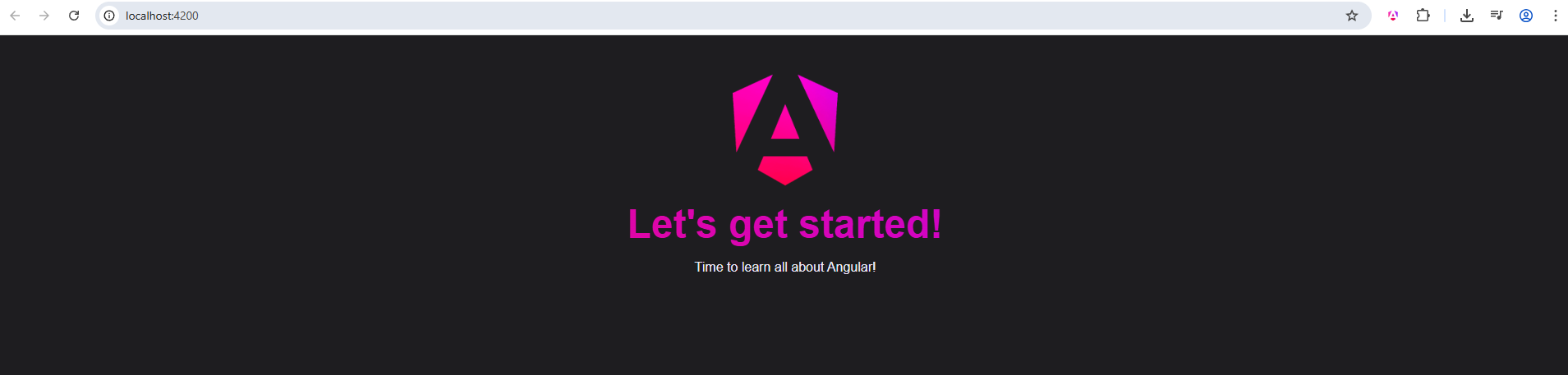
ng serve

It:

* Compiles your code
* Starts a development server
* Opens the app at:

http://localhost:4200

Visit that address in your browser, and you should see the default Angular welcome page.



**✅ Summary**

In this lesson, you:

* Explored the structure of an Angular CLI project
* Learned about the role of each major file and folder
* Installed project dependencies using npm install
* Started the dev server using npm start

In the **next lesson**, we’ll explore **how Angular renders this content** into the browser—beginning with the index.html, main.ts, and the **root component**.

# Understanding Components & How Content Ends Up on the Screen

**Lesson 10: How Angular Renders Content on the Screen**

**How Does Angular Content Appear in the Browser?**

In the previous lesson, we launched the development server and previewed the Angular app in the browser.

Now, let's answer the fundamental question:

**How does Angular actually render content into the browser window?**

**Step 1: The Role of index.html**

Open src/index.html.

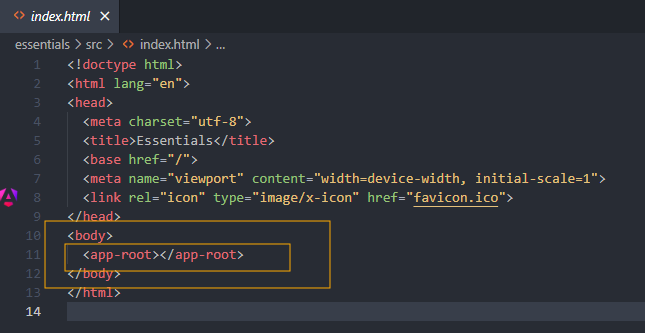
You'll notice it's nearly empty, except for this:

<body>

<app-root></app-root>

</body>

* <app-root> is **not a standard HTML element**
* The browser **doesn’t understand it on its own**



This is a **custom Angular component**, and it’s Angular’s job to replace it with meaningful content.

**Step 2: Angular Bootstraps in main.ts**

In src/main.ts, you’ll find:

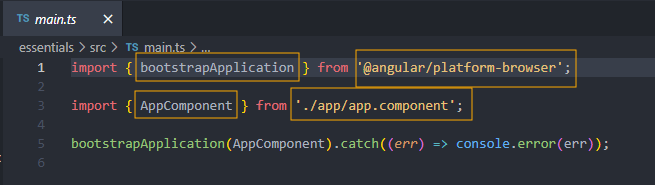
import { bootstrapApplication } from '@angular/platform-browser';

import { AppComponent } from './app/app.component';

bootstrapApplication(AppComponent);

Here’s what this does:

* When the app loads, the code in main.ts is **the first code executed**
* Angular’s bootstrapApplication() function:
  + **Bootstraps** the app using AppComponent
  + Searches for the component’s **selector** in the HTML (<app-root>)
  + **Replaces that selector** with the component’s rendered template

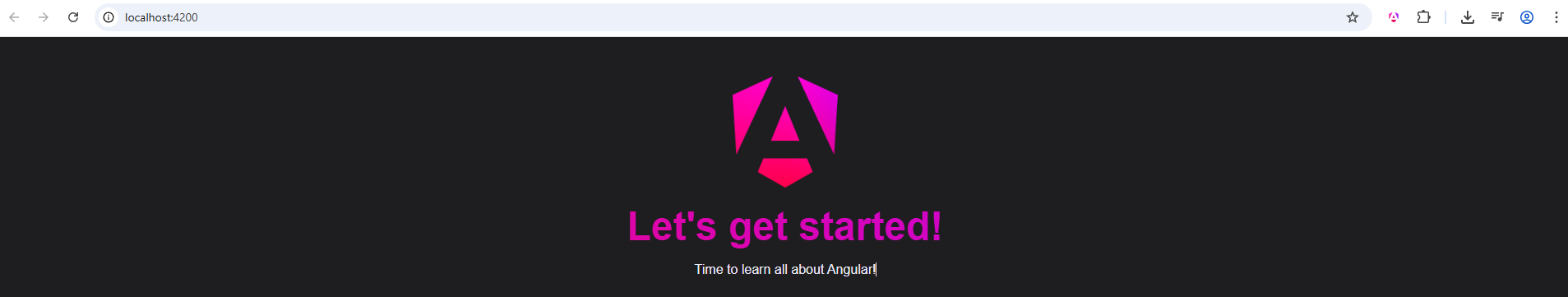


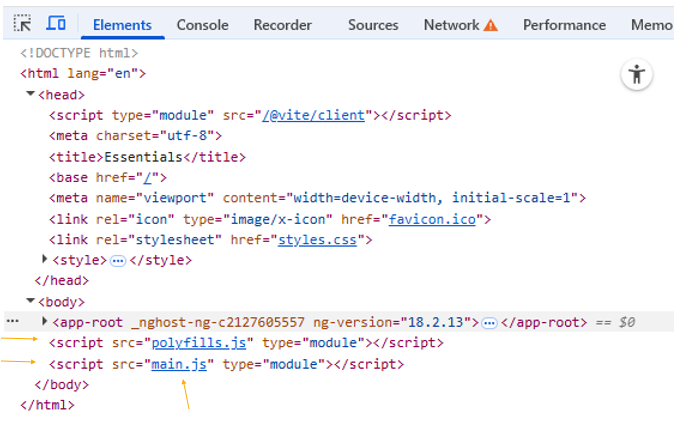
**Step 3: Where’s the Script Tag?**

Interestingly, index.html does **not** contain any <script> tags referencing main.ts.

So how is the app loaded?

* When you run ng serve (via npm start), the **Angular CLI**:
  + Compiles TypeScript to JavaScript
  + Injects compiled script files into index.html dynamically
  + Serves the site from memory to <http://localhost:4200>





**Step 4: How AppComponent Becomes a UI Element**

In src/app/app.component.ts:

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

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent {

title = 'My Angular App';

}

Let’s break this down:

**🔹 @Component Decorator**

This is a **TypeScript decorator** that marks the class AppComponent as an Angular component. It tells Angular how this component should behave.

Decorators are metadata providers. Angular uses them to:

* Define the component’s **HTML selector** (<app-root>)
* Link the component to an **HTML template**
* Attach **CSS styles** to the component

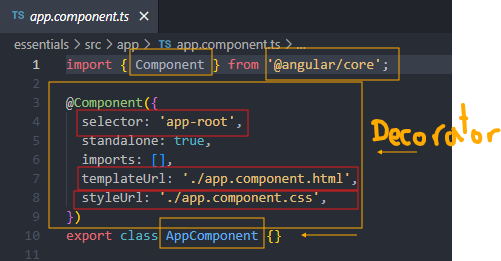
**🔹 selector: 'app-root'**

This property defines the **custom HTML tag** Angular will search for in index.html. When Angular finds <app-root>, it replaces it with the component’s template.

**🔹 templateUrl and styleUrls**

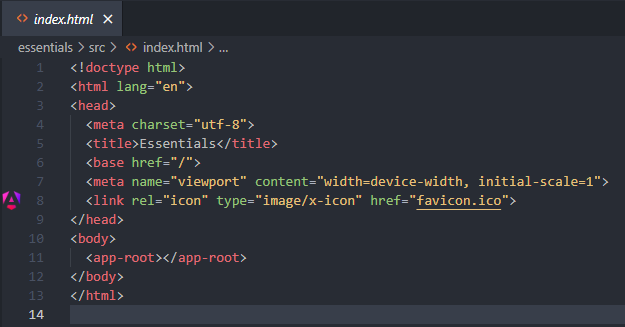
* templateUrl points to the external HTML file for this component
* styleUrls links to a CSS file whose styles apply **only** to this component

This scoping ensures your component styles don’t unintentionally affect others.



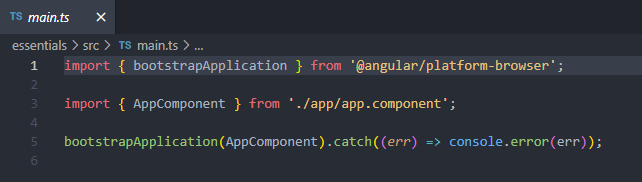
**✅ Visual Flow of Component Rendering**

User opens site → index.html loads



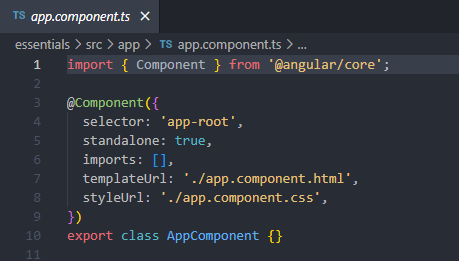
↓

Angular CLI injects JS → main.ts runs



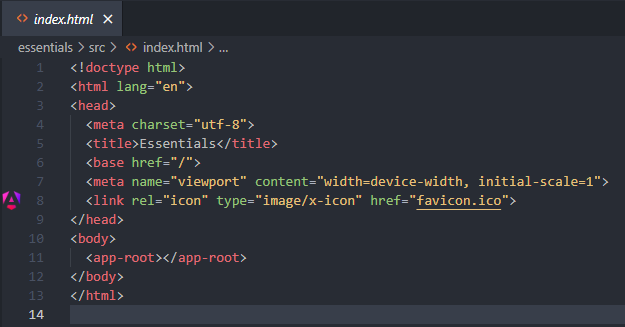
↓

bootstrapApplication(AppComponent)



↓

Angular looks for <app-root> in index.html



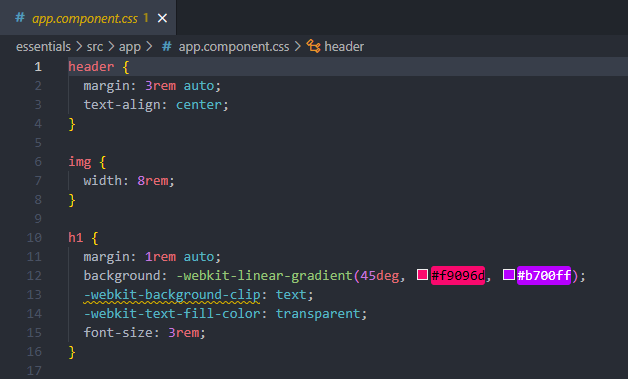
↓

Replaces <app-root> with app.component.html markup



↓

Applies scoped styles from app.component.css



**✅ Summary**

Angular renders components like this:

1. **index.html** loads and contains a custom tag like <app-root>
2. **main.ts** bootstraps the Angular application using AppComponent
3. Angular uses the selector defined in @Component to find and replace the custom tag
4. The **template HTML** is rendered in its place
5. The **CSS** is scoped and applied to only that component

This is how Angular "takes over" the DOM and renders your dynamic UI.

# Creating a First Custom Component

**Lesson 11: Creating a New Component – The HeaderComponent**

Now that we understand how the **root component** is rendered on the screen, we’re ready to begin building the **demo application** we mentioned earlier in this section.

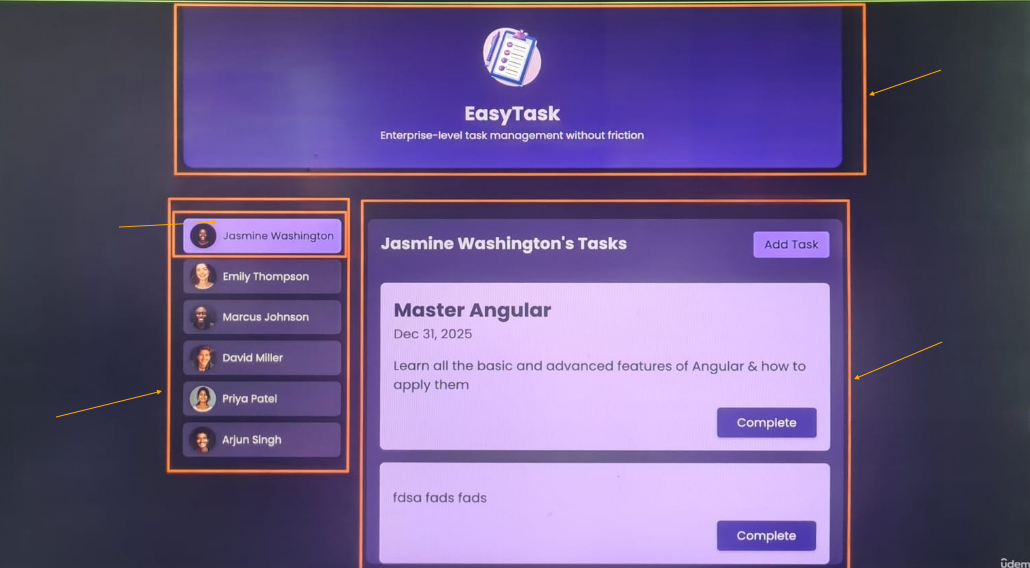
**🧱 Component-Based Architecture in Angular**

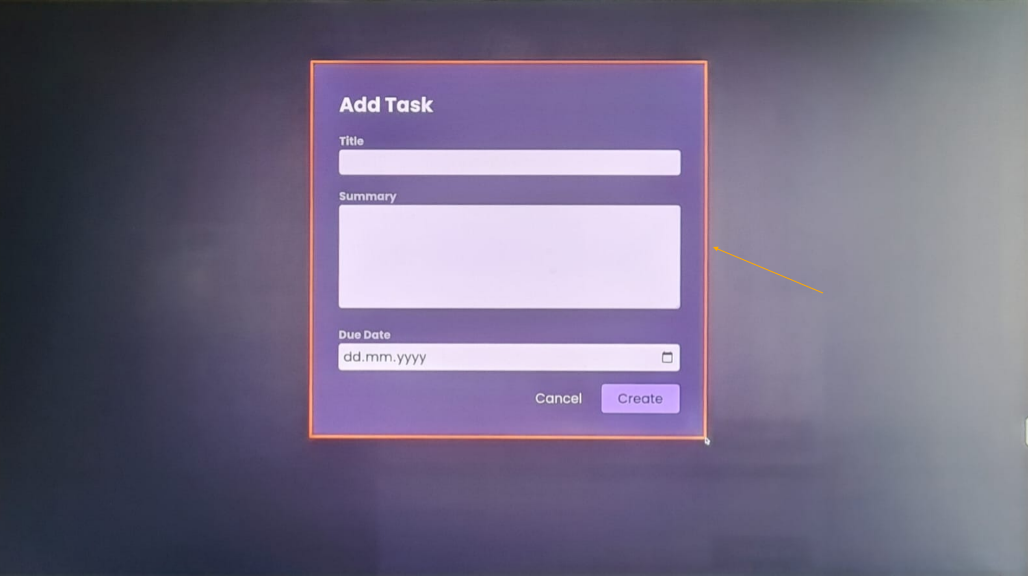
If you look at the final version of the demo app, you'll see that it's made up of **multiple UI sections**, such as:

* A **header** at the top
* A **sidebar** with task categories
* A **main content** area to the right
* A **dialog window** for adding tasks

Each of these UI parts can (and should) be implemented as **separate Angular components**. This is the core idea of Angular development:

Break the UI into **reusable, isolated building blocks**, and then compose them together.





**🎯 Step 1: Starting with the Header Component**

We’ll begin with the **header**, which will be the first custom component we create.

Angular components are generally built using **three files**:

| **File** | **Purpose** |
| --- | --- |
| header.component.ts | Defines the component’s logic |
| header.component.html | Holds the component’s template |
| header.component.css | Contains scoped styles |

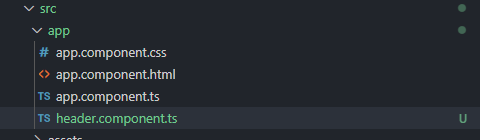
This is the **traditional naming convention** used since Angular 2.

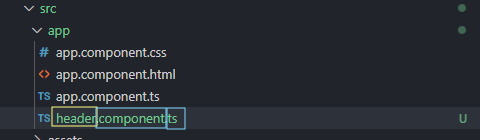
**📛 Angular 20 File Naming Note**

Starting from **Angular 20**, the CLI now encourages simpler file names like:

* header.ts instead of header.component.ts
* header.html instead of header.component.html

You are free to follow either convention. Most projects in the industry still use the traditional format, which is what we’ll use in this course for consistency.





**🛠 Step 2: Creating the Component Class**

Inside header.component.ts, we start by defining the **component class**:

export class HeaderComponent {

}

* The class must be **exported** so it can be used elsewhere
* By convention, the class name should describe what it does
  + In this case: HeaderComponent

**🧩 Step 3: Turning the Class into a Component**

In Angular, components are created using **decorators**. Specifically, we use @Component from the Angular core package.

**Full Setup:**

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

@Component({

selector: 'app-header',

templateUrl: './header.component.html',

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

})

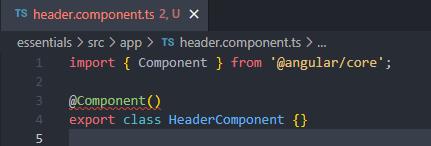
export class HeaderComponent {

}

**📌 What Each Part Means:**

* @Component(...): This is a **decorator** that marks this class as an Angular component
* selector: Defines the **custom HTML tag** you’ll use (e.g., <app-header>)
* templateUrl: Points to the external HTML file
* styleUrls: Links to a list of **CSS files scoped** to this component

These styles will apply **only** to the HeaderComponent, not globally.



**✅ Summary**

You now understand:

* How to plan your app as a set of components
* How to **create and name** a new component
* How to write the **class and decorator** for the component
* The structure and role of the component’s files

Next, we’ll:

* Create the header.component.html and header.component.css files
* Write our **template** and **styles**
* Render <app-header> in the app!

# [Optional] JavaScript Refresher: Classes, Properties & More

**[Optional] JavaScript Refresher: Classes, Properties & More**

Angular makes heavy use of classes - a feature that's supported by vanilla JavaScript and TypeScript (though TypeScript "extends" it and adds some extra features as you'll see).

A class is essentially a blueprint for objects. Any properties and methods defined in the class will exist on all objects that are created based on the class.

For example, if you had this class (in vanilla JavaScript):

1. class Person {
2. constructor(name, age) {
3. this.name = name;
4. this.age = age;
5. }
7. greet() {
8. console.log('Hi, I am ' + this.name);
9. }
10. }

You could instantiate it (and create objects) like this:

1. const person1 = new Person('Max', 35);
2. const person2 = new Person('Anna', 32);

And you could then access the properties and methods defined by the class:

1. console.log(person1.age);
2. person2.greet();

When using Angular, you'll often define classes which are NEVER instantiated by you!

For example, components are created as classes - i.e., you create blueprints for custom HTML elements. But it's Angular that actually instantiates the classes in the end. You never call new SomeComponent() anywhere in your code.

In addition, Angular uses TypeScript - therefore, you often use TS-supported "enhancements" to classes.

For example decorators:

1. @Component({})
2. class SomeComponent {}

Decorators like @Component are used by Angular to add metadata & configuration to classes (and other things, as you'll see throughout the course).

In addition, TypeScript gives you more control over how properties are defined in classes.

You can, for example, mark properties (and methods) as private, public (the default) and protected to control which parts of your code can access which property (or method). You can learn more about these keywords [here](https://www.typescriptlang.org/docs/handbook/2/classes.html#member-visibility).

And, in general, you can learn more about TypeScript's support for classes [here](https://www.typescriptlang.org/docs/handbook/2/classes.html).

That being said, you don't have to study classes in-depth right now. You'll see most of those important features in action throughout this course.

For the moment, it's just important to understand that this classes feature exists, what it does (= create blueprints for objects) and how to work with classes.

# Configuring the Custom Component

**Lesson 12: Configuring the HeaderComponent**

In the previous lesson, we created a HeaderComponent class and imported the @Component decorator. Now, let’s complete its configuration and see how we can **use it in the app**.

**🧠 Step 1: The Configuration Object**

When you use the @Component decorator, you pass it a **configuration object** that tells Angular how the component should behave.

@Component({

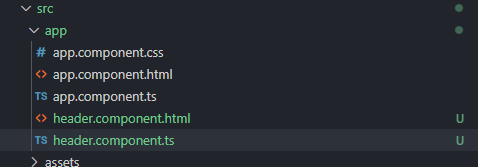
selector: 'app-header',

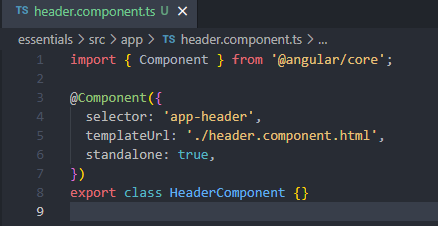
templateUrl: './header.component.html',

styleUrls: ['./header.component.css'],

standalone: true

})





Let’s break this down:

**🏷 selector: Defining the Custom Tag**

The selector tells Angular:

*“Look for an HTML element with this tag and replace it with this component’s template.”*

**✅ Best Practice:**

Use **custom selectors** with at least **two words**, separated by a dash:

selector: 'app-header'

❌ Avoid single-word tags like header, which might conflict with built-in HTML elements  
✅ Good examples: app-header, my-app-navbar, custom-footer

The prefix (app-) is a naming convention. You can customize it, but app- is widely used and recognized.

**🧾 templateUrl: Linking to an External HTML File**

Angular components usually reference an external HTML file for the template:

templateUrl: './header.component.html'

You could technically write inline templates like this:

template: `<h1>Easy Task</h1>`

…but this is discouraged unless your template is extremely short (1–2 lines). For anything more complex, always use templateUrl.

**🛠 Creating the HTML File:**

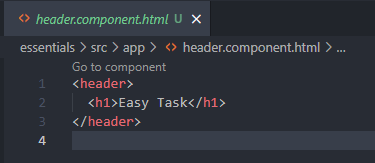
File name: header.component.html  
Contents:

<header>

<h1>Easy Task</h1>

</header>

Here we use the standard HTML <header> element to semantically define the top section of our UI.



**🎨 styleUrls: Adding Scoped CSS**

You can also create a CSS file for the component’s styles:

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

These styles will be **scoped only to this component**, meaning they won't leak into other parts of your app.

We’ll expand on this in the next lesson when we add styles.

**🚀 standalone: true: Modern Angular Development**

This setting is very important:

standalone: true

This marks your component as a **Standalone Component**, meaning:

* It **doesn't require a module** to be declared in
* It's the **recommended modern format** starting from Angular 14
* It can be easily used and composed with other standalone components

**Version Compatibility:**

| **Angular Version** | **Default Value** | **Required?** |
| --- | --- | --- |
| Angular 18 | false | ✅ Set manually |
| Angular 19+ | true | ✅ Optional to set |

✅ Tip: Always set standalone: true for compatibility across versions.

**🧠 What Are Standalone Components?**

* Angular used to require every component to be part of an **NgModule**
* Since Angular 14, you can use **standalone components** instead
* Standalone components are **easier to manage**, especially in small to mid-size apps
* You’ll learn about the older NgModule-based components later in the course for compatibility with legacy projects

**✅ Summary**

Your HeaderComponent setup now looks like this:

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

@Component({

selector: 'app-header',

templateUrl: './header.component.html',

styleUrls: ['./header.component.css'],

standalone: true

})

export class HeaderComponent { }

You’ve now completed the configuration for your component. In the next lesson, we’ll:

* **Render <app-header> in the app**
* Explain **how to use a standalone component**
* Add **component-level CSS styles**

# Using the Custom Component

**Lesson 13: Rendering and Registering Standalone Components**

Now that our HeaderComponent is fully configured and ready, it’s time to **use it in the application**. But how exactly does Angular become aware of the component and display it?

Let’s walk through this.

**🧪 First Attempt: Placing <app-header> in index.html**

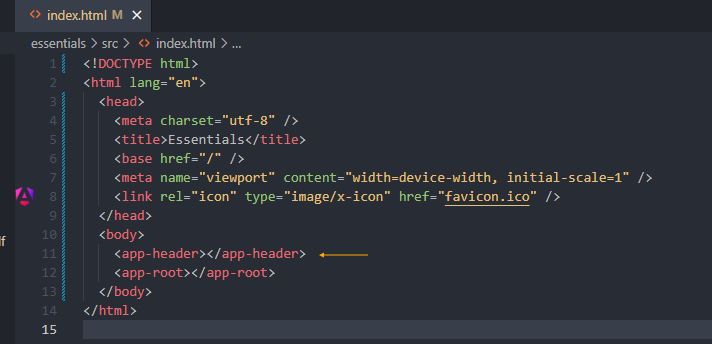
You might think you can simply add the custom element in index.html, like this:

<body>

<app-root></app-root>

<app-header></app-header>

</body>

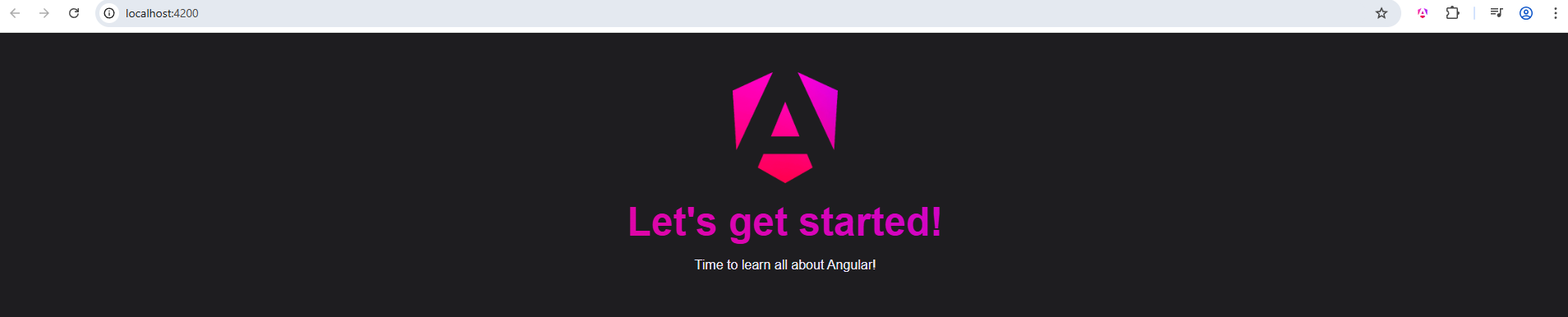


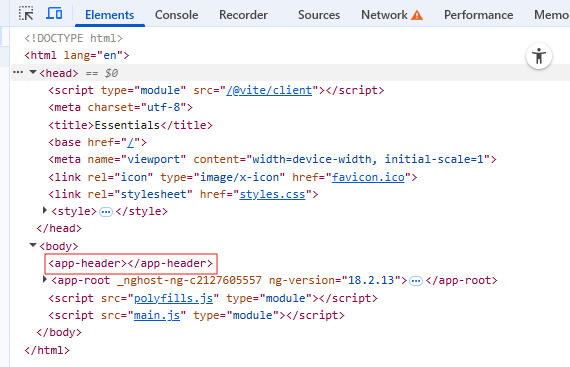
But if you do this:

* You’ll see **no output from <app-header>**
* If you inspect the page in your browser’s dev tools, you’ll see:

<app-header></app-header>

… but it’s **empty**, and Angular **ignores it**

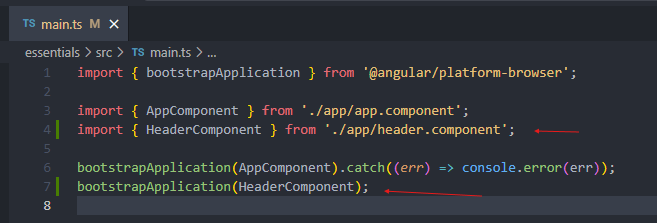


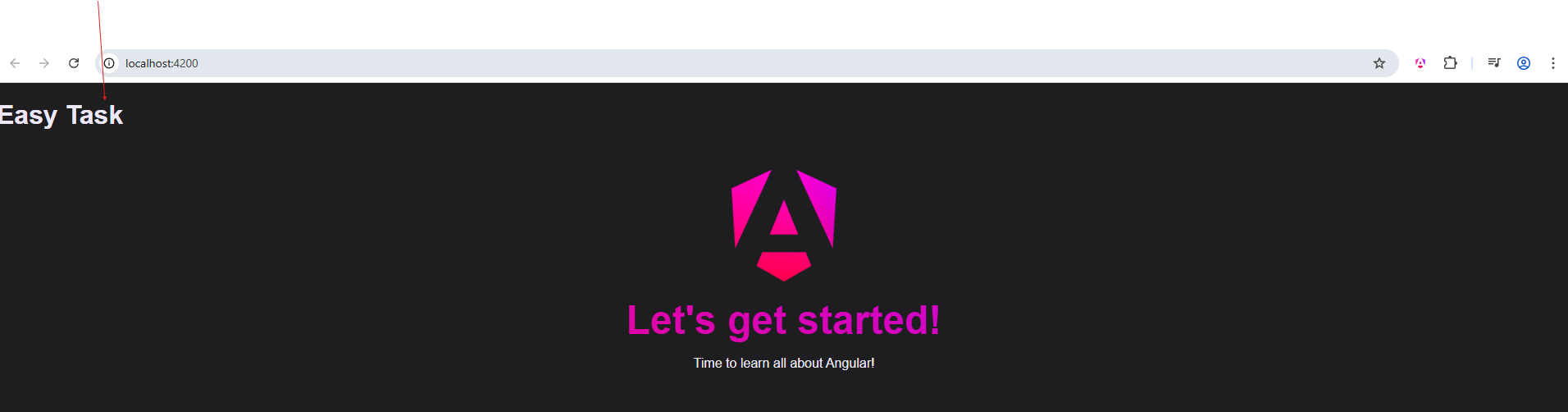


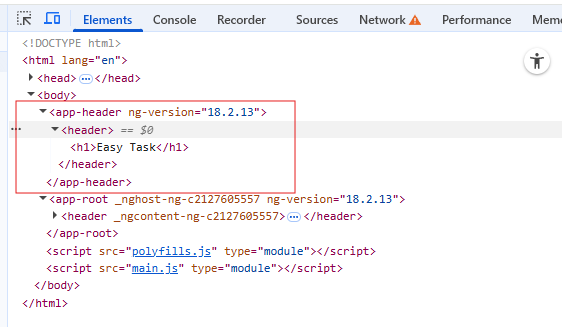
**❗ Why Doesn’t Angular Render It?**

Angular doesn’t automatically scan your project for components. Instead, it only becomes aware of a component when you **explicitly register** it in one of two ways:

1. Using it as the **bootstrapped root component**







1. **Importing it** into another component’s imports array

**🧱 The Right Way: Nest Components in the Component Tree**

Instead of calling bootstrapApplication() multiple times (once for each component), the Angular **best practice** is:

Bootstrap a **single root component** (usually AppComponent) and use all other components **within its template or sub-components**

This builds a **component tree**:

AppComponent

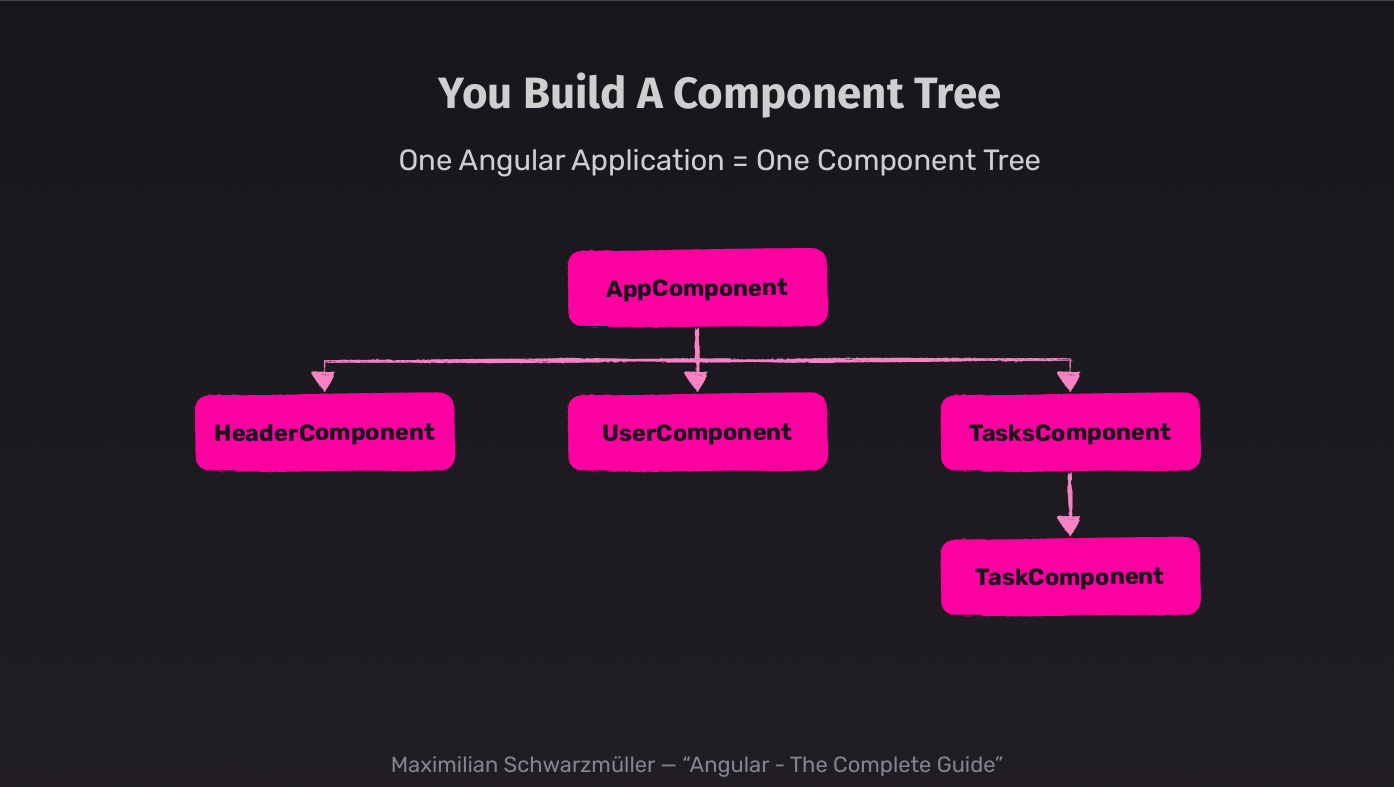
│

├── HeaderComponent

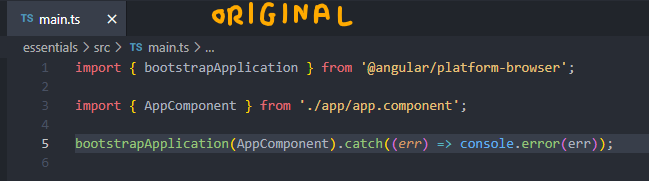
├── SidebarComponent

└── DialogComponent

* This tree structure enables **data sharing**, **event communication**, and **hierarchical structure**
* It's **central to Angular’s design**



**✅ Step-by-Step: Rendering the HeaderComponent Correctly**



**🔹 1. Use the Component in AppComponent Template**

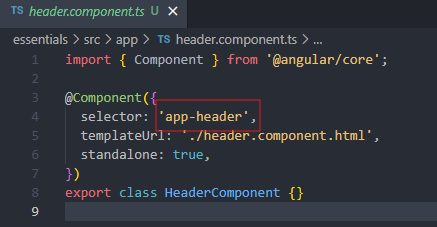
In app.component.html, replace the existing static <header> markup with:

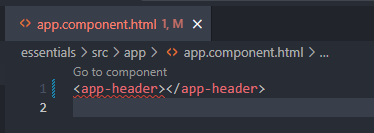
<app-header></app-header>

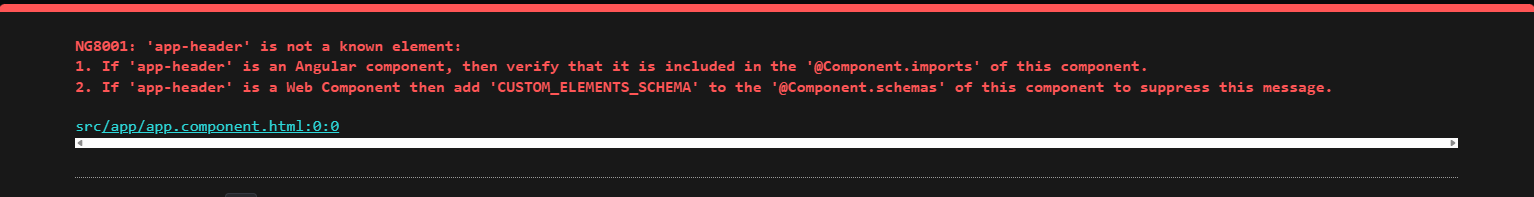
But if you do this without further steps, you’ll get an error in the terminal:

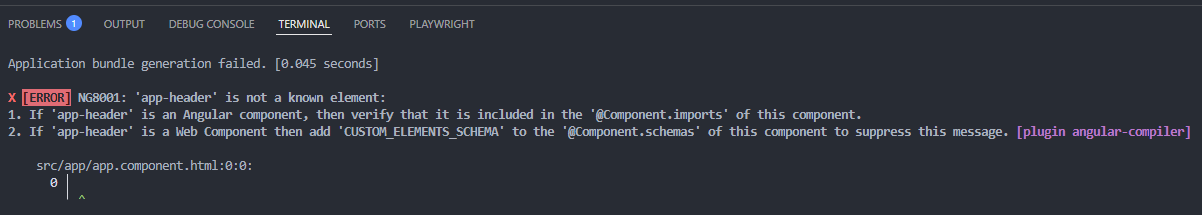
'app-header' is not a known element

This means Angular doesn’t yet know how to process this component.







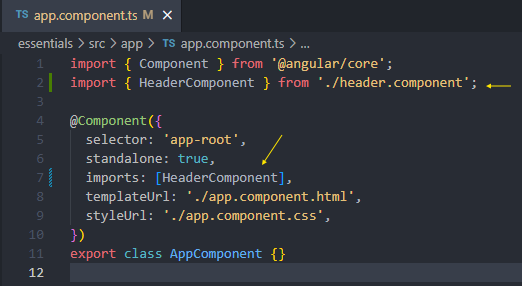


**🔹 2. Register the Component in AppComponent**

Go to app.component.ts and:

1. **Import** the HeaderComponent:
2. import { HeaderComponent } from './header.component';
3. **Add it to the imports array** of the @Component decorator:
4. @Component({
5. selector: 'app-root',
6. standalone: true,
7. imports: [HeaderComponent],
8. templateUrl: './app.component.html',
9. styleUrls: ['./app.component.css']
10. })

✅ This step registers HeaderComponent for use inside the AppComponent’s template.





**🔁 What Happens Now?**

When Angular compiles the app:

1. It bootstraps AppComponent
2. Finds <app-header> in its template
3. Recognizes that HeaderComponent is imported
4. Replaces <app-header> with the content from header.component.html

✅ Result: The header appears on the screen as expected.

**🧠 Why This Matters**

This component registration system:

* Ensures only **relevant components** are included where needed
* Makes Angular’s **tree-shaking and optimization** more efficient
* Enables clean, maintainable **component encapsulation**

**✅ Summary**

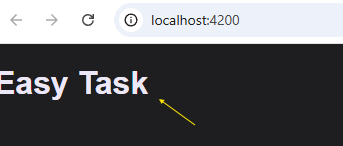
* Angular **does not auto-discover** components
* You must **register** components using the imports array (if standalone)
* Use **one bootstrapApplication()** call—build your UI through a **component tree**
* You can now see <app-header> correctly rendered in your app

In the next lesson, we’ll add **styles** to HeaderComponent and explore **how Angular scopes CSS**, along with visual illustrations of what that CSS does.

# Styling the Header Component & Adding an Image

**Lesson 14: Styling the HeaderComponent and Finalizing It**

You’ve now successfully rendered the content of the HeaderComponent. But stylistically, it still looks very basic. Let’s fix that by adding **scoped styles**, a **logo image**, and a **refined layout**.



**🎨 Step 1: Add header.component.css**

Create a file named:

header.component.css

This file contains the following styles:

header {

display: flex;

flex-direction: column;

align-items: center;

gap: 1rem;

width: 90%;

max-width: 50rem;

margin: 0 auto 2rem auto;

text-align: center;

background: linear-gradient(to bottom, #2c0a4c, #450d80);

padding: 1rem;

border-bottom-right-radius: 12px;

border-bottom-left-radius: 12px;

box-shadow: 0 1px 8px rgba(0, 0, 0, 0.6);

}

img {

width: 3.5rem;

object-fit: contain;

}

h1 {

font-size: 1.25rem;

margin: 0;

padding: 0;

}

p {

margin: 0;

font-size: 0.8rem;

text-wrap: balance;

}

@media (min-width: 768px) {

header {

padding: 2rem;

}

img {

width: 4.5rem;

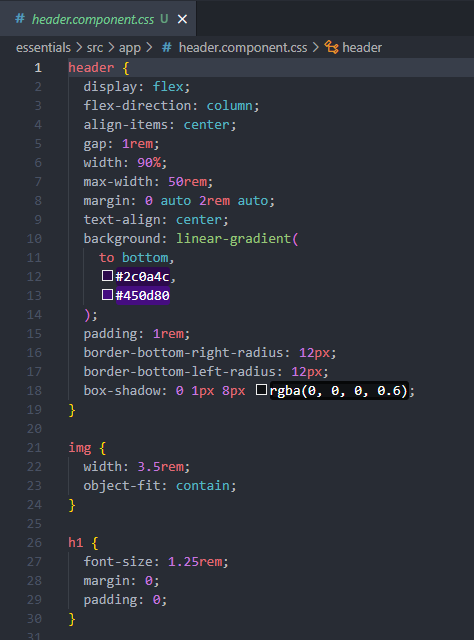
}

h1 {

font-size: 1.5rem;

}

}





**✅ Key Highlights:**

| **Selector** | **Explanation** |
| --- | --- |
| header | Creates a centered container with rounded bottom corners, a purple gradient, padding, and shadow |
| img | Restricts the logo’s width and ensures it doesn’t stretch |
| h1, p | Centered, modern, minimal headline and subtext |
| @media | Ensures responsiveness on larger screens |

**🖼️ Visual Illustration**

Here’s how the header CSS would visually affect a <div> element if applied:

* The purple gradient background
* Rounded corners
* Box shadow
* Centered content and spacing between items

**🧾 Step 2: Link the Stylesheet**

In header.component.ts:

@Component({

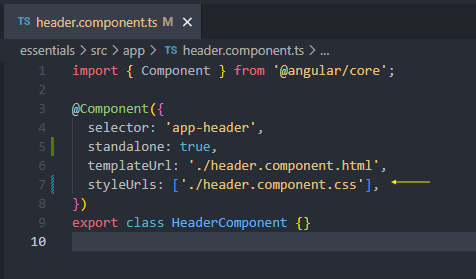
...

styleUrls: ['./header.component.css'],

...

})

✅ Use styleUrls (plural) for multiple stylesheets. You can also use styles: [...] for inline CSS—but that’s discouraged except for 1-liners.



**🧩 Step 3: Update the HTML**

In header.component.html, use this structure:

<header>

<img src="assets/task-management-logo.png" alt="A todo list" />

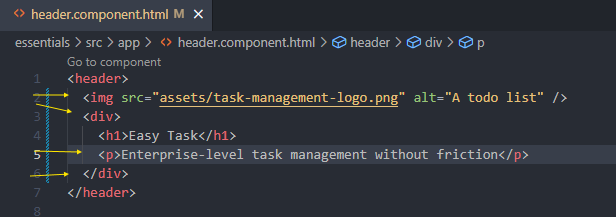
<h1>Easy Task</h1>

<p>Enterprise-level task management without friction</p>

</header>

This will display the logo and app title in a styled container.

Ensure the image task-management-logo.png is placed in your /src/assets/ folder.



**🌍 Step 4: Global Styling via styles.css**

Replace your global styles.css with the updated one:

\* {

box-sizing: border-box;

}

html {

height: 100%;

}

body {

font-family: "Poppins", sans-serif;

background: radial-gradient(circle at top left, #181023, #0b0519);

color: #c3b3d8;

margin: 0;

padding: 0;

height: 100%;

}

These styles apply to all components globally—setting a dark theme and loading the **Poppins** font.



**🔗 Step 5: Update index.html with Google Fonts**

To enable the font, replace your index.html head section with this updated version:

<link rel="preconnect" href="https://fonts.googleapis.com" />

<link rel="preconnect" href="https://fonts.gstatic.com" crossorigin />

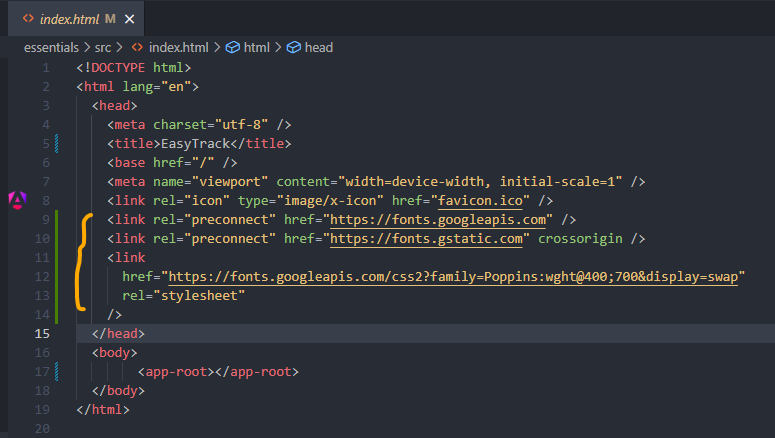
<link

href="https://fonts.googleapis.com/css2?family=Poppins:wght@400;700&display=swap"

rel="stylesheet"

/>

✅ These links ensure the custom font is loaded before rendering.



**🛠 Final Configuration: angular.json**

Make sure angular.json contains this assets array:

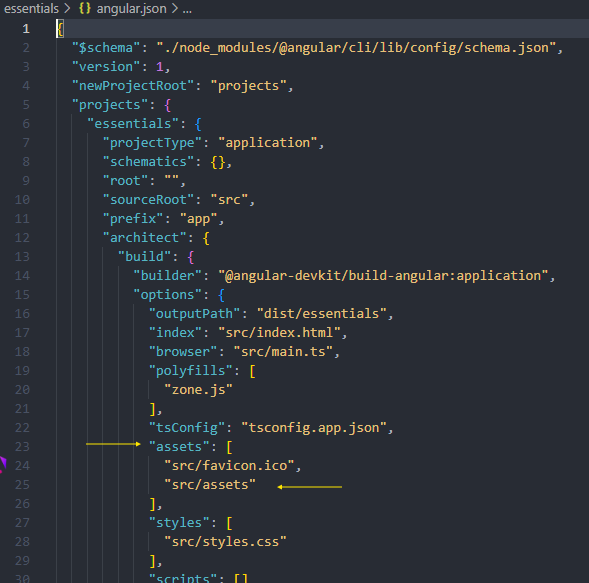
"assets": [

"src/favicon.ico",

"src/assets"

]

This ensures images like the task logo are served correctly.

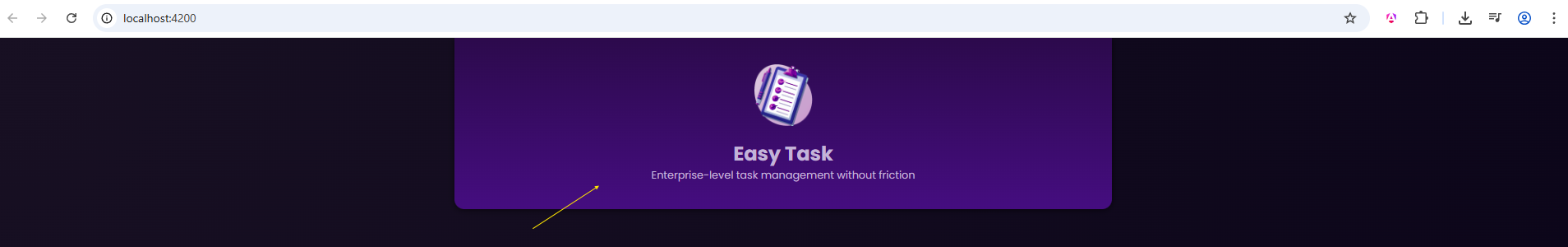


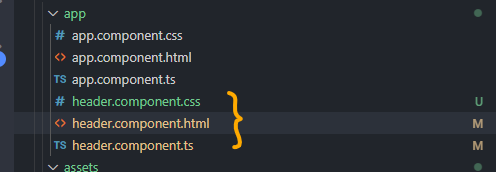
**✅ Summary**

You now have:

* A beautifully styled, responsive HeaderComponent
* Custom font integration and background
* Image handling and scoped component styles
* Clean separation between **global** and **component-level styles**

🎉 You’ve completed your **first fully styled, production-ready Angular component**!





# Managing & Creating Components with Angular CLI

**Lesson 15: Organizing and Generating Angular Components**

Congratulations! 🎉 You’ve just built and styled your first custom component — the HeaderComponent.

As you continue developing Angular applications, you’ll be building **many more components**, and this means **managing your file structure** becomes increasingly important.

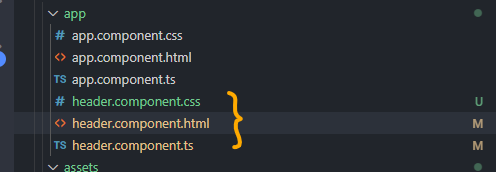
**🗂 Problem: Component Clutter**

By default, all new components you create manually are placed inside the src/app folder.

Each component typically includes:

* A .ts file
* An .html file
* A .css file
* (Optionally) a .spec.ts file for testing

With many components, this results in a crowded and hard-to-manage folder.



**✅ Solution: Organize with Subfolders**

To keep things clean and maintainable, it’s **best practice** to create a dedicated subfolder for each component.

For example:

src/

└── app/

├── header/

│ ├── header.component.ts

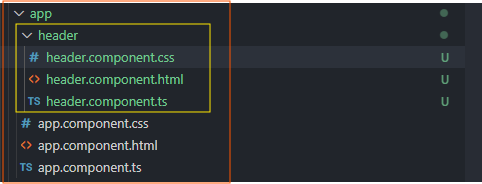
│ ├── header.component.html

│ └── header.component.css

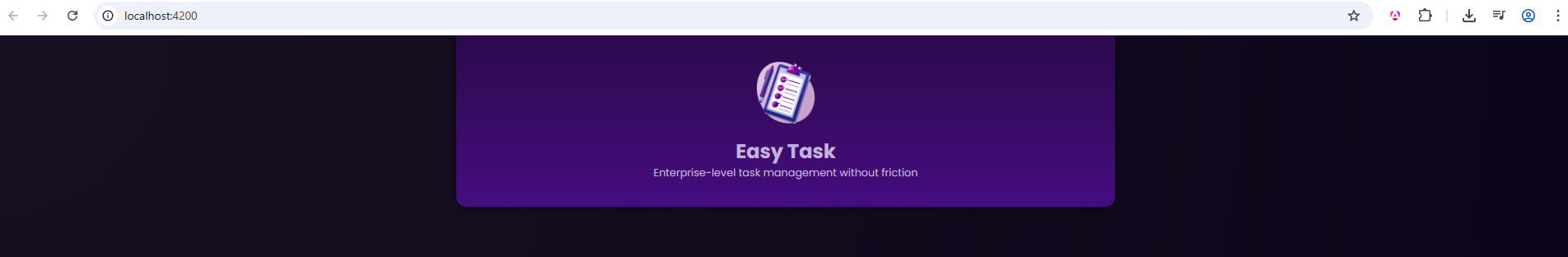
Move all three header files into a header subfolder:

* Update any **import paths** in app.component.ts to reflect the move
  + Most modern IDEs (like VS Code) will update them automatically

Once done, your app will **continue to work** exactly as before — just with better structure.







**⚡ Using the Angular CLI to Generate Components**

Instead of manually creating folders and files, Angular provides a **powerful CLI tool** that can generate components for you.

**👨‍💻 Syntax:**

ng generate component folder-name/component-name

or the shorthand:

ng g c folder-name/component-name

For example, to generate a UserComponent in a user folder:

ng g c user

✅ This command creates a user folder inside src/app and generates the following files:

src/

└── app/

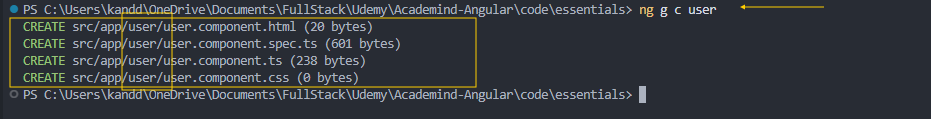
└── user/

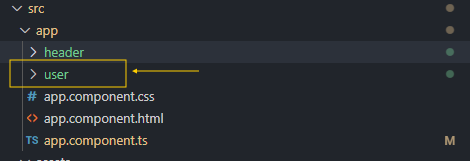
├── user.component.ts

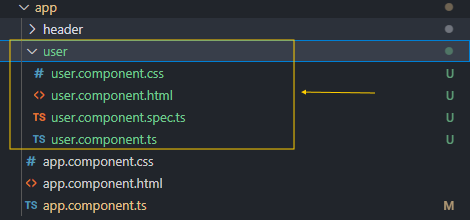
├── user.component.html

├── user.component.css

└── user.component.spec.ts







**🔍 A Look Inside user.component.ts**

The CLI generates a fully configured component, including:

* A class named UserComponent
* A proper selector: 'app-user'
* A templateUrl and styleUrls correctly linked
* standalone: true enabled (if you're using Angular 14+)
* An empty imports array (optional — remove if unused)

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

@Component({

selector: 'app-user',

standalone: true,

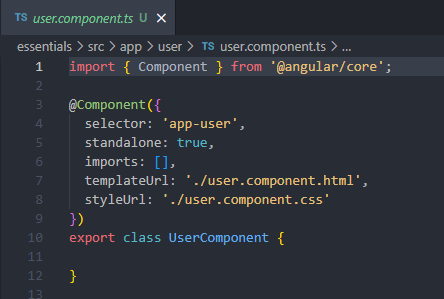
imports: [],

templateUrl: './user.component.html',

styleUrls: ['./user.component.css']

})

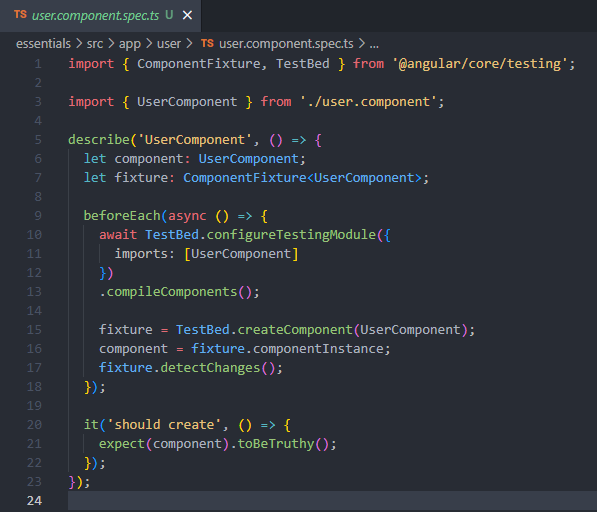
export class UserComponent { }



**🧪 About user.component.spec.ts**

This file is for **automated testing** using Jasmine/Karma. Since you’re not writing tests right now:

✅ You can safely **delete this file** and re-add it later when you learn testing later in the course.



**✅ Summary**

In this lesson, you learned:

* How to **organize component files into folders**
* Why this keeps your codebase **clean and scalable**
* How to use the **Angular CLI** to generate components in seconds
* What files the CLI generates and what you can keep or remove

You now have all the tools to **scale your application cleanly** using efficient structure and tools.

# Styling & Using Our Next Custom Component

Great! Let's now **review and explain the current state of your project** based on the last lecture and your uploaded files. You’ve:

1. Created a new UserComponent using the Angular CLI.
2. Updated the user.component.html markup with a div, button, img, and span.
3. Pasted styles into user.component.css.
4. Integrated the <app-user /> tag into the AppComponent.
5. Pasted updated styles into app.component.css.

**✅ Resulting Code (Combined Summary)**

**📄 user.component.html**

<div>

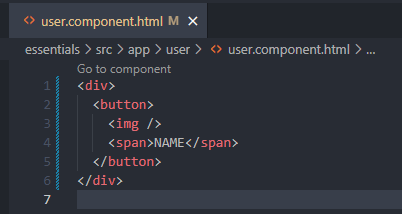
<button>

<img src="" alt="User avatar" />

<span>Username</span>

</button>

</div>



**🎨 user.component.css Explanation**

div {

border-radius: 6px;

box-shadow: 0 1px 6px rgba(0, 0, 0, 0.1); /\* Soft shadow \*/

overflow: hidden;

}

🖼️ **Image Effect on a div**:

button {

display: flex;

align-items: center;

gap: 0.5rem;

padding: 0.35rem 0.5rem;

background-color: #433352;

color: #c3b3d1;

border: none;

font: inherit;

cursor: pointer;

width: 100%;

min-width: 10rem;

text-align: left;

}

🧠 This makes the button stretch the full width of the parent and positions the image and text horizontally with spacing (gap).

button:hover,

button:active,

.active {

background-color: #9965dd;

color: #150722;

}

🎨 Applies hover/active/focus styling.

img {

width: 2rem;

object-fit: contain;

border-radius: 50%;

box-shadow: 0 1px 8px rgba(0, 0, 0, 0.3);

}

📷 The image becomes a **circular avatar** with a glow-like shadow.

span {

margin: 0;

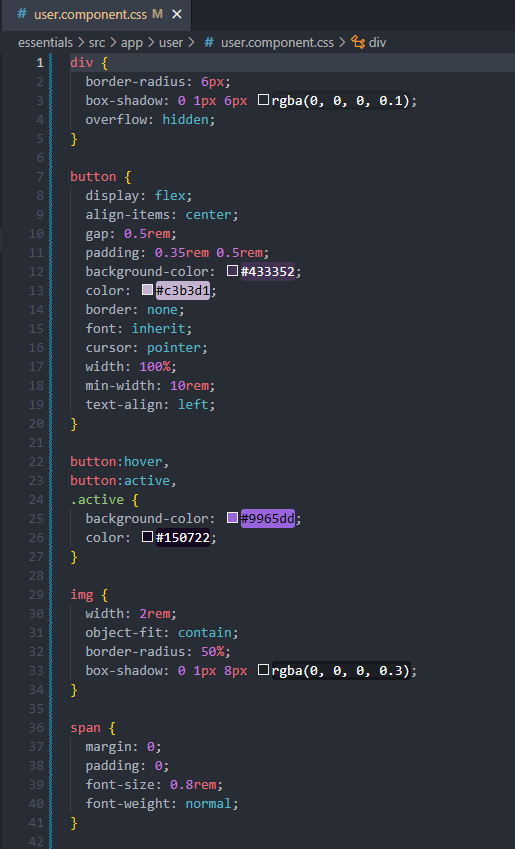
padding: 0;

font-size: 0.8rem;

font-weight: normal;

}

Simple name text aligned to the avatar.



**🎨 app.component.css Explanation**

main {

width: 90%;

max-width: 50rem;

margin: 2.5rem auto;

display: grid;

grid-auto-flow: row;

gap: 2rem;

}

Lays out the main app body in a **single column grid** on small screens and adds spacing between items.

#users {

display: flex;

gap: 0.5rem;

overflow: auto;

list-style: none;

margin: 0;

padding: 0;

}

This styles the user list for horizontal scrolling and removes default list appearance.

@media (min-width: 768px) {

main {

grid-template-columns: 1fr 3fr;

margin: 4rem auto;

}

#users {

flex-direction: column;

}

#fallback {

font-size: 1.5rem;

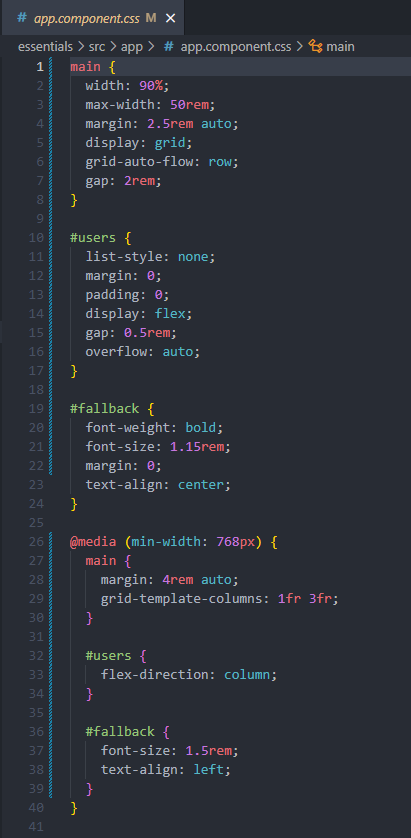
text-align: left;

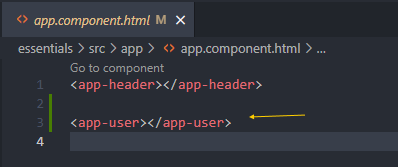
}

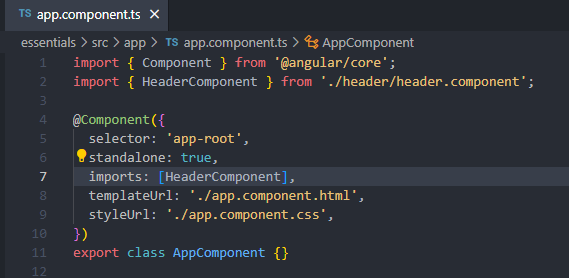
}

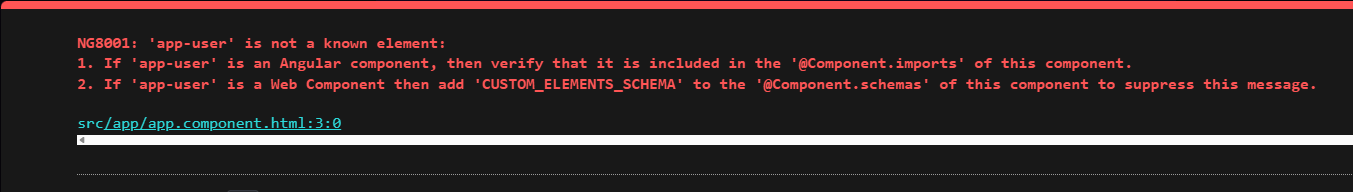
On larger screens:

* The layout becomes **two-column**.
* User list becomes vertical.
* #fallback gets larger and aligned left.

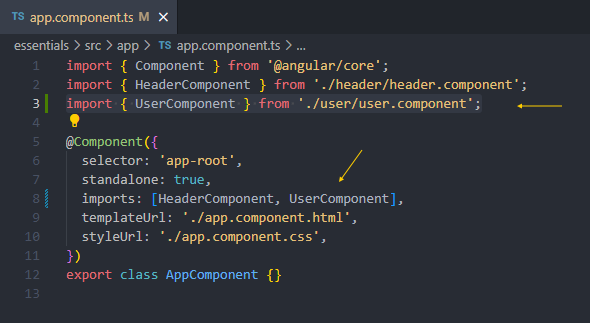


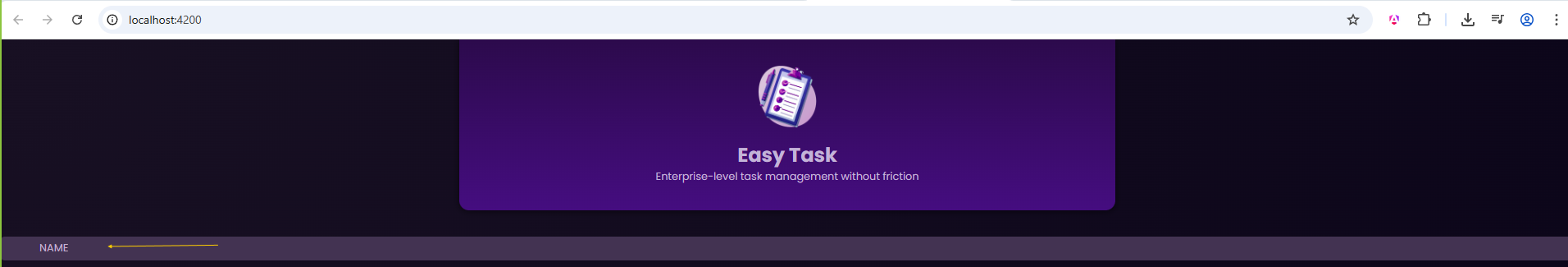


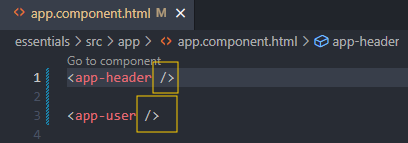


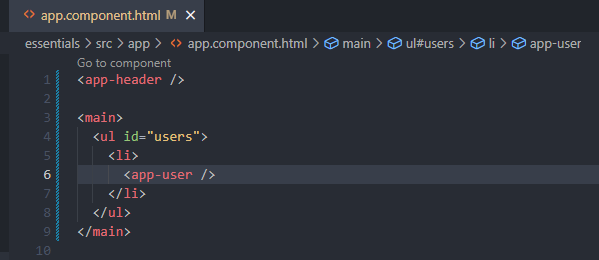


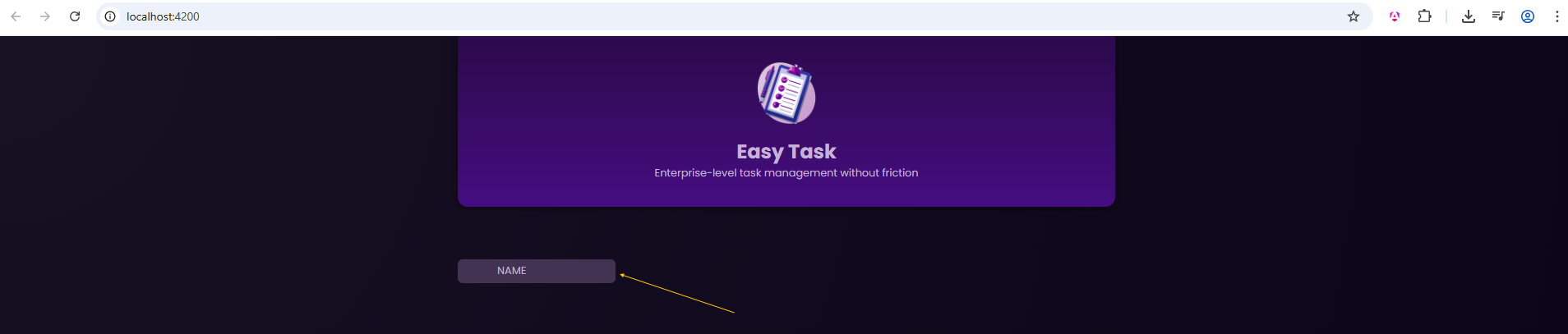












**🔄 What's Next?**

We'll now:

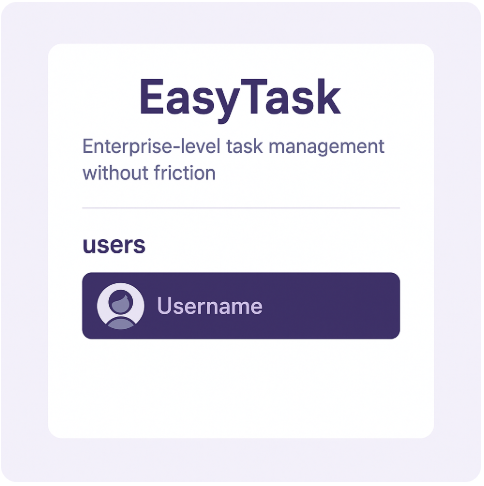
* Dynamically set the **image path** and **user name**.
* Bind properties using Angular syntax.
* Introduce **@Input()** for reusability of the UserComponent.

Let me know if you'd like to:

* See a visual HTML mockup of the current rendered result.
* Move on to binding user data using @Input.
* Add click events for interaction.

Would you like to proceed with data binding now?

See a visual HTML mockup of the current rendered result



Here’s a summary of your current project structure based on the uploaded src.zip file:

**✅ Core Files**

* src/index.html
* src/main.ts
* src/styles.css (global styles)

**✅ App Structure**

src/

├── app/

│ ├── app.component.css

│ ├── app.component.html

│ ├── app.component.ts

│ ├── header/

│ │ ├── header.component.css

│ │ ├── header.component.html

│ │ └── header.component.ts

│ └── user/

│ ├── user.component.css

│ ├── user.component.html

│ ├── user.component.spec.ts 👈 can be removed if not testing yet

│ └── user.component.ts

├── assets/

│ ├── angular-logo.png

│ └── task-management-logo.png

Everything looks **correct and well-organized**:

* Your components are each placed in their own folders (header, user).
* The styling files follow Angular conventions.
* Image assets are in the correct location and named clearly.

**Next Steps**

To continue:

1. Add image source and user name binding logic inside user.component.ts.
2. Update the user.component.html to use the image and span correctly.
3. Optionally remove user.component.spec.ts if you’re not testing right now.

Would you like me to:

* Show the current full content of a specific file (e.g., user.component.html)?
* Or proceed with updating the user.component.ts + user.component.html with realistic user data and image logic?

Let me know!

# Preparing User Data (To Output Dynamic Content)

Great! Based on this transcript from Maximilian Schwarzmüller's Angular course, here's how to **continue your app** with the discussed features, including integration of the dummy.users.ts file, random user selection, and dynamic rendering of content using Angular features like property binding and interpolation.

**✅ What This Lesson Covers**

* Adding the dummy.users.ts file
* Storing image assets in the correct location
* Dynamically selecting a random user
* Rendering the user image and name dynamically using:
  + {{ }} interpolation
  + [src] property binding

**🗂 Step-by-Step Integration**

**1. ✅ Add dummy.users.ts in /src/app/**

Create the file dummy.users.ts inside src/app/ with the following content:

export const DUMMY\_USERS = [

{

id: 'u1',

name: 'Max',

image: 'user-1.jpg'

},

{

id: 'u2',

name: 'Anna',

image: 'user-2.jpg'

},

{

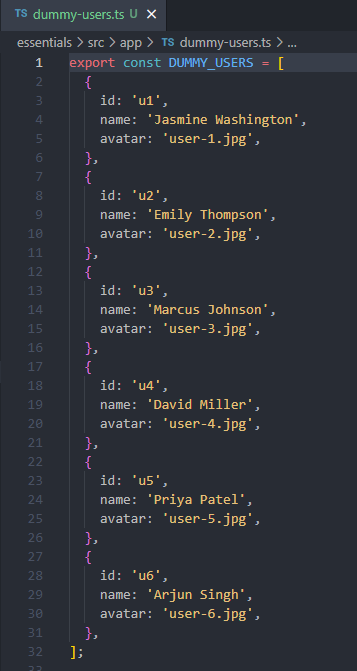
id: 'u3',

name: 'Chris',

image: 'user-3.jpg'

}

];



📁 These image files (user-1.jpg, user-2.jpg, user-3.jpg) must be placed in src/assets/users/.

**2. ✅ Extract the User Images**

Unzip and copy all images from the provided users.zip into:

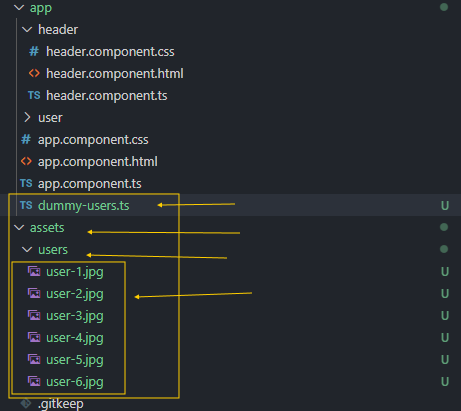
src/assets/users/

You should now have:

src/assets/users/user-1.jpg

src/assets/users/user-2.jpg

src/assets/users/user-3.jpg



**3. ✅ Update user.component.ts**

Import the dummy users and randomly pick one on component initialization:

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

import { DUMMY\_USERS } from '../dummy.users';

@Component({

selector: 'app-user',

templateUrl: './user.component.html',

styleUrls: ['./user.component.css'],

standalone: true

})

export class UserComponent {

user = DUMMY\_USERS[Math.floor(Math.random() \* DUMMY\_USERS.length)];

}

**4. ✅ Update user.component.html**

Update the markup to use Angular’s **interpolation** ({{ }}) and **property binding** ([src]):

<div>

<button>

<img [src]="'assets/users/' + user.image" alt="{{ user.name }}" />

<span>{{ user.name }}</span>

</button>

</div>

**✅ Explanation of Angular Features**

| **Feature** | **What it does** | **Example** |
| --- | --- | --- |
| {{ user.name }} | Interpolates the user’s name into the DOM | <span>{{ user.name }}</span> |
| [src] | Dynamically sets the src attribute of the image | <img [src]="'assets/users/' + user.image" /> |

**🧠 Visual Summary**

Imagine that the randomly picked user is:

{ id: 'u2', name: 'Anna', image: 'user-2.jpg' }

Rendered output becomes:

<img src="assets/users/user-2.jpg" alt="Anna" />

<span>Anna</span>

**🖼 CSS Visual Aid**

If you want me to generate an image showing how the user.component.css styles apply to this layout (especially how the button, image, and span appear), let me know — I can illustrate it.

**✅ What’s Next?**

In the next lesson, Maximilian will likely explain *how Angular detects changes*, and possibly move toward *handling events* like button clicks or expanding the app to *multiple users*.

When you're ready, go ahead and upload or paste the **next transcript**, and I’ll help you build on this!

Great! Now that you’ve uploaded:

1. ✅ dummy-users.ts – containing an array of user data
2. ✅ users.zip – containing user profile images to be extracted into src/assets/users/

We’re ready to continue implementing dynamic content in the UserComponent.

**🎯 Goal**

Render a **random user** from the dummy-users.ts file by dynamically displaying:

* Their **name**
* Their **profile image** (from assets/users/)

**✅ Step-by-Step Integration Plan**

**1. ✔ Add the Dummy Users File**

Move dummy-users.ts into your src/app/ folder.

Example dummy-users.ts content:

export const DUMMY\_USERS = [

{ id: 'u1', name: 'Daniel', image: 'u1.jpg' },

{ id: 'u2', name: 'Sarah', image: 'u2.jpg' },

{ id: 'u3', name: 'Max', image: 'u3.jpg' }

];

**2. ✔ Extract Images**

* Unzip users.zip
* Copy all image files (e.g., u1.jpg, u2.jpg) into:
* src/assets/users/

Make sure your angular.json has:

"assets": [

"src/favicon.ico",

"src/assets"

]

**3. 🧠 Update user.component.ts**

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

import { DUMMY\_USERS } from '../dummy-users';

@Component({

selector: 'app-user',

templateUrl: './user.component.html',

styleUrls: ['./user.component.css'],

standalone: true,

})

export class UserComponent {

user = this.getRandomUser();

private getRandomUser() {

const randomIndex = Math.floor(Math.random() \* DUMMY\_USERS.length);

return DUMMY\_USERS[randomIndex];

}

}

**4. 🖼️ Update user.component.html**

<div>

<button>

<img [src]="'assets/users/' + user.image" [alt]="user.name" />

<span>{{ user.name }}</span>

</button>

</div>

**🔍 Angular Concepts Introduced:**

* **Property binding** with [src] and [alt]
* **Interpolation** using {{ user.name }}

# Storing Data in a Component Class

Thank you! Here's a cleaned-up and structured version of **Lecture 20: *Storing Data in a Component Class***, which you can directly insert into your Word document. I’ve corrected grammar, spelling, and phrasing for clarity, and I also included the associated code block for your UserComponent. Let’s break it down step-by-step:

**📘 Lecture 20: Storing Data in a Component Class**

To output **dynamic user data**, we first need to **draw a random user** from our data source. To do this, we now write our first piece of logic inside the UserComponent class body.

Until now, we haven't had the need to place any code inside a component class body because we were dealing only with static content. But now that we want to render different users dynamically, we'll need to store and use component-level data.

**🧠 Step 1: Define a Property in the Component Class**

When using TypeScript in Angular, you can define a property (or "variable") directly in the class. You do **not** prefix it with let or const. Angular automatically makes such properties available in the associated template.

selectedUser = DUMMY\_USERS[0];

But we don’t want to just always select the first user. We want a **random user** each time.

**🧮 Step 2: Draw a Random User**

First, make sure to **import the user data array** from dummy-users.ts. Since the file is one level up from the user folder, use a relative path.

import { DUMMY\_USERS } from '../../dummy-users';

Then we create a randomIndex constant (outside the class) and use it to set the selectedUser inside the class.

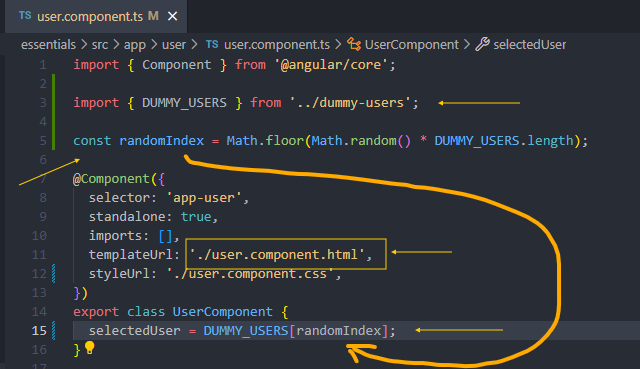
const randomIndex = Math.floor(Math.random() \* DUMMY\_USERS.length);

export class UserComponent {

selectedUser = DUMMY\_USERS[randomIndex];

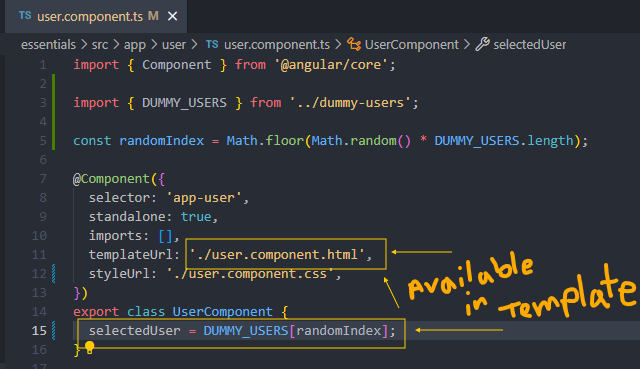
}

This ensures a **random user** is selected from the array when the component is initialized.



**🧩 Step 3: Access Class Properties in the Template**

All public class properties defined in the component are **automatically available in the component’s template**. That’s one of Angular’s core features!



We can now **bind** this property in the user.component.html template to dynamically render the user’s image and name.

Here’s what your template might look like:

<div>

<button>

<img [src]="'assets/users/' + selectedUser.image" alt="User photo" />

<span>{{ selectedUser.name }}</span>

</button>

</div>

* {{ selectedUser.name }} uses **interpolation** to display the user’s name.
* [src]="'assets/users/' + selectedUser.image" uses **property binding** to dynamically build the image path.

✅ This is the first time we're outputting **dynamic content** in Angular — a huge milestone in learning component-based development!

**📄 Full TypeScript Code for user.component.ts**

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

import { DUMMY\_USERS } from '../../dummy-users';

const randomIndex = Math.floor(Math.random() \* DUMMY\_USERS.length);

@Component({

selector: 'app-user',

templateUrl: './user.component.html',

styleUrls: ['./user.component.css'],

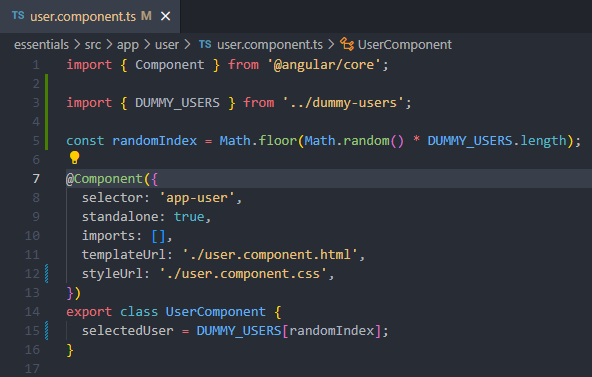
standalone: true

})

export class UserComponent {

selectedUser = DUMMY\_USERS[randomIndex];

}



# Outputting Dynamic Content with String Interpolation

**📘 Lecture 21: Outputting Dynamic Content with String Interpolation**

**🎯 Goal**

Learn how to **bind data from your component class to the HTML template** using Angular's **string interpolation** syntax ({{ }}), and output the randomly selected user’s name.

**🧠 Concept Summary**

In Angular, when you want to display **dynamic data** from your component in the HTML template, you can use:

**✅ String Interpolation**

This technique uses **double curly braces** to insert dynamic values into your HTML.

{{ someProperty }}

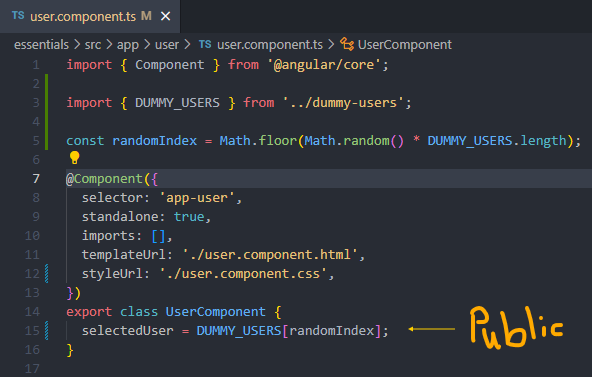
For example:

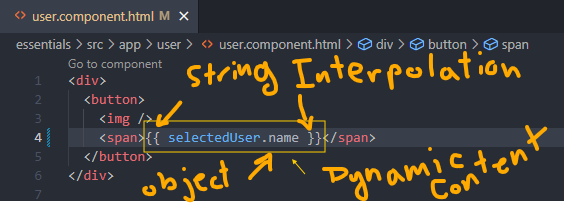
{{ selectedUser.name }}

Here, selectedUser is a class property, and .name accesses the name value.

⚠️ Note:

* The property must be **public** (by default, all properties are public unless marked private or protected).
* You can access nested object fields like selectedUser.avatar.





**💻 Code Integration**

**1️⃣ user.component.ts**

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

import { DUMMY\_USERS } from '../../dummy-users';

const randomIndex = Math.floor(Math.random() \* DUMMY\_USERS.length);

@Component({

selector: 'app-user',

templateUrl: './user.component.html',

styleUrls: ['./user.component.css'],

standalone: true,

})

export class UserComponent {

selectedUser = DUMMY\_USERS[randomIndex];

}

**2️⃣ user.component.html**

Update your HTML file to include the name dynamically:

<div>

<button>

<img [src]="'assets/users/' + selectedUser.avatar" alt="User Avatar" />

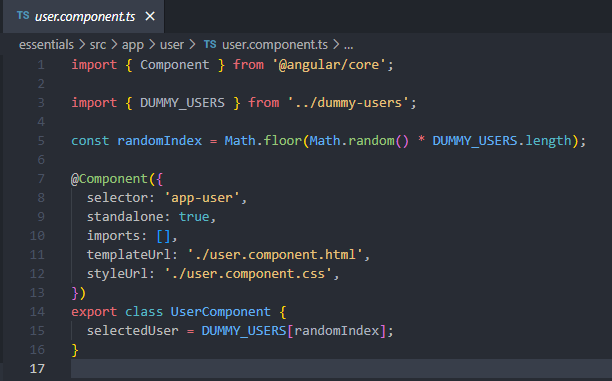
<span>{{ selectedUser.name }}</span>

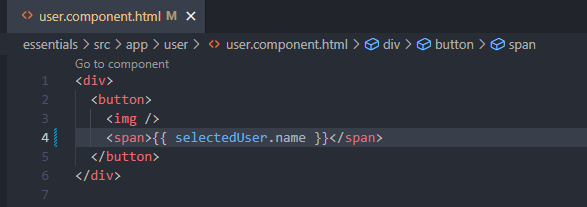
</button>

</div>

Here’s what’s happening:

* {{ selectedUser.name }} uses string interpolation to output the name.
* [src]="'assets/users/' + selectedUser.avatar" uses **property binding** to dynamically generate the image path.





**🧾 Visual Explanation: String Interpolation**

Here's how Angular processes:

<span>{{ selectedUser.name }}</span>

📦 In user.component.ts:

selectedUser = { name: 'Jane Doe', avatar: 'jane.png', id: 'u1' };

🧠 Angular replaces:

<span>{{ selectedUser.name }}</span>

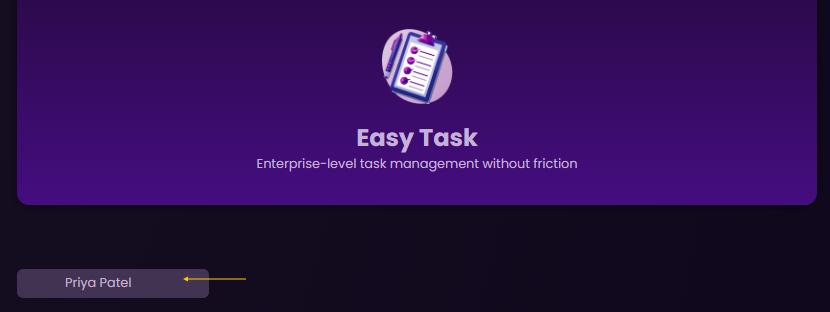
➡️ With:

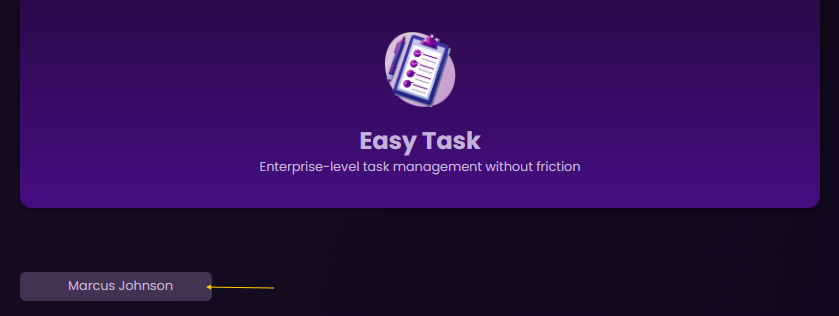
<span>Jane Doe</span>

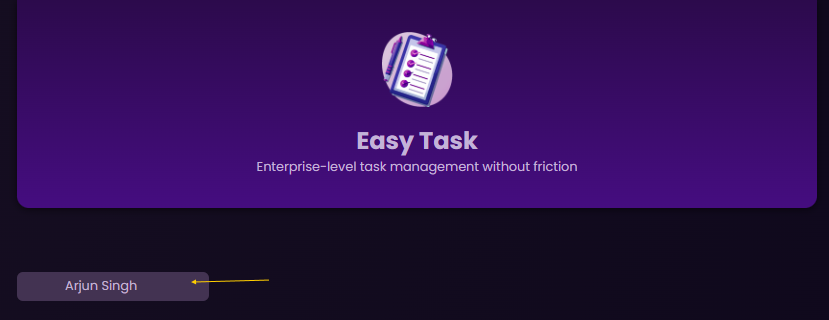
**✅ Expected Behavior**

Every time you reload the app:

* A **different user's name and avatar** are shown.
* The avatar image should load from assets/users/username.png.







# Property Binding & Outputting Computed Values

**🎓 Lecture 21: Property Binding & Outputting Computed Values**

When it comes to outputting dynamic content in Angular templates, **string interpolation** with double curly braces ({{ }}) is a fundamental feature. However, there's another **essential Angular feature**: **property binding**.

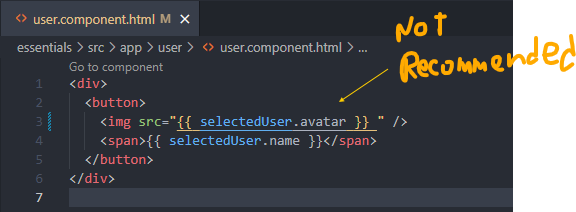
**✅ Why Property Binding?**

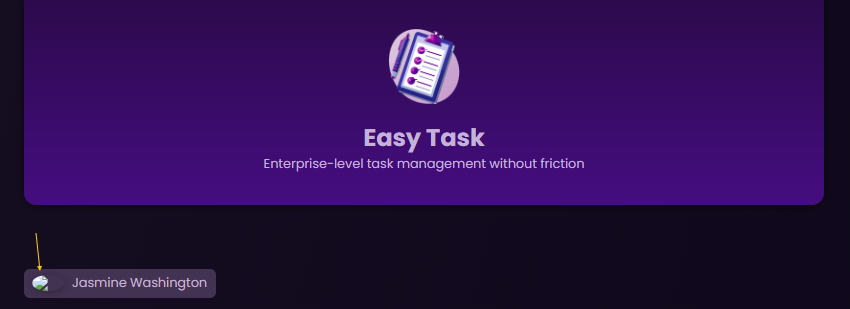
Let’s say we want to display a user’s image. We need to set the src attribute of an <img> element to point to the user's image path.

Initially, you **could** do this:

<img src="{{ selectedUser.avatar }}">

This uses **string interpolation** *within an attribute*, and yes — this works.





🔍 **However**, it’s **not the recommended approach** for attributes like src, alt, or other DOM properties.

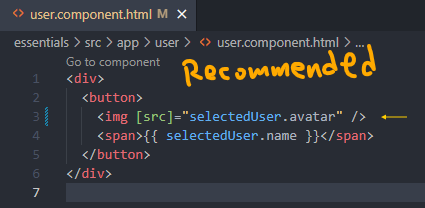
Instead, Angular provides a better solution:

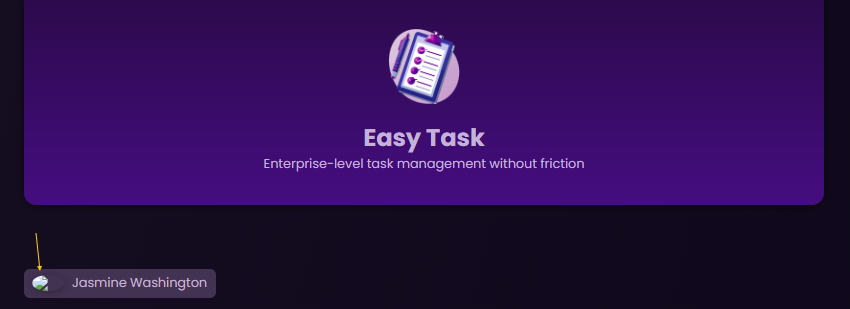
**🔧 Property Binding Syntax**

The **property binding** syntax looks like this:

<img [src]="value">

You wrap the **property name** (e.g., src) in **square brackets** [], and assign the **component expression** on the right without {{ }}.



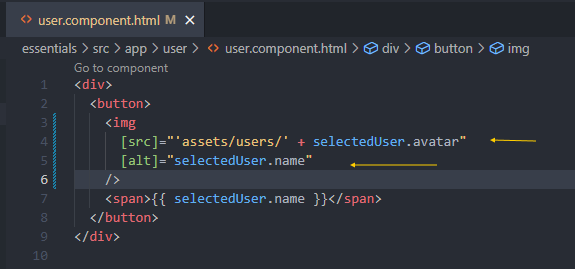


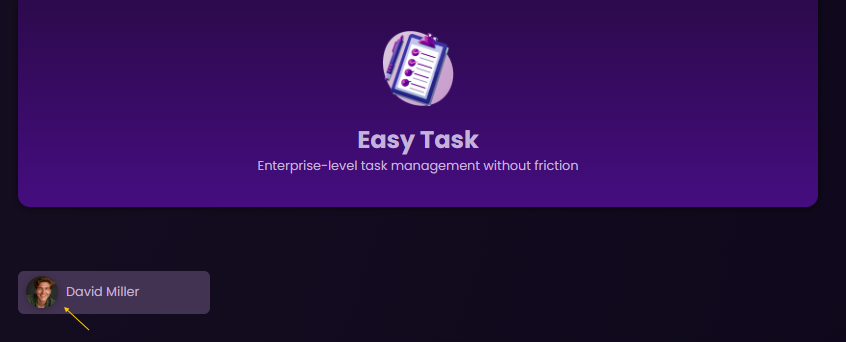
**🛠 Binding src and Constructing Dynamic Paths**

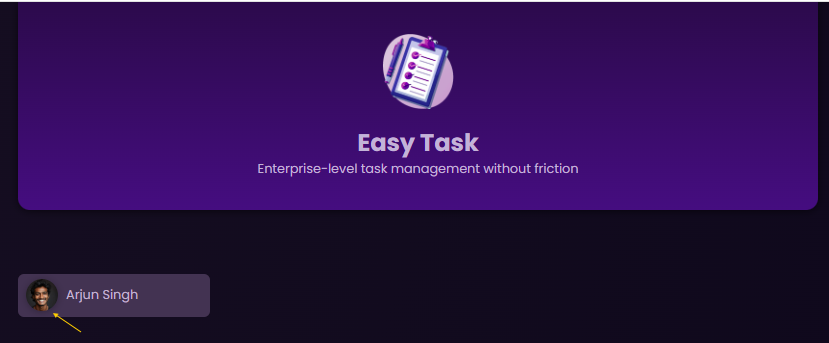
Since your user images are in the assets/users/ folder, simply binding selectedUser.avatar won’t work. You need to **concatenate** a full path dynamically:

<img [src]="'assets/users/' + selectedUser.avatar" [alt]="selectedUser.name">

* The first binding dynamically sets the image path.
* The second binds the user's name to the image’s alt text.
* You can also format the code for readability using your IDE shortcut.







**💡 Tip: Using Expressions**

Just like in string interpolation, property binding supports **full JavaScript expressions** (excluding assignments). You can write:

<p>{{ 1 + 1 }}</p>

<p>{{ selectedUser.name.toUpperCase() }}</p>

**🖼 Result: What Should You See?**

Once you set up your selectedUser logic in the component class and bind the image and name using Angular’s template syntax, the app will:

* Show a **random user’s name** and **avatar**.
* Change these on every page reload (thanks to the random user logic).
* Maintain **clean and readable** Angular template code.

✅ **Summary: Key Takeaways**

| **Feature** | **Use Case** | **Syntax Example** |
| --- | --- | --- |
| String Interpolation | Inline content between HTML tags | <p>{{ selectedUser.name }}</p> |
| Property Binding | Set DOM element properties like src, alt | <img [src]="'assets/users/' + selectedUser.avatar"> |

Would you like me to show a **mock HTML preview** of the rendered component now? Or shall we continue with the **next lecture transcript** (e.g., *Adding Custom Types*)?

# Attribute Binding

**Attribute Binding**

In the previous lecture, you were introduced to **"Property Binding"** - a key Angular feature that allows you to bind element properties to dynamic values.

For example, <img [src]="someSrc"> binds the src property of the underlying [HTMLImageElement](https://developer.mozilla.org/en-US/docs/Web/API/HTMLImageElement) DOM object to the value stored in someSrc.

Whilst it might look like you're binding the src attribute of the <img> tag, you're actually NOT doing that. Instead, property binding really targets the underlying DOM object property (in this case a property that's also called src) and binds that.

This might look like a subtle detail (and often it indeed doesn't matter) but it's important to understand this difference between element attributes and property. [This article](https://jakearchibald.com/2024/attributes-vs-properties/) can help with understanding this difference.

Whilst it won't make a difference in Angular apps in many cases, it DOES matter if you're trying to set attributes on elements dynamically. Attributes which don't have an equally-named underlying property.

For example, when binding [ARIA attributes](https://developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA), you can't target an underlying DOM object property.

Since "Property Binding" wants to target properties (and not attributes), that can be a problem. That's why Angular offers a slight variation of the "Property Binding" syntax that does allow you to bind attributes to dynamic values.

It looks like this:

1. <div
2. role="progressbar"
3. [attr.aria-valuenow]="currentVal"
4. [attr.aria-valuemax]="maxVal">...</div>

By adding attr in front of the attribute name you want to bind dynamically, you're *"telling"* Angular that it shouldn't try to find a property with the specified name but instead bind the respective attribute - in the example above, the aria-valuenow and aria-valuemax attributes would be bound dynamically.

# Using Getters for Computed Values

**🎓 Lecture 22: Using Getters for Computed Values**

Now that you've learned how to output dynamic data in Angular templates using **string interpolation** ({{ }}) and **property binding** ([property]="value"), it's time to explore a **best practice** for handling computed values.

**❓ The Problem: Complex Template Expressions**

You might often need to **compute a value** before outputting it — such as concatenating a string for an image path:

<img [src]="'assets/users/' + selectedUser.avatar">

This works fine, but embedding logic like this **directly in the template** can make the code:

* Harder to read
* Harder to maintain
* Harder to test



**✅ The Solution: Use a Getter**

To keep templates clean, Angular developers often **move computed logic into the component class** using a **getter**.

**🧠 What Is a Getter?**

A **getter** is a special kind of method in JavaScript/TypeScript that acts like a property. You define it with the get keyword, and it’s accessed **without parentheses**.

**🧩 Updated Code in the Component Class**

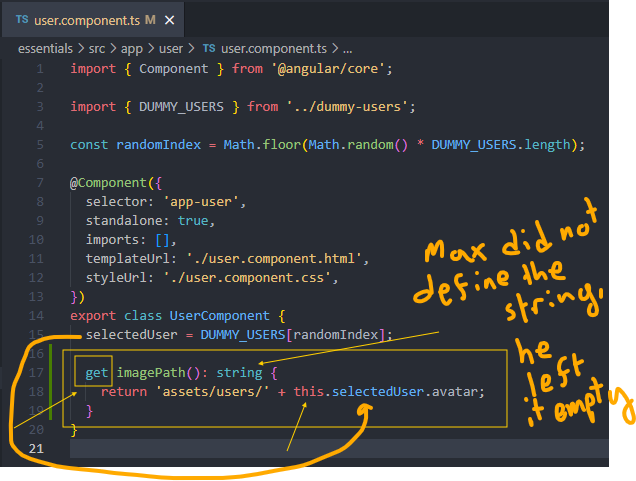
Here's how you would create a getter in your UserComponent:

get imagePath(): string {

return 'assets/users/' + this.selectedUser.avatar;

}

* imagePath is the name of the getter.
* Inside the method, we **use this.selectedUser.avatar** because we are inside the class body and need to reference the property correctly.

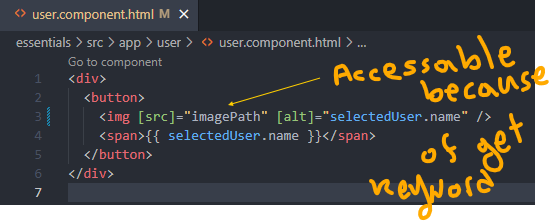


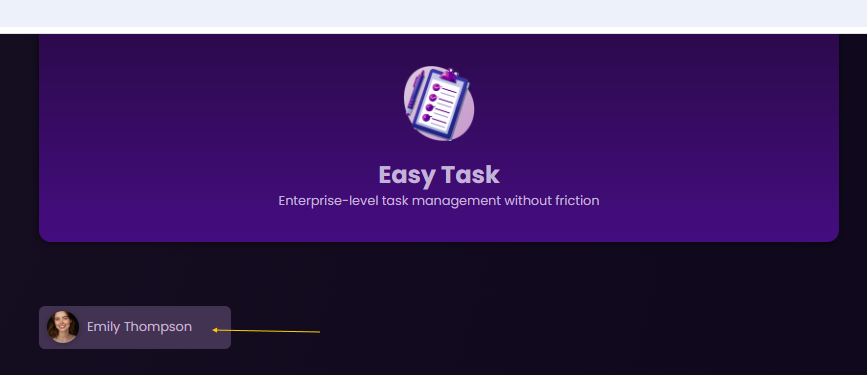
**📥 Updating the Template**

You can now clean up your template like this:

<img [src]="imagePath" [alt]="selectedUser.name">

* No need to construct the path directly in the template.
* The template now only binds to clean, readable values.
* The image path is now **computed in a central place** in the component logic.





**✅ Summary: When & Why to Use Getters**

| **❓ Use Case** | **✅ Benefit** |
| --- | --- |
| Need to compute a value from one or more properties | Keeps templates clean and readable |
| Repeating logic in template | Centralizes logic for reuse |
| Improves maintainability & testing | Keeps logic in TypeScript, not HTML |

**🏁 Final Output (After Refactor)**

Your user.component.ts:

export class UserComponent {

selectedUser = DUMMY\_USERS[randomIndex];

get imagePath(): string {

return 'assets/users/' + this.selectedUser.avatar;

}

}

Your user.component.html:

<img [src]="imagePath" [alt]="selectedUser.name">

<span>{{ selectedUser.name }}</span>

# Listening to Events with Event Binding

Perfect! Here's your cleaned-up and structured transcript for:

**🎓 Lecture 23: Listening to Events with Event Binding**

**🧭 Goal**

So far, we've output dynamic content using interpolation and property binding. Now it's time to **listen to user input**, which is crucial for building interactive web apps.

Let’s add an **event listener to a button** in the UserComponent and print a message to the console when it's clicked.

**🧪 Use Case: Handle Button Clicks**

We already have a button in our component template. Let’s attach a **click event listener** to it.

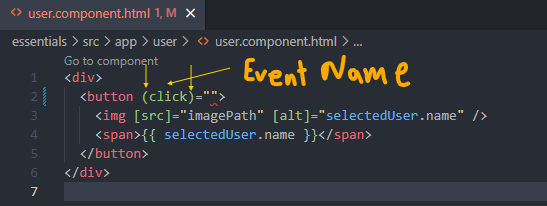
**🧩 Angular Syntax: Event Binding**

To **bind an event in Angular**, you:

1. Write the **event name** (e.g., click) **inside parentheses**
2. Set it equal to the code or function you want to run

**Example:**

<button (click)="onSelectUser()">Select</button>



**🛠 How It Works**

**Step 1: Add the Method to the Component Class**

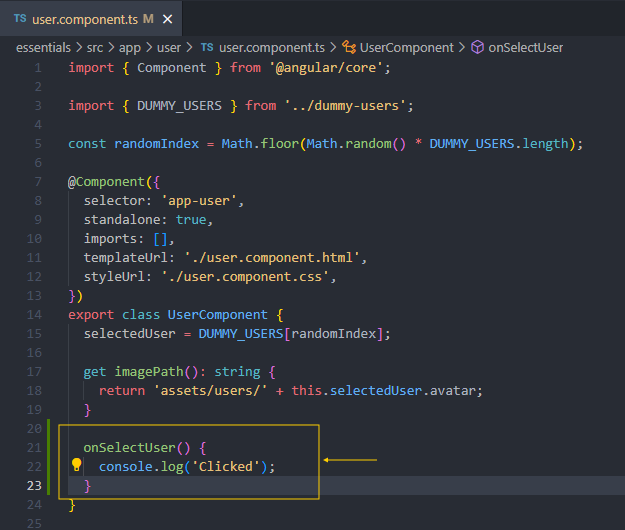
In your user.component.ts:

onSelectUser() {

console.log('Clicked');

}

* This is a regular TypeScript method.
* Naming convention: Many Angular developers prefix methods with on to show they are triggered by events (e.g., onClick, onSubmit). This is optional but helps readability.



**Step 2: Reference Method in Template**

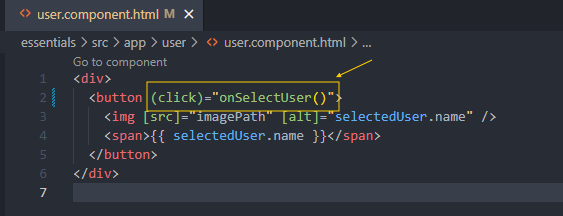
In your user.component.html:

<button (click)="onSelectUser()">

<!-- button content -->

</button>

* You must **include parentheses** to signal it’s a function call.
* Angular will call it **only when the event occurs**, **not immediately**.



**🧪 Bonus Tip: Event Auto-Completion**

If you're using **VS Code**, place your cursor between () after an element (e.g., <button (|)>) and press **Ctrl + Space** to get a list of supported events like:

* click
* mouseenter
* keydown
* blur
* submit

**🧪 Test It**

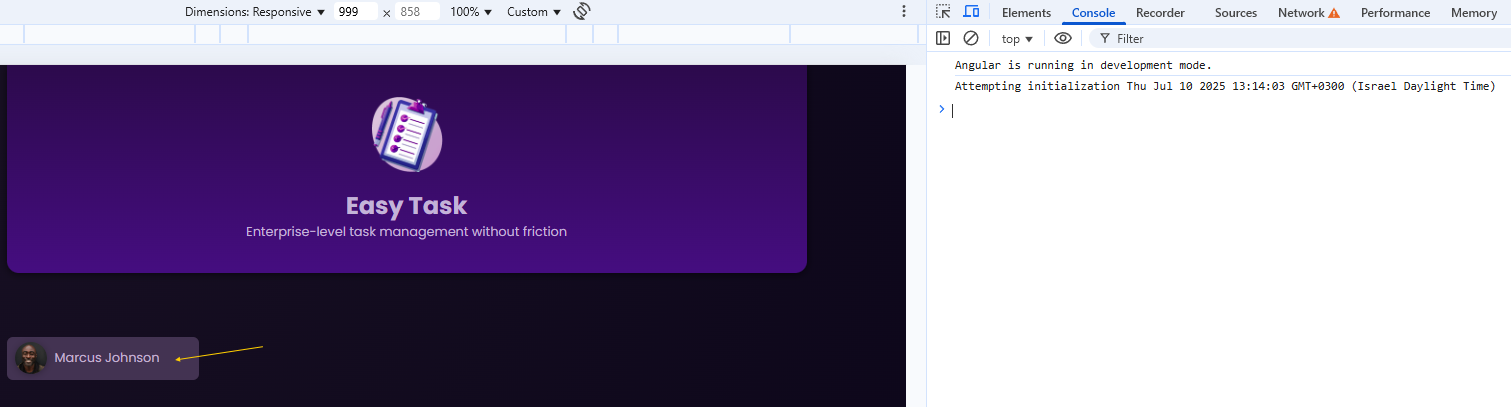
Open the browser's **DevTools Console** (F12 or Right-click > Inspect > Console) and:

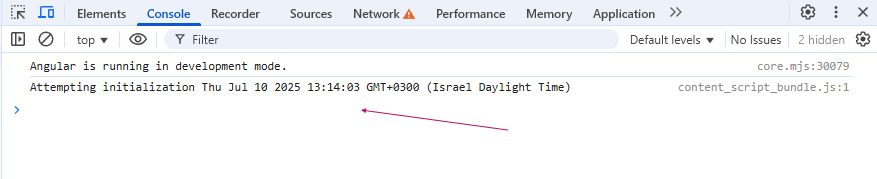
1. Click the button.
2. You should see Clicked logged each time.

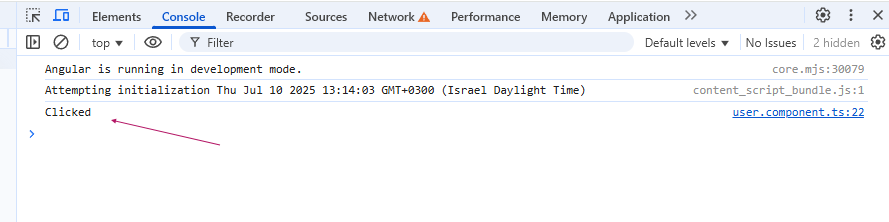
Clicked

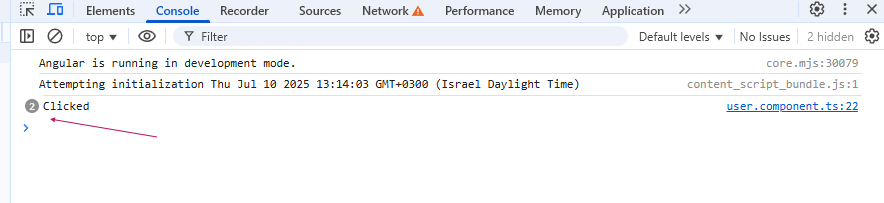
Clicked

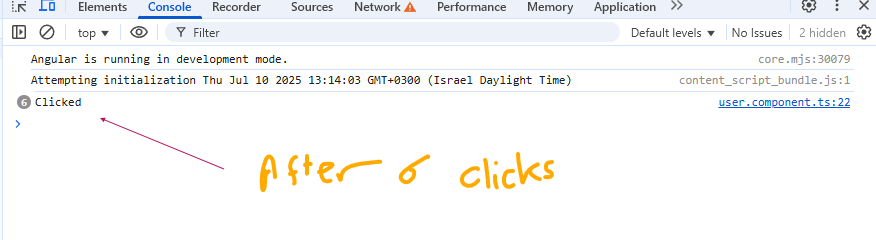
Clicked











**✅ Summary: Event Binding in Angular**

| **Feature** | **Syntax** | **Use it for...** |
| --- | --- | --- |
| Event Binding | (event)="handler()" | Responding to user input |
| Method in Class | onEventName() { ... } | Logic executed when event fires |

# Managing State & Changing Data

**🎓 Lecture 24: Managing State & Changing Data**

**🧭 Goal**

We now understand:

* How to **output dynamic data** (via interpolation/property binding)
* How to **listen to events** (via event binding)

It’s time to bring those two together to make the UI **react to user interaction** by **updating what’s visible on the screen**.

**🧩 Use Case: Update the Displayed User on Click**

So far:

* A **random user** is chosen **once** on component initialization.
* Every click **logs "Clicked"** in the console.

Now:  
✅ Let’s **update the displayed user dynamically** when the button is clicked.

**💡 Concept: State in Angular**

The term **state** refers to **data held in a component class** that **impacts the UI**.

In Angular, managing state is simple:

* You don't need special APIs or hooks.
* Just assign new values to class properties — Angular takes care of updating the view.

**🛠 Implementation**

**1. Update the Event Handler**

In user.component.ts, modify the onSelectUser() method to **pick a new random user** instead of just logging to console.

onSelectUser() {

const randomIndex = Math.floor(Math.random() \* DUMMY\_USERS.length);

this.selectedUser = DUMMY\_USERS[randomIndex];

}

* This generates a **new random index** on **every click**.
* this.selectedUser is updated with the new user.
* Because Angular uses **change detection**, the UI automatically re-renders with the new data.

**⚙ Full Updated Code in user.component.ts (relevant parts)**

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

import { DUMMY\_USERS } from '../../dummy-users';

@Component({

selector: 'app-user',

templateUrl: './user.component.html',

styleUrls: ['./user.component.css'],

standalone: true

})

export class UserComponent {

selectedUser = DUMMY\_USERS[Math.floor(Math.random() \* DUMMY\_USERS.length)];

get imagePath() {

return 'assets/users/' + this.selectedUser.avatar;

}

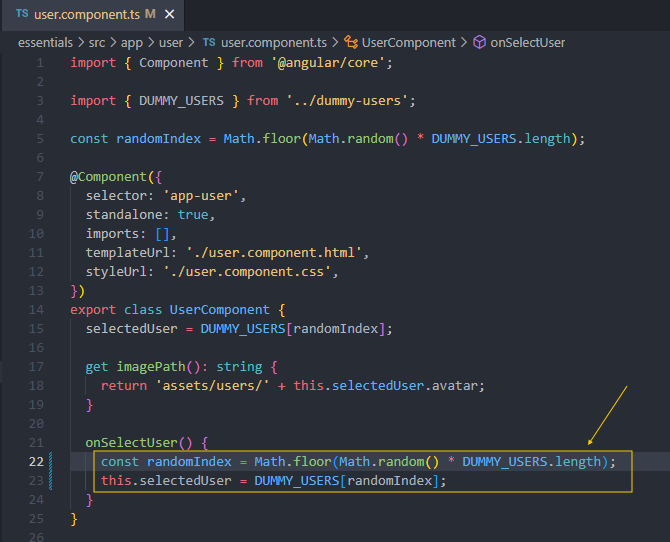
onSelectUser() {

const randomIndex = Math.floor(Math.random() \* DUMMY\_USERS.length);

this.selectedUser = DUMMY\_USERS[randomIndex];

}

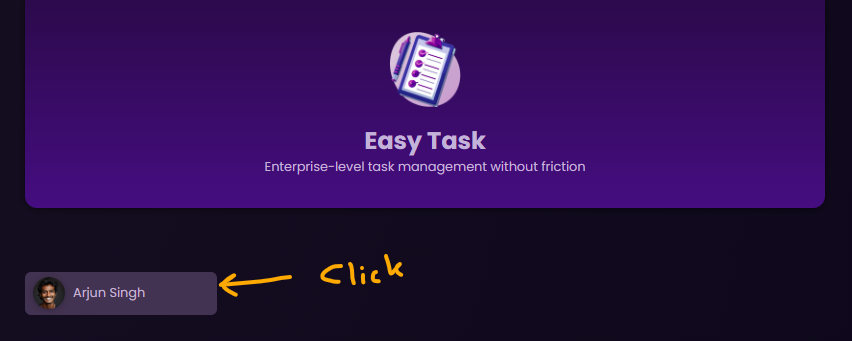
}

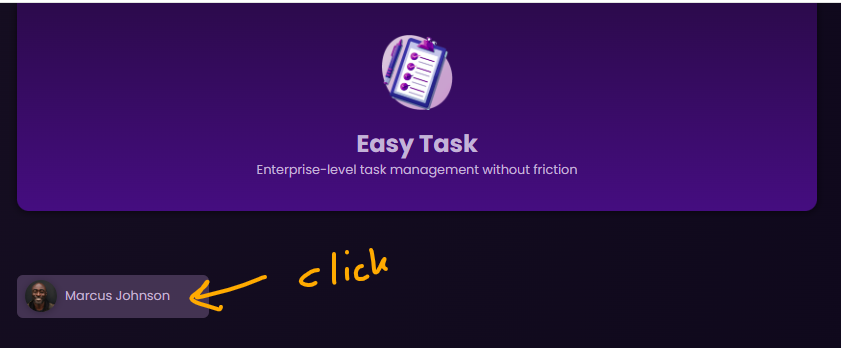


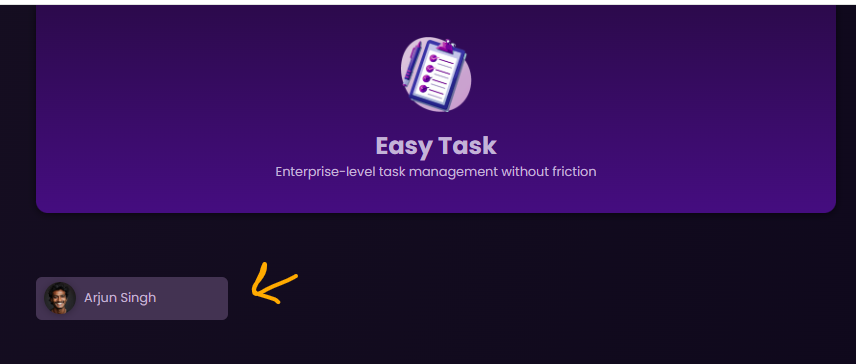
**🎯 Result**

* Reload the app
* Click the button repeatedly

✅ The **user image** and **name** update with every click!







**🧠 Summary**

| **Feature** | **Description** |
| --- | --- |
| State | Data stored in class properties that affect the UI |
| Change detection | Angular automatically updates the view when state changes |
| Interaction → Update | Events like clicks modify state → UI is refreshed |

Angular makes it seamless: **no boilerplate**, **no setup** — just update the data, and Angular handles the rest.

# A Look Behind the Scenes of Angular’s Change Detection Mechanism

**🎓 Lecture 25: A Look Behind the Scenes of Angular’s Change Detection Mechanism**

**🧩 Recap: What You Just Learned**

In the previous lecture:

* You changed **component state** (via this.selectedUser = ...)
* Angular **automatically updated the UI**

But **how** did that happen?

**🔍 What Is Angular Change Detection?**

Angular uses a **change detection mechanism** to:

* Watch for **changes in your component’s state**
* **Re-evaluate** the component’s **template**
* If the rendered DOM should change → **Angular updates the UI**

This is what enables Angular’s **reactive behavior**.

**⚙️ How Does Angular Know Something Changed?**

Angular leverages a powerful library called **zone.js** under the hood.

**✅ What does zone.js do?**

It:

* **Patches** browser APIs like setTimeout, DOM events, etc.
* **Listens** to:
  + User events (e.g., click, input)
  + Timers (e.g., setTimeout)
  + HTTP responses, etc.
* **Notifies Angular** whenever such events happen

As a result, Angular knows:

“Hey, something external just happened — time to check for changes!”

**🛠 Example Behind the Scenes:**

When you call:

this.selectedUser = DUMMY\_USERS[randomIndex];

Angular:

1. Sees that it happened **within a "zone"** (thanks to zone.js)
2. **Triggers change detection**
3. Evaluates the component's **template**
4. Finds the bound properties ({{ selectedUser.name }}, [src], etc.)
5. Sees that their values have changed
6. **Updates the DOM accordingly**

**🎩 Why It Feels Like Magic**

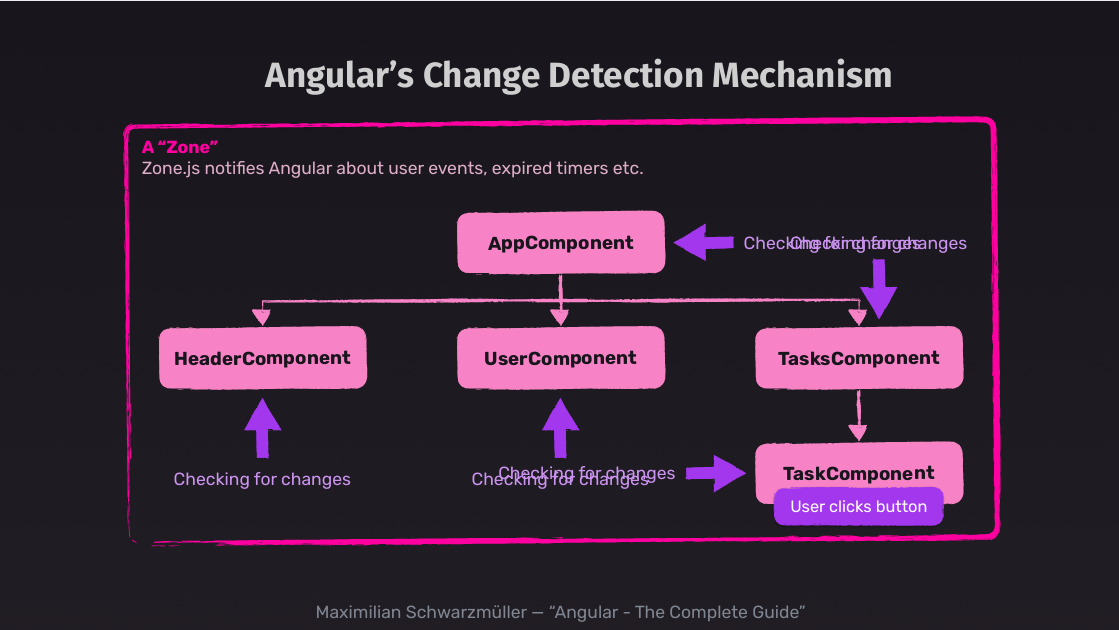
* You didn’t call any "render" or "update" function
* Angular **just knew** when to update the UI
* That’s **zone.js** + Angular’s **change detection** doing the work

**🤓 Important Notes**

| **Concept** | **Meaning** |
| --- | --- |
| zone.js | A library that intercepts async operations & notifies Angular |
| Change Detection | Angular's system to check for changed values and re-render templates |
| No manual triggers | You don’t need setState() like in React — Angular handles it for you |

**🧠 Summary**

* Angular’s change detection is what powers UI updates.
* It is **automatic**, thanks to **zone.js**.
* You simply change a value in your component → Angular **reacts**.



# Introducing Signals

Here is the cleaned-up, structured transcript for:

**🎓 Lecture: Introducing Signals**

**🔁 Recap**

You've learned how to:

* **Manage state** using standard class properties
* **React to changes** via Angular’s built-in **change detection** (powered by zone.js)

Now, it's time to learn a **modern** way of managing state in Angular:

**🔔 What Are Signals?**

**Signals** are a new state management mechanism introduced in **Angular 16+**.

They allow you to:

* Declare **reactive values**
* Automatically **track changes**
* Update the UI **more efficiently** (without relying on zone.js)

**🏗️ Setting Up a Signal**

To create a Signal:

import { signal } from '@angular/core';

selectedUser = signal(DUMMY\_USERS[randomIndex]);

✅ This replaces:

selectedUser = DUMMY\_USERS[randomIndex];

Think of a **Signal** like a **container** that:

* **Holds** a value
* **Notifies Angular** when that value changes

**🔄 Updating a Signal**

To update the signal:

this.selectedUser.set(DUMMY\_USERS[newRandomIndex]);

This **automatically triggers a UI update** for all template parts using the signal.

**📥 Reading from a Signal**

You **must read** from a Signal using function call syntax:

<span>{{ selectedUser().name }}</span>

This:

* Triggers **tracking**
* Sets up Angular’s internal subscription to this Signal
* Ensures Angular will rerender this span when the signal updates

**📊 Why Use Signals?**

| **Feature** | **Traditional State** | **Signal State** |
| --- | --- | --- |
| Triggers UI Update | zone.js-based event tracking | Direct dependency tracking |
| Update Detection | Broad & general | Fine-grained & specific |
| Syntax Simplicity | Easy | Slightly more verbose (.set(), ()) |
| Recomputing values | Manual via getters | Automatic via computed() |

**🧠 Creating Computed Values**

To create derived values with signals, use computed():

import { computed } from '@angular/core';

imagePath = computed(() => {

return 'assets/users/' + this.selectedUser().avatar;

});

Signals can **depend on other signals** and Angular tracks this dependency **automatically**.

You still access it in the template as a function:

<img [src]="imagePath()" />

**🧪 Summary**

* ✅ **Signals** are containers for reactive values
* 🔁 Use .set() to update
* 📥 Use () to read
* 🧠 Use computed() for derived values
* 🚀 More efficient than traditional zone.js detection
* ⚠️ **Angular 16+** required — older projects might not support them

**🧭 What’s Next?**

You’ll continue using **traditional state management** for the rest of this section, but:

* A **dedicated Signals section** is coming later in the course
* You’re now equipped to recognize and optionally use Signals in real-world apps

# We Need More Flexible Components!

Here is the cleaned-up and structured transcript for:

**🎓 Lecture: We Need More Flexible Components!**

**🧱 Recap & Motivation**

You've now learned:

* Core Angular concepts: components, state, binding, event handling, signals
* How to build dynamic and reactive UIs

✅ But the **demo app** so far only displays **one user**, which changes when clicked.

**🧩 The Goal**

We don’t want:

* A single user that changes on click

We **do want**:

* A **list of users**, each shown by a **reusable** UserComponent
* Clicking a user should **load their tasks**

**♻️ Reusable Components**

One of the **core strengths** of Angular components is **reusability**.

Examples:

| **Component** | **Reused?** |
| --- | --- |
| HeaderComponent | ❌ No – Used once |
| UserComponent | ✅ Yes – Used for each user |

**🛠️ Current Limitation**

Currently:

// In UserComponent

const randomIndex = ...;

selectedUser = DUMMY\_USERS[randomIndex];

This logic:

* Picks a **random user** on component load
* Executes **only once** – always picks same user per instance
* Doesn’t allow **custom user input per component**

Also:

* Clicking the user changes the selection → not needed anymore

**🧹 Cleanup Required**

To make UserComponent reusable:

1. ❌ **Remove**:
   * randomIndex
   * selectedUser (Signal or not)
   * onSelectUser() method
2. ❌ **Remove unused imports**:
   * signal, computed
   * DUMMY\_USERS
3. ✅ **Keep only** the template and styling logic

**🎯 The New Goal**

We want to be able to do this in the AppComponent:

<app-user [user]="someUser"></app-user>

Just like how you'd pass a value to a standard HTML element like:

<img [src]="imagePath">

So the component should:

* **Accept external input**
* **Display different users passed from the outside**

Angular supports this via **Input binding**, which will be covered in the **next lecture**.

**📌 Summary**

| **Step** | **Status** |
| --- | --- |
| Explained reusability need | ✅ |
| Cleaned up user component | ✅ |
| Highlighted next feature | 👉 Input Binding |

# Defining Component Inputs

Here is the structured and cleaned-up transcript for:

**🎓 Lecture: Defining Component Inputs**

**🧩 Goal**

To make components **more flexible and reusable**, we want to:

* Pass **data** into components from the **outside**.
* Replace hardcoded values with **dynamic inputs**.

**✅ Step 1: Define a Settable Property (Input)**

Inside the component that should accept data (e.g., UserComponent):

import { Input } from '@angular/core';

@Input() avatar: string;

🧠 Notes:

* Use the @Input() **decorator** above the property.
* Property must be declared **public** (default).
* TypeScript requires a type (: string) since there's no initializer.
* Add ! after the property name to **suppress TS error** about uninitialized value:
* @Input() avatar!: string;

**✅ Step 2: Use the Component with Property Binding**

Inside the parent component template (e.g., AppComponent):

<app-user [avatar]="users[0].avatar"></app-user>

You can repeat it for more users:

<app-user [avatar]="users[1].avatar"></app-user>

<app-user [avatar]="users[2].avatar"></app-user>

✅ We'll soon loop through users dynamically, but for now we do it manually.

**✅ Step 3: Make Dummy User Data Available**

Inside AppComponent.ts:

import { DUMMY\_USERS } from './dummy-users';

users = DUMMY\_USERS;

Now the users array is available in the template via property binding.

**✅ Step 4: Add More Inputs (e.g., Name)**

In UserComponent.ts:

@Input() name!: string;

Then in AppComponent.html:

<app-user

[avatar]="users[0].avatar"

[name]="users[0].name">

</app-user>

Repeat for the other users as needed.

**✅ Step 5: Reintroduce Image Path with a Getter**

In UserComponent.ts, reintroduce the computed path:

get imagePath(): string {

return 'assets/users/' + this.avatar;

}

This avoids hardcoding path logic in the template.

**✅ Step 6: Update the Template**

In UserComponent.html:

* **Remove** signal parentheses () – we’re no longer using Signals.
* Use the new inputs:

<img [src]="imagePath" [alt]="name" />

<h3>{{ name }}</h3>

💡 Keep event binding syntax like (click)="..." intact.

**✅ Final Result**

After saving and reloading:

* You see a **list of users**, each showing:
  + Their own name
  + Their own avatar
* The component is now **reusable** and **customizable** 🎉

# Required & Optional Inputs

Here’s the structured and developer-focused summary of the lecture:

**🎓 Lecture: Required & Optional Inputs**

**🧩 Problem Statement**

You’re using @Input() to make a component reusable, but:

* You're using **non-null assertion (!)** to tell TypeScript the input *will* be provided.
* **However**, there’s no guarantee at runtime — inputs can be **forgotten**.

❌ This can cause silent bugs, like missing user names or broken images.

**✅ Solution: Use @Input({ required: true })**

Angular 16+ introduces a required option in the @Input() decorator:

@Input({ required: true }) name!: string;

@Input({ required: true }) avatar!: string;

🔍 This does 2 things:

1. Keeps the type string (no string | undefined)
2. Ensures the Angular compiler **enforces the input at usage**

If a required input is missing:

<!-- ❌ Missing name -->

<app-user [avatar]="user.avatar"></app-user>

You’ll see:

Error: The required input 'name' must be specified.

✅ Your IDE and compiler will flag it — before runtime.

**💡 Why It Matters**

| **Without required: true** | **With required: true** |
| --- | --- |
| TypeScript believes it’s always there (because of !) | Angular **actually ensures** it's passed |
| You can forget to bind an input | You'll get compile-time or IDE errors |
| Bugs may only appear at runtime | Bugs are caught early during development |

**🆓 Optional Inputs**

If a prop should be optional:

@Input() imageBorder?: boolean;

No required: true, no !, and the type is string | undefined (or whatever).

**🧪 Pro Tip: Always Do This for Required Props**

Always use:

@Input({ required: true }) propName!: Type;

It’s cleaner, safer, and gives immediate feedback during development.

**🧠 Summary**

| **What you did** | **Why it helps** |
| --- | --- |
| Used @Input({ required: true }) | Enforces presence of critical inputs |
| Avoided silent runtime bugs | Compile-time + IDE enforcement |
| Improved dev-time safety | Faster feedback loop when wiring components |

# Using Signal Inputs

Here's a structured and developer-focused breakdown of the lecture:

**🎓 Lecture: Using Signal Inputs in Angular**

**🔄 Two Ways to Accept Inputs in Angular Components**

**1. Classic Decorator-based Approach**

@Input({ required: true }) name!: string;

@Input({ required: true }) avatar!: string;

**2. Modern Signal-based Approach (Angular 16+)**

import { input } from '@angular/core';

avatar = input<string>('');

name = input.required<string>();

☝️ Note: This input function is lowercase — **not** the same as @Input (decorator).

**🔍 Signal Inputs: Key Concepts**

| **Concept** | **Classic Inputs** | **Signal Inputs** |
| --- | --- | --- |
| Syntax | @Input() | input() / input.required() |
| Type safety | ! for required props | Uses TypeScript generics (e.g., input<string>) |
| Default value | via = | via input('default') |
| Required | @Input({ required: true }) | input.required<Type>() |
| Writable? | ✅ You can reassign | ❌ Signal inputs are **readonly** |
| Reading | {{ name }} | {{ name() }} in template |
| Change detection | Via Zone.js | Via signal tracking (faster & finer-grained) |

**✅ Declaring Signal Inputs**

import { input } from '@angular/core';

avatar = input.required<string>();

name = input.required<string>();

* Inputs are **Signal objects** now
* Signal **values** are accessed via function calls (name())

**⚠️ Signal Inputs Are Readonly**

You **cannot modify** a signal input from inside the component:

// ❌ This will throw an error

this.avatar.set('newAvatar.png'); // ❌ Not allowed

✅ You can **read** the value using:

const currentAvatar = this.avatar();

But they must be updated **from the parent component**.

**📄 Example Template Adjustments**

Old (classic inputs):

<p>{{ name }}</p>

<img [src]="imagePath" />

New (signal inputs):

<p>{{ name() }}</p>

<img [src]="imagePath()" />

**🧠 Derived (Computed) Signal Input Values**

Use computed() to derive values from signal inputs:

import { computed } from '@angular/core';

imagePath = computed(() => `assets/users/${this.avatar()}`);

* Angular tracks dependencies automatically
* Re-computes only when avatar() changes

**🧪 Testing Signal Inputs in Parent Template**

Nothing changes **outside** the component:

<app-user [name]="user.name" [avatar]="user.avatar"></app-user>

* The parent does **not** need to use signals
* Works seamlessly with classic data

**❓When to Use Signals for Inputs?**

| **Use Case** | **Recommendation** |
| --- | --- |
| Small projects | Stick with classic inputs |
| Performance-sensitive UI | Prefer signals (more efficient) |
| Codebase using Angular < 16 | Signals **not available** |
| Large enterprise teams | Often still use classic @Input() |

ℹ️ Signals are powerful but **still new**. Most existing projects use the decorator approach.

**🧼 Final Code Decision in This Section**

Maximilian reverts to the **classic @Input()** syntax for this section:

@Input({ required: true }) name!: string;

@Input({ required: true }) avatar!: string;

But you'll continue seeing **signals** used later in the course, especially in advanced scenarios.

**🧠 Summary**

| **Classic Input (@Input)** | **Signal Input (input())** |
| --- | --- |
| Familiar, widely used | Newer, more efficient |
| Writable from inside | Readonly |
| Requires ! or ? | No ! needed, value is wrapped |
| Zone.js change detection | Signal tracking mechanism |
| Used in most current projects | Emerging in newer projects |

# We Need Custom Events!

Here’s a structured and detailed breakdown of the lecture **"We Need Custom Events!"**:

**🎓 Lecture: We Need Custom Events!**

**🔁 Recap: What We Have So Far**

* We’ve **created reusable components** using @Input() to pass data **into** a component.
* Now, we need to pass **data out** of a component — i.e., **communicate upward** to a parent component.

**🧩 Why Do We Need Custom Events?**

Let’s consider the current setup:

<app-user></app-user>

Inside app-user, a user is displayed. But when the user is clicked:

* We **do not want to handle anything inside the user component**
* Instead, we want the **parent component (AppComponent) to react**, e.g., show tasks for the selected user

**🧭 Communication Flow: From Child to Parent**

| **Action** | **Component** |
| --- | --- |
| User clicks a button | UserComponent (child) |
| Parent reacts by showing tasks | AppComponent (parent) |

We need a way to **emit a custom event** from the child so the parent can listen to it.

**⚙️ Angular’s Solution: @Output() Decorator**

Just as @Input() lets a parent pass data **into** a component,  
@Output() lets a child **emit events** to the parent.

**🗨️ Real-World Analogy**

Imagine the UserComponent is a **button** on a remote.  
Clicking it sends a signal to the **TV** (AppComponent) to change channels.

We don’t care how the button looks —  
We just want it to **notify us when clicked**.

**🧪 The Problem**

The user component can’t make UI changes outside of itself.

So, we need to:

1. **Emit an event** when a user is clicked
2. **Catch** that event in the parent
3. **Act** (e.g., show tasks)

**🔜 Coming Next**

In the **next lecture**, we’ll learn how to:

* Use @Output() and EventEmitter
* Define and emit custom events
* Bind event handlers in the parent component template

**🧠 Summary**

| **Concept** | **Purpose** |
| --- | --- |
| @Input() | Pass data **from parent to child** |
| @Output() | Send events **from child to parent** |
| Goal | Enable component interaction and communication |

Inputs = data in ⬇️  
Outputs = events out ⬆️

# Working with Outputs & Emitting Data

Here’s a clear breakdown of the lecture **"Working with Outputs & Emitting Data"** from the Angular course:

**🎓 Lecture: Working with Outputs & Emitting Data**

**🔁 Context Recap**

* We already passed **data into** components using @Input().
* Now we’ll learn to pass **data out** of a component — from child to parent — using @Output().

**🎯 Goal**

When a **user is clicked** inside the UserComponent, we want the **AppComponent** to be notified and take action (like showing user tasks).

**🛠️ Step-by-Step Guide**

**1. Define a Custom Event**

In UserComponent:

@Output() select = new EventEmitter<string>();

* @Output() tells Angular this property is a custom event.
* EventEmitter<T> defines the type of data to emit — here, a string (the user ID).

**2. Emit the Event**

Inside the onSelectUser() method:

this.select.emit(this.id);

* this.id must be available as an input (@Input() id!: string;)
* This emits the user's ID to the parent component.

**3. Receive the Event in AppComponent**

In AppComponent, bind to the custom event using parentheses:

<app-user

[id]="users[0].id"

(select)="onSelectUser($event)"

></app-user>

* (select) is the name of the event defined in UserComponent.
* $event holds the emitted data (in this case, the user ID).

**4. Handle the Event**

In AppComponent.ts:

onSelectUser(id: string) {

console.log('Selected user with id', id);

}

* The parameter type is explicitly defined as string for TypeScript safety.

**💡 Tip**

You can add (select)="onSelectUser($event)" to every instance of <app-user> so that **each user emits their ID when clicked**.

**🧪 Demo Output**

When clicking user items:

Selected user with id u1

Selected user with id u2

...

✅ Custom events are working correctly!

**🧠 Summary**

| **Feature** | **Purpose** |
| --- | --- |
| @Output() | Expose a custom event to parent |
| EventEmitter | Emit values from child to parent |
| $event | Holds the emitted value in template |
| onSelectUser() | Handles the emitted event in parent |

Would you like to continue to the next lecture:  
**"Emitting Complex Data"**?