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**

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:

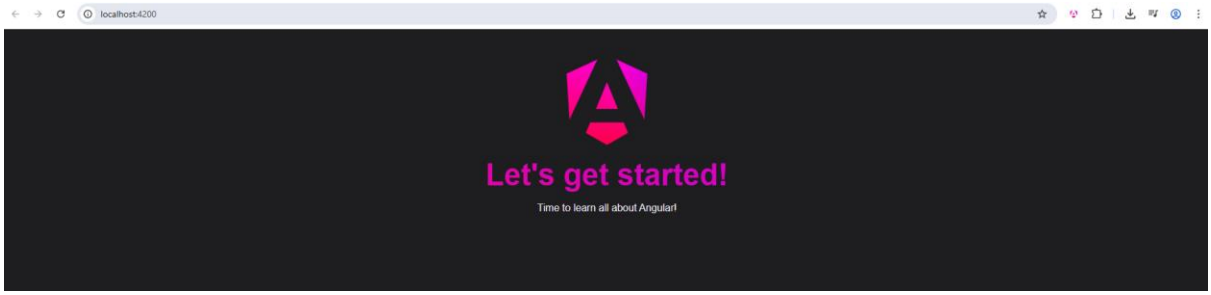
```
ng serve
```

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?

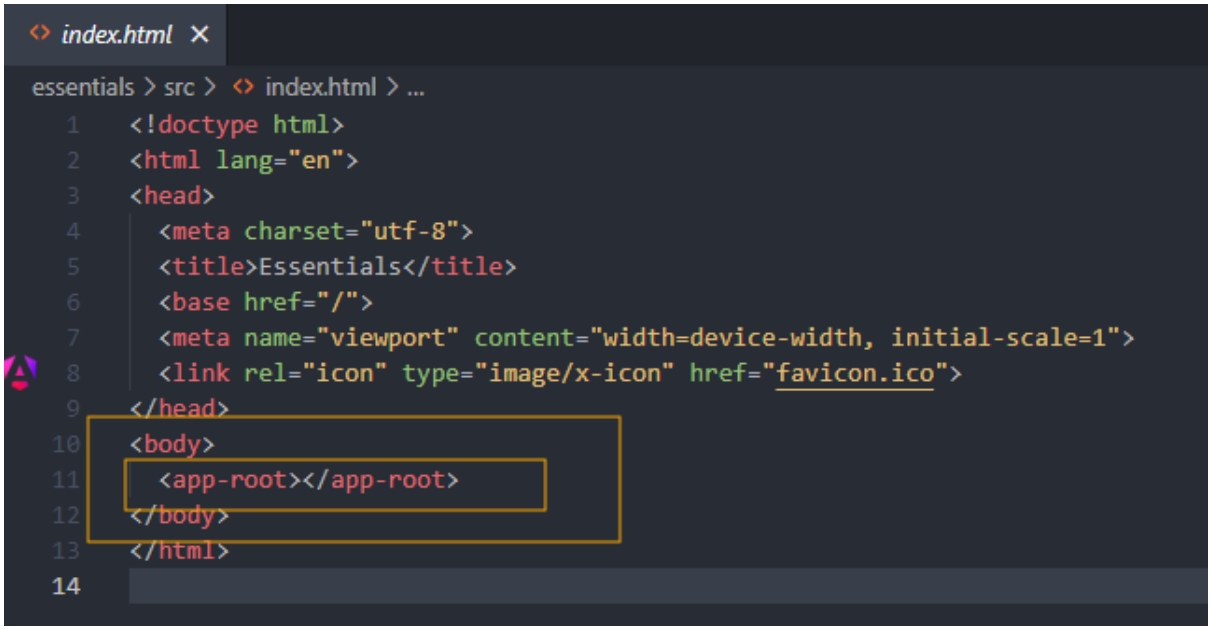
Step 1: The Role of `index.html`

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`

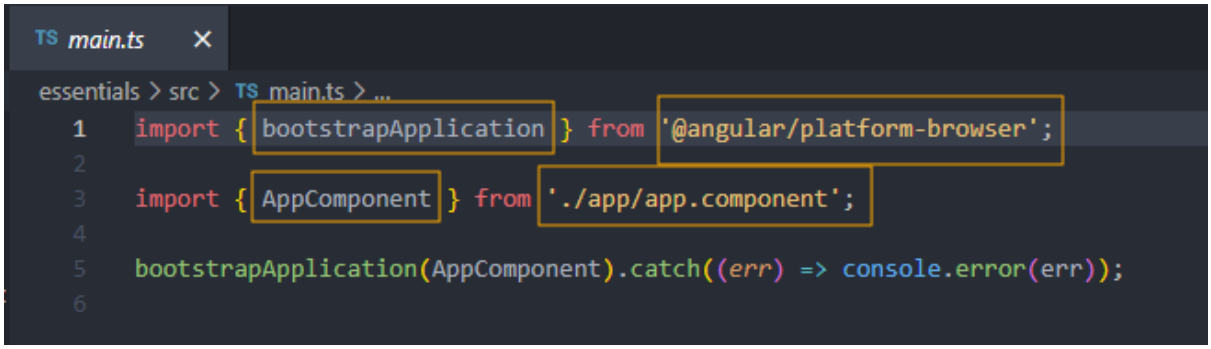
In `src/main.ts`, you'll find:

```
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

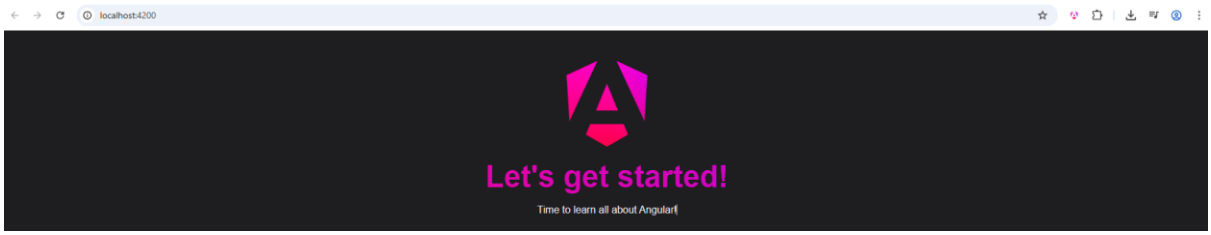


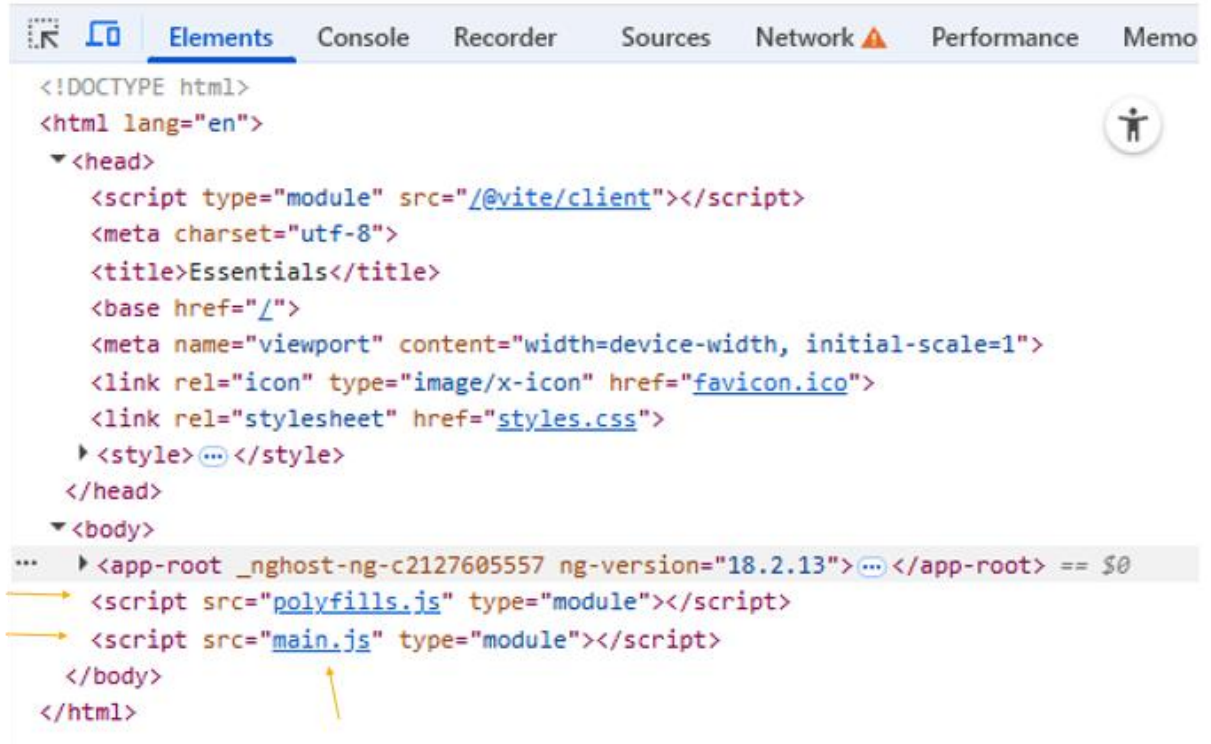
Step 3: Where's the Script Tag?

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

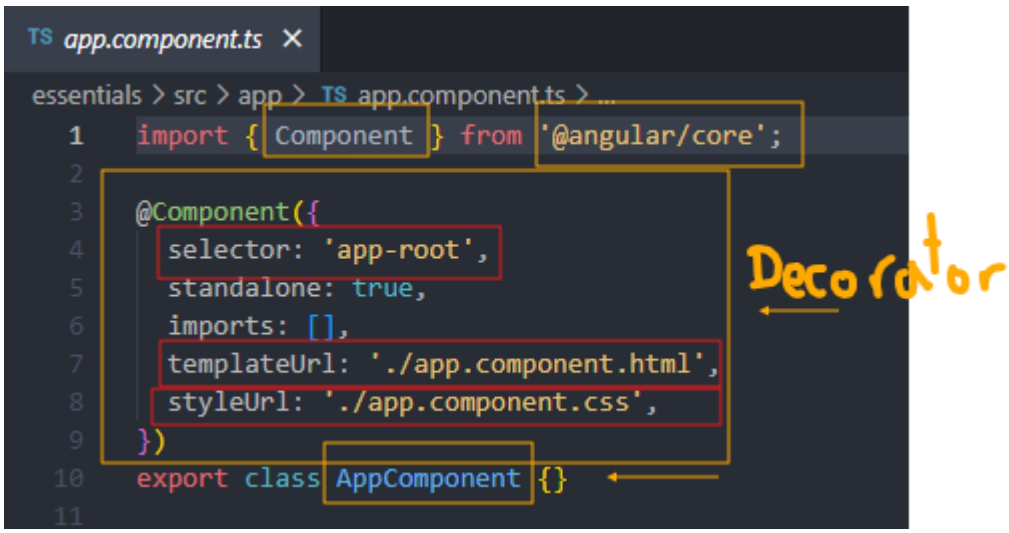
🔹 `selector: 'app-root'`

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

index.html

```
essentials > src > index.html > ...
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Essentials</title>
  <base href="/">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body>
  <app-root></app-root>
</body>
</html>
```

↓

Angular CLI injects JS → main.ts runs

main.ts

```
essentials > src > main.ts > ...
import { bootstrapApplication } from '@angular/platform-browser';

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

bootstrapApplication(AppComponent).catch((err) => console.error(err));
```

↓

bootstrapApplication(AppComponent)

app.component.ts

```
essentials > src > app > app.component.ts > ...
import { Component } from '@angular/core';

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

↓

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

index.html

```
essentials > src > index.html > ...
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Essentials</title>
  <base href="/">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body>
  <app-root></app-root>
</body>
</html>
```

↓

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

app.component.html

```
essentials > src > app > app.component.html > header
Go to component
<header>
  <img src="assets/angular-logo.png" alt="The Angular logo: The letter 'A'" />
  <h1>Let's get started!</h1>
  <p>Time to learn all about Angular!</p>
</header>
```

↓

Applies scoped styles from app.component.css

app.component.css 1

```
essentials > src > app > # app.component.css > header
header {
  margin: 3rem auto;
  text-align: center;
}

img {
  width: 8rem;
}

h1 {
  margin: 1rem auto;
  background: -webkit-linear-gradient(45deg, #f9096d, #b700ff);
  -webkit-background-clip: text;
  -webkit-text-fill-color: transparent;
  font-size: 3rem;
}
```

✅ 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.

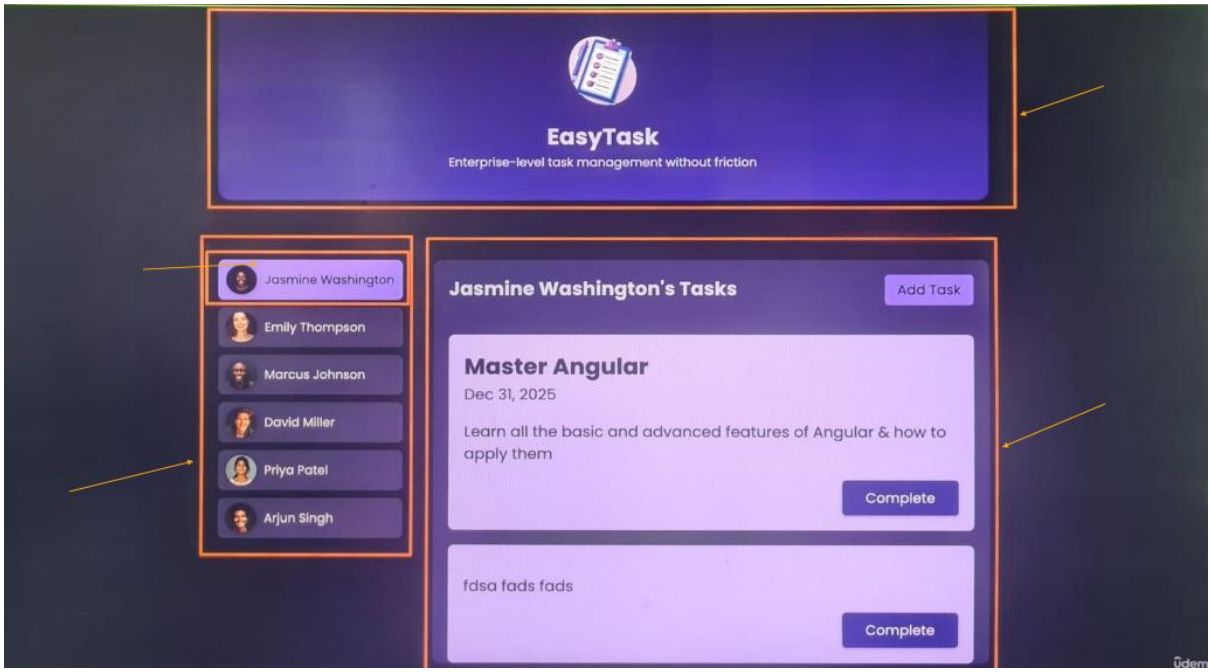
🧱 Component-Based Architecture in Angular

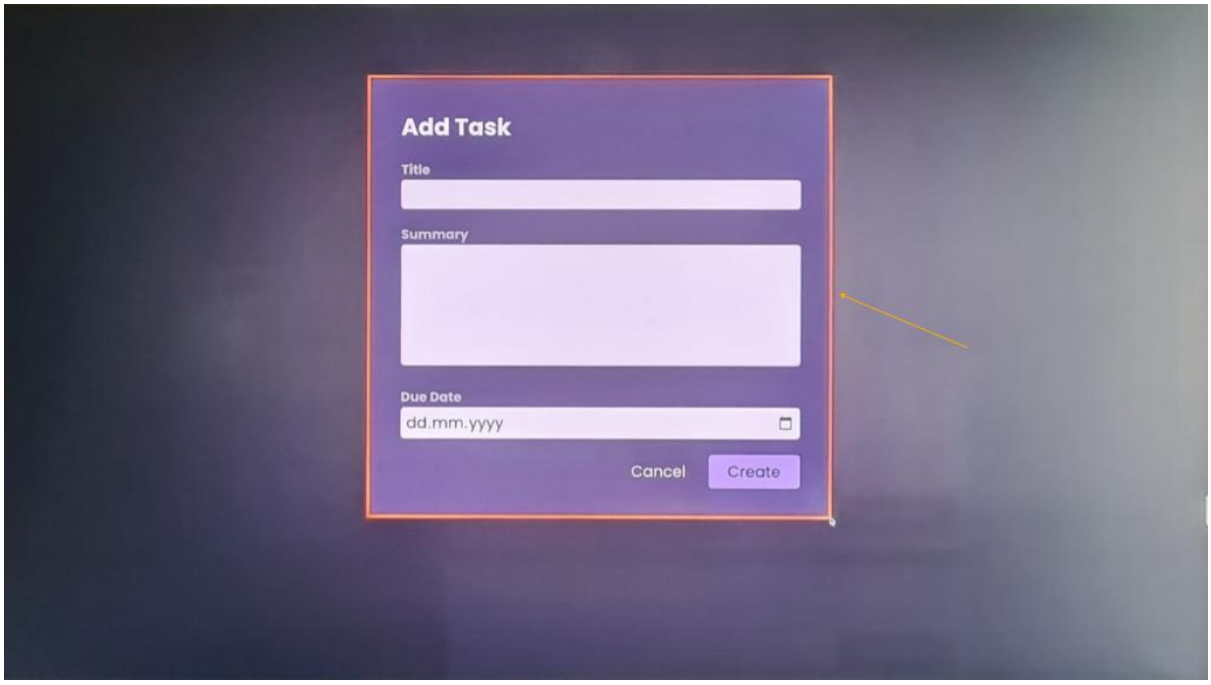
If you look at the final version of the demo app, you'll see that it's made up of **multiple UI sections**, such as:

- 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

File	Purpose
`header.component.css`	Contains scoped styles

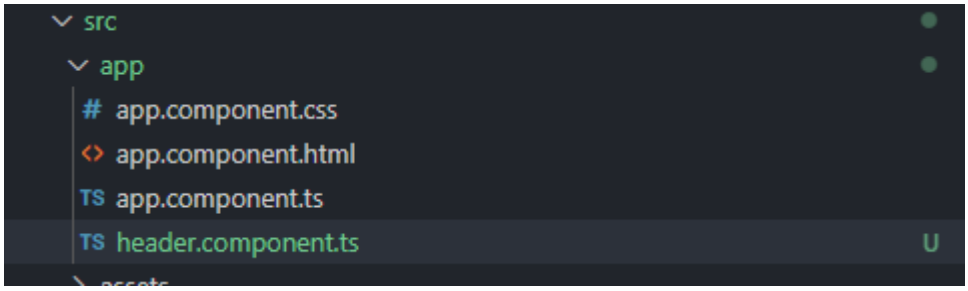
This is the **traditional naming convention** used since Angular 2.

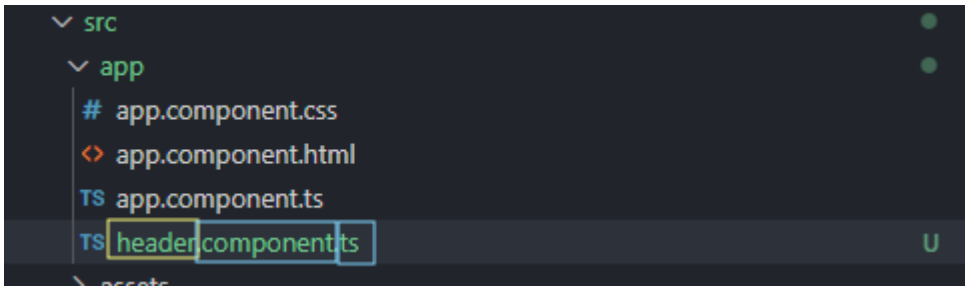
Angular 20 File Naming Note

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):

```
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }

  greet() {
    console.log('Hi, I am ' + this.name);
  }
}
```

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

```
const person1 = new Person('Max', 35);
const person2 = new Person('Anna', 32);
```

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

```
console.log(person1.age);
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:

```
@Component({})
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.

And, in general, you can learn more about TypeScript's support for classes here.

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

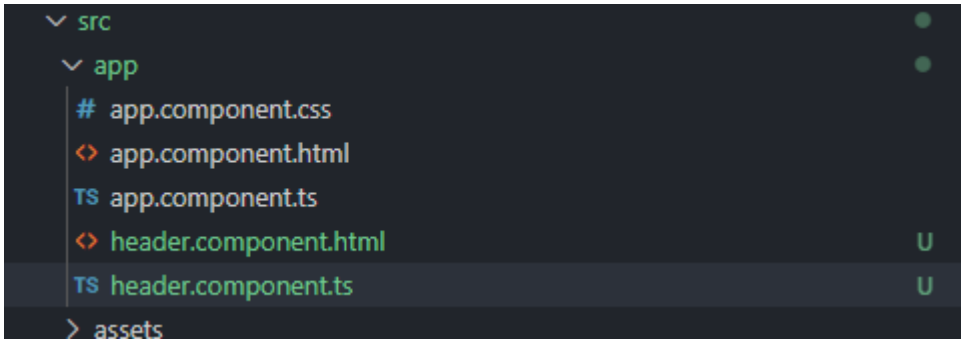
Lesson 12: Configuring the HeaderComponent

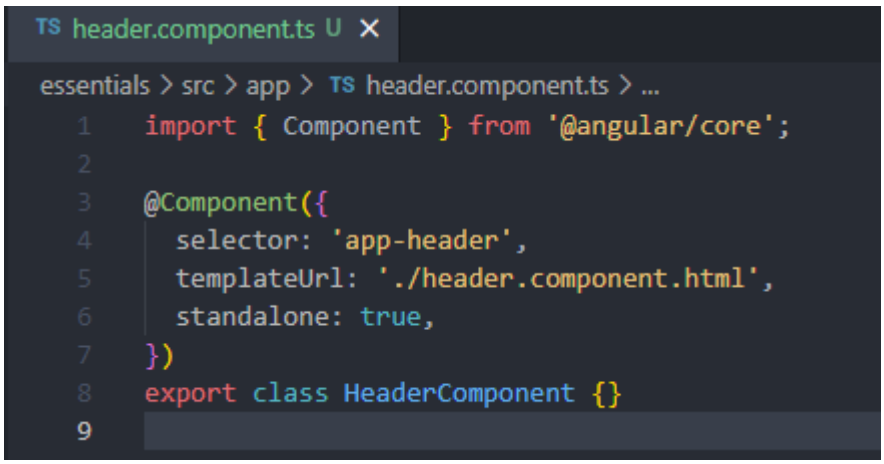
In the previous lesson, we created a `HeaderComponent` class and imported the `@Component` decorator. Now, let's complete its configuration and see how we can **use it in the app**.

🧠 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>`
```

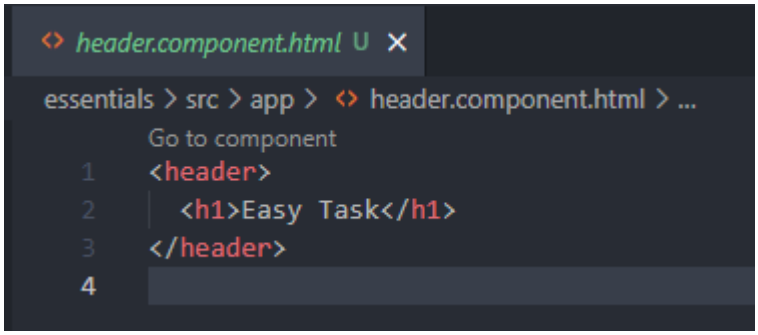
…but this is discouraged unless your template is extremely short (1–2 lines). For anything more complex, always use `templateUrl`.

🛠 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>
```

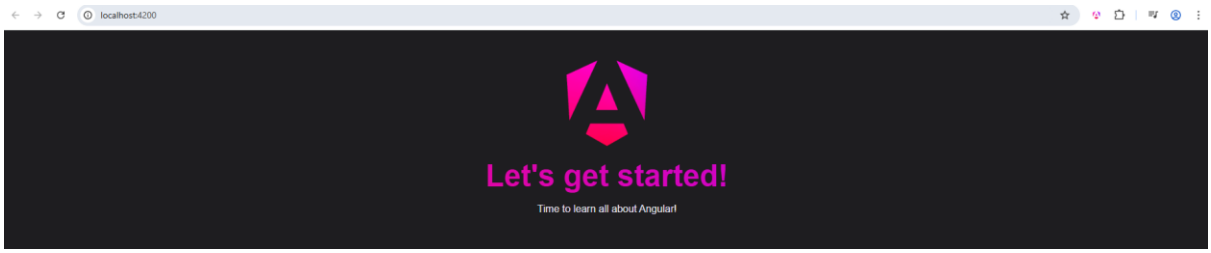
```
index.html M
essentials > src > index.html > ...
1  <!DOCTYPE html>
2  <html lang="en">
3    <head>
4      <meta charset="utf-8" />
5      <title>Essentials</title>
6      <base href="/" />
7      <meta name="viewport" content="width=device-width, initial-scale=1" />
8      <link rel="icon" type="image/x-icon" href="favicon.ico" />
9    </head>
10   <body>
11     <app-header></app-header>
12     <app-root></app-root>
13   </body>
14 </html>
15
```

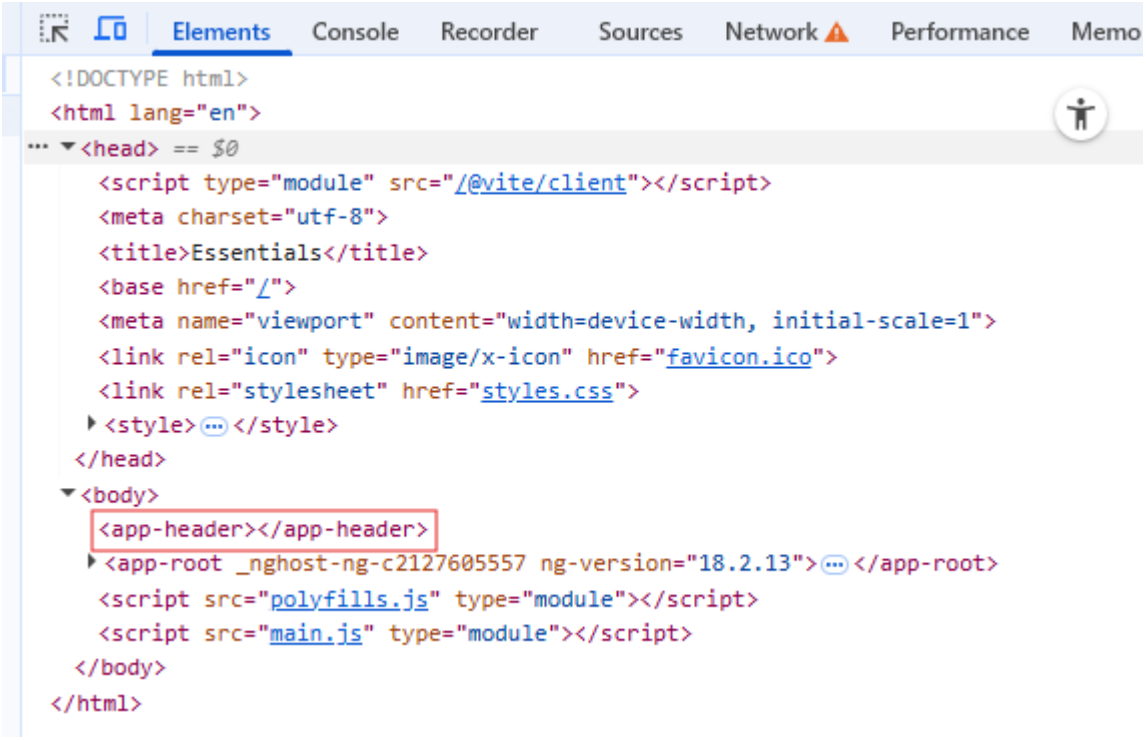
But if you do this:

- 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**

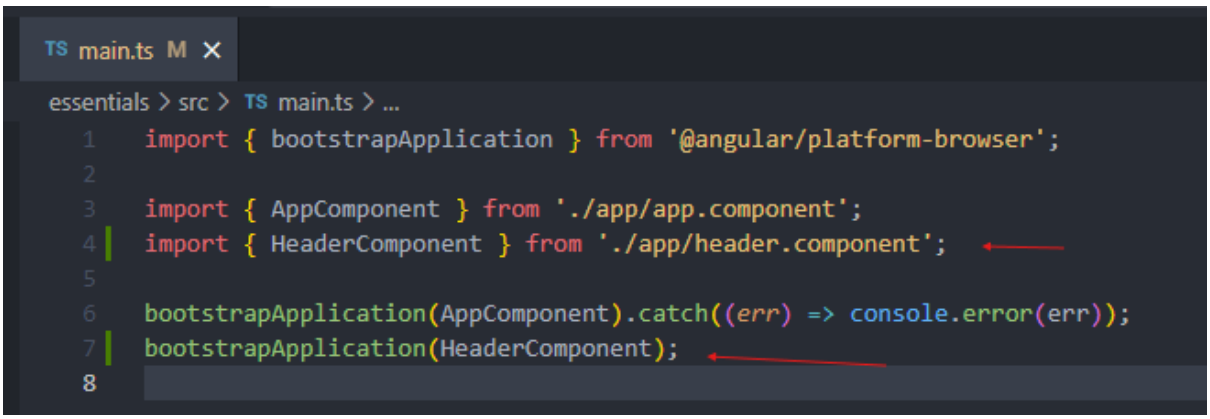


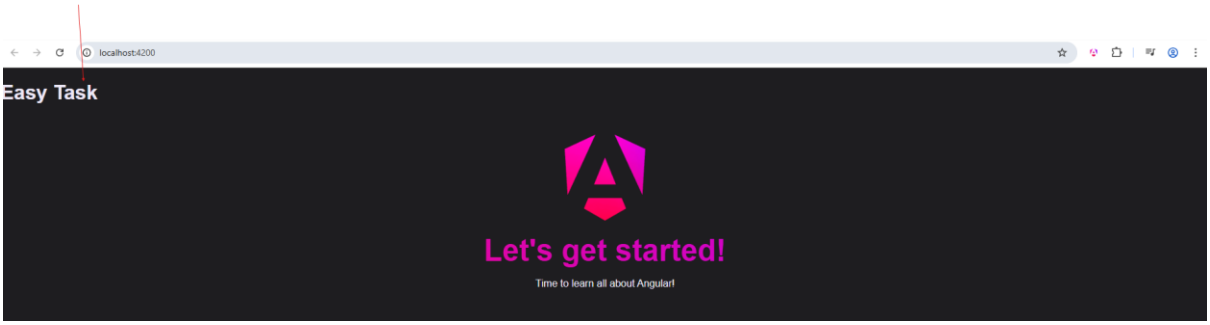


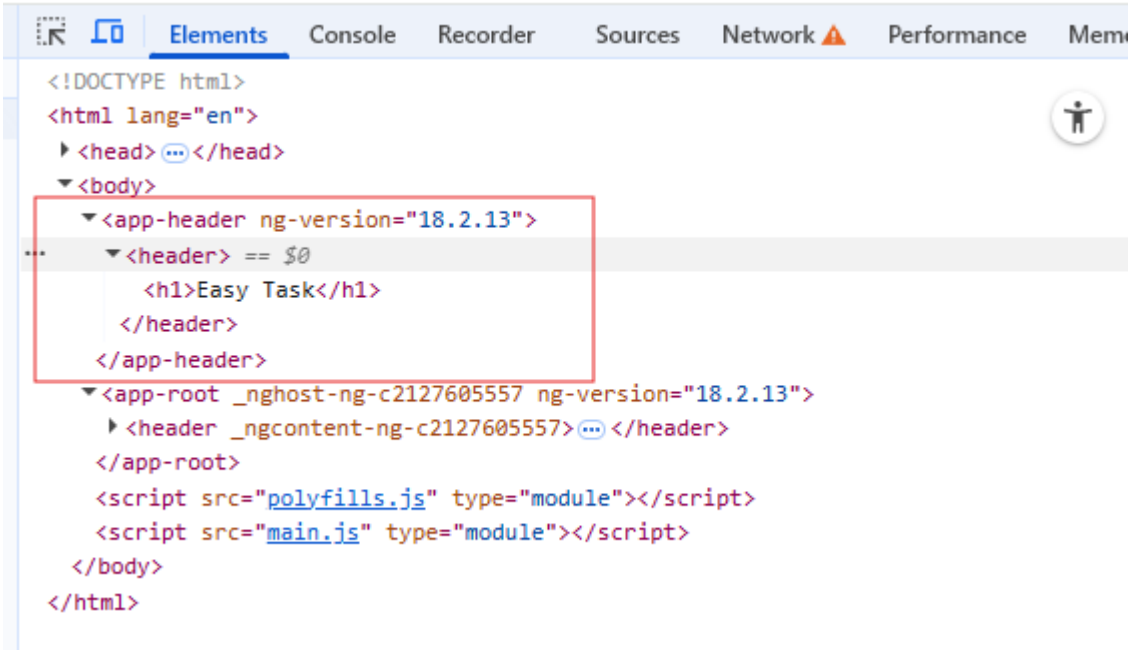
❗ Why Doesn’t Angular Render It?

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**







2. **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:

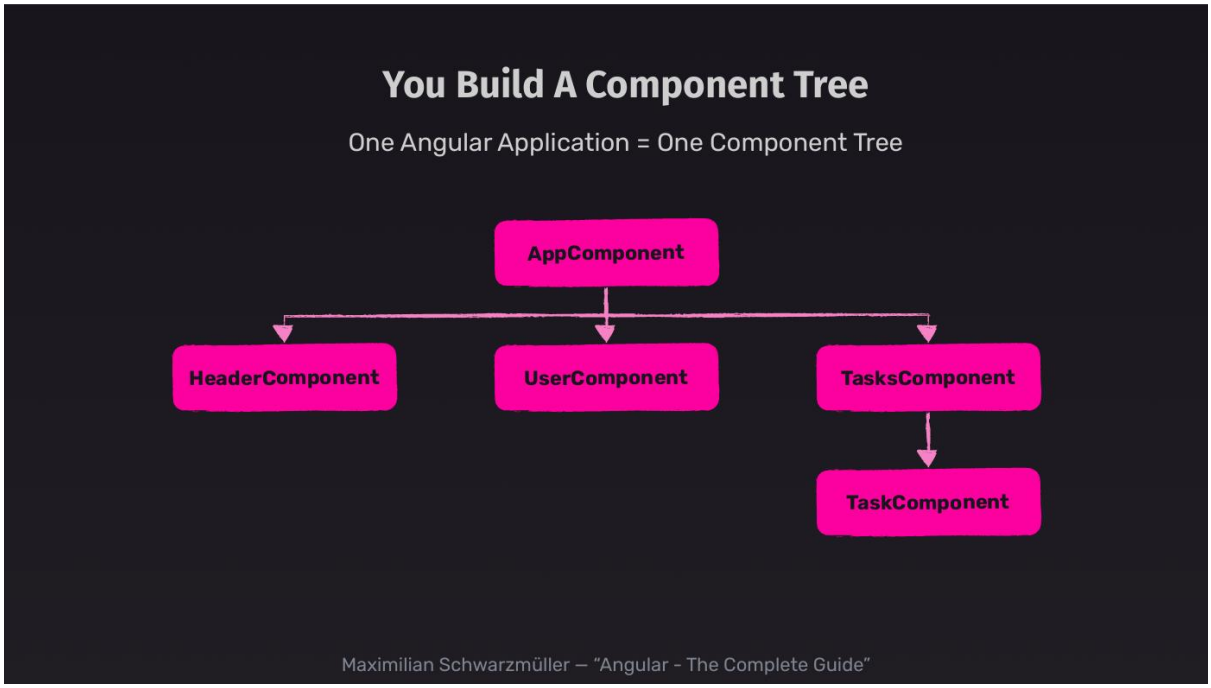
Bootstrap a **single root component** (usually `AppComponent`) and use all other components **within its template or sub-components**

This builds a **component tree**:

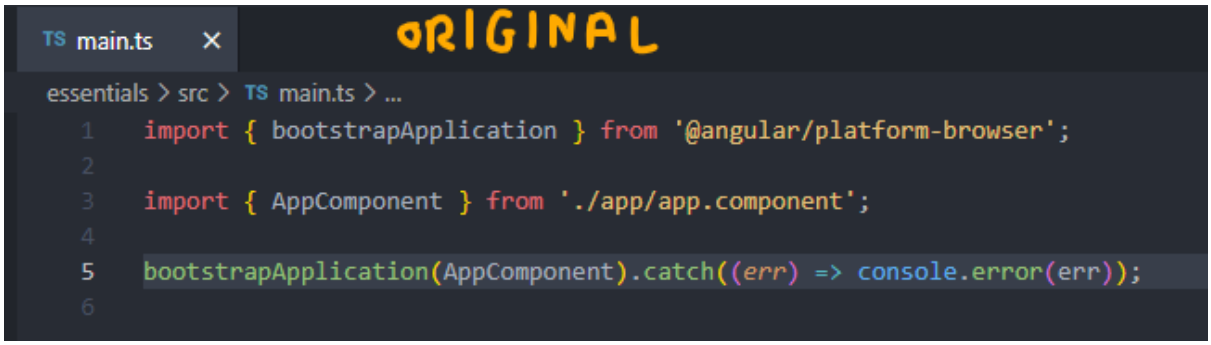
```
AppComponent
├── HeaderComponent
├── SidebarComponent
└── DialogComponent
```

- 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

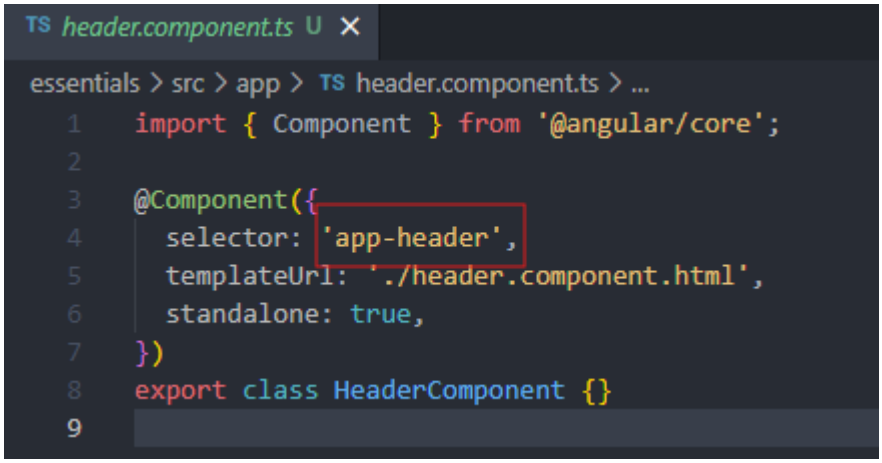
In `app.component.html`, replace the existing static `<header>` markup with:

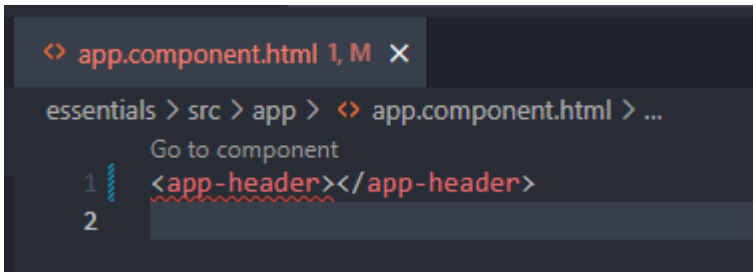
```
<app-header></app-header>
```

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:

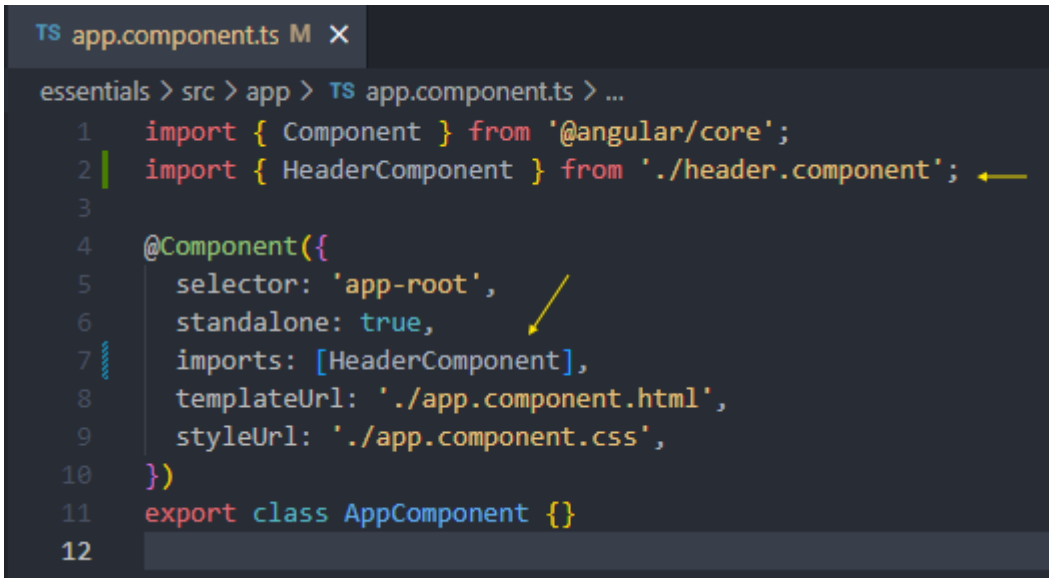
1. **Import** the `HeaderComponent`:

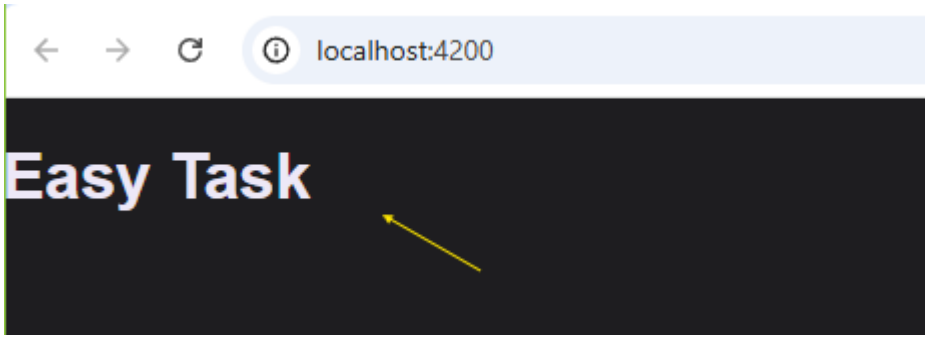
```
import { HeaderComponent } from './header.component';
```

2. **Add it to the `imports` array** of the `@Component` decorator:

```
@Component({
  selector: 'app-root',
  standalone: true,
  imports: [HeaderComponent],
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
```

✅ 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.
