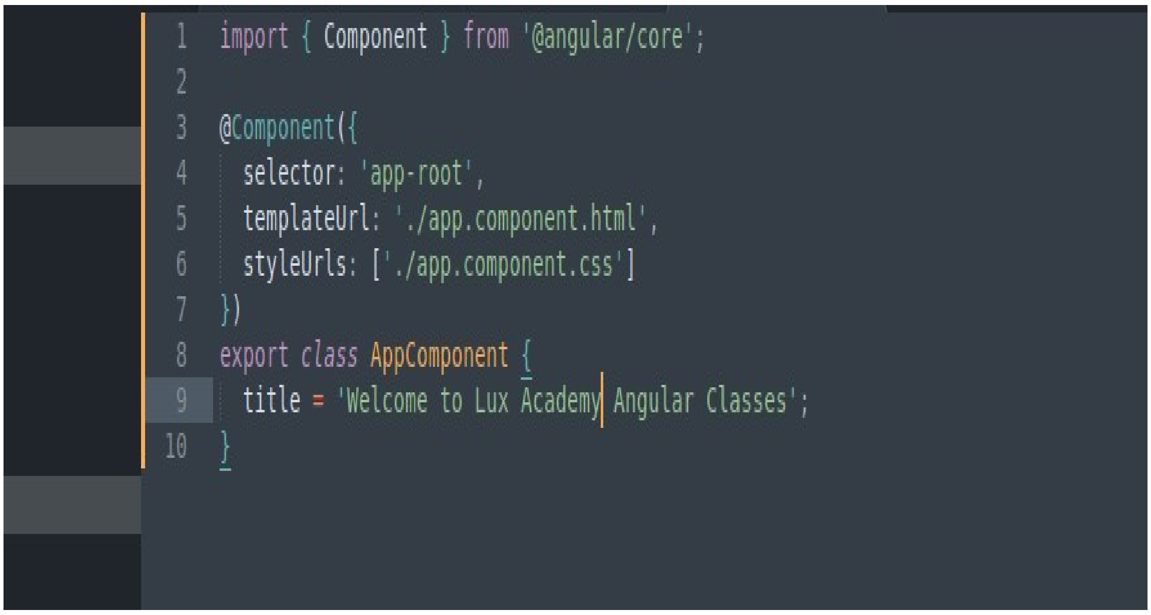
Components In Angular

The component is the main building block of any Angular applications. Components are composable, we can build larger components using multiple smaller components. Angular is basically a component-based MVVM framework. So, before going to discuss how to create or use a component in Angular 8, first, we need to understand what is a component and why it is required in the current web-based application developments?

What is a Component?

A Component is basically a class that is defined for any visible element or controls on the screen. Every component class has some properties and by using them, we can manipulate the behavior or looks of the element on the screen. So, we can create, update or destroy our own components as per the requirement at any stage of the application. But in TypeScript, a component is basically a TypeScript class decorated with an @Component() decorator. From an HTML point of view, a component is a user-defined custom HTML tag that can be rendered in the browser to display any type of UI element along with some business logic.



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

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

Decorators are mainly JavaScript functions that amend the decorated class in some way. A component is an independent, complete block of code that has the required logic, view, and data as one unit. As a developer for creating a component, we just need to define a class by providing some related configuration objects as an argument or parameter to the decorator function. Logically every component in Angular 8 acts as an MVC concept itself. Since each component class is an independent unit, it is highly reusable and can be maintained without messing with other components.

Why need Component-Based Architecture?

As per the current trend in web application development, the component-based architecture will be act as the most usable architecture in future web development. Because, with the help of this technique, we can reduce both development time and cost in a large volume in any large-scale web development projects. That's why technical experts currently recommend implementing this architecture in web-based application development. So, before going to discuss components in-depth, let's discuss why this component-based architecture is required for the web-based development:

Reusability – Component-based frameworks are much more useful due to its reusability features in the development. In this framework, components are the most granular units in the development process, and development with components allows gives us a provision of reusability in future development cycles. Since today, technology is changing rapidly. So, if we develop an application in a component-based format, then we are able to swap the best components in and out. A component-based architecture approach allows your application to stay up-to-date over time instead of rebuilding it from scratch.

- Increase Development Speed – Component-based development always supports agile methodology based development. Components can be stored in a library that the team can access, integrate, and modify throughout the development process.

In a broad sense, every developer has specialized skills. As an example, someone can be an expert in JavaScript, another in CSS, etc. With this framework, every specialized developer can contribute to developing a proper component.

- Easy Integration − So, in the component–based framework we can develop a library repository related to the component. This component repository can be used as a centralized code repository for the current development as well as the future new development also. As the other centralized code repository, we can maintain this library in any source control. In this way, the developer can access those repositories and can be updated with new features or functionality as per the new requirement and submit for approval through their own process.
- Optimize Requirement and Design − We can use component–library as a base UI component reference source and so using this source analysis team members like product managers, business analysis or technical leaders need to spend less time for finalizing the UI design for their new requirements. Because they already have a bunch of fully tested components with full functionality. Just they need to decide the process about the enhancement points including new business logic only. In this way, this component–based framework provides faster speed for the development process lifecycle.
- Lower Maintenance Costs − Since the component–based framework supports reusability, this framework reduces the requirement of the total number of developers when we want to create a new application. Logic–based components are normally context–free, and UI–based components always come with great UX and UI.

So, the developer can now focus on integrating those components in the application and how to establish connections between these types of components. Also, other system attributes like security, performance, maintainability, reliability, and scalability, (which are normally known as non-functional requirements or NFRs) can also be tested.

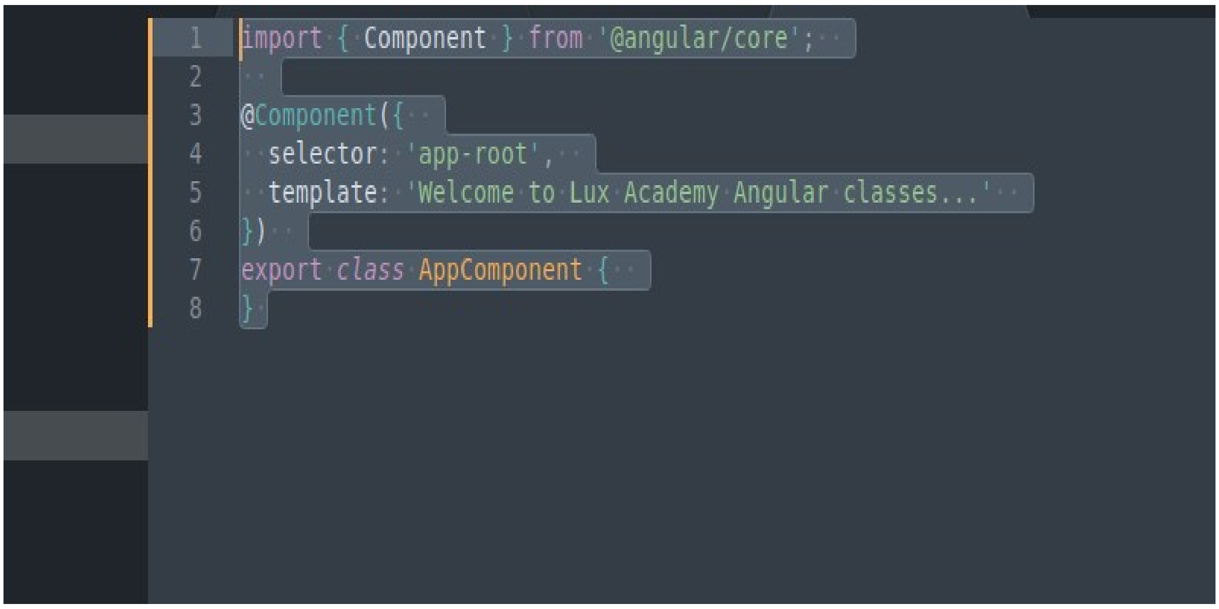
@Component Metadata

So in Angular, when we want to create any new component, we need to use the @Component decorator. @Component decorator basically classifies a TypeScript class as a component object. Actually, @Component decorator is a function that takes different types of parameters. In the @Component decorator, we can set the values of different properties to finalize or manipulate the behavior of the components. The most commonly used properties of the @Component decorator are as follows:

1. selector – A component can be used by the selector expression. Many people treat components like a custom HTML tag because finally when we want to use the component in the HTML file, we need to provide the selector just like an HTML tag.
2. Template – The template is the part of the component which is rendered in the browser. In this property, we can pass the HTML tags or code directly as inline code. Sometimes, we call this the inline template. To write multiple lines of HTML code, all code needs to be covered within the tilt (`) symbol.
3. TemplayeUrl – This is another way of rendering HTML tags in the browser. This property always accepts the HTML file name with its related file path. Sometimes it is known as the external template. The use of this property is much better when we want to design any complex UI within the component.

4. ModuleId – This is used to resolve the related path of template URL or style URL for the component objects. It contains the Id of the related modules in which the component is attached or tagged.
5. styles / stylesUrls – Components can be used in their own style by providing custom CSS, or they can refer to external style sheet files, which can be used by multiple components at a time. To provide an inline style, we need to use styles, and to provide an external file path or URL, we need to use styleUrls.
6. Providers – In the real–life application, we need to use or inject different types of custom services within the component to implement the business logic for the component. To use any user–defined service within the component, we need to provide the service instance within the provider. Basically, the provider property is always allowed array–type value. So that we can define multiple service instance names that can be provided by comma separation within this property at a time.

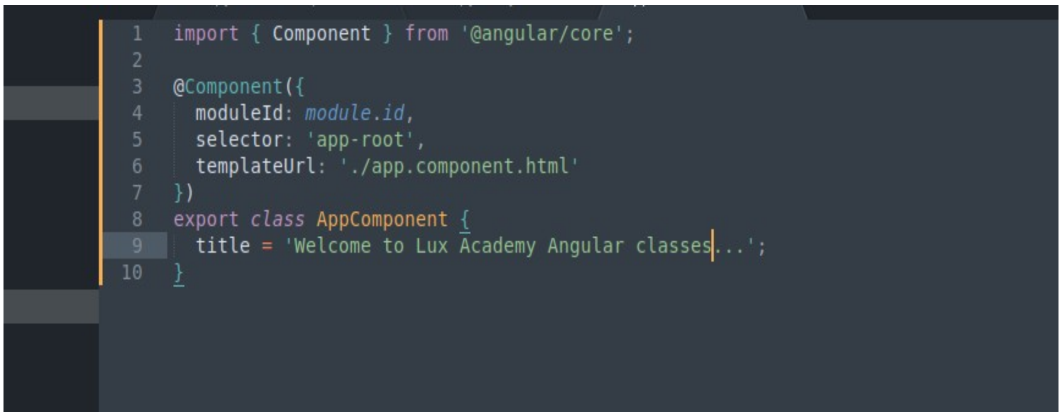
In the below example, we demonstrate how to define a component using some of the above properties like selector and template:



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

@Component({
  selector: 'app-root',
  template: 'Welcome to Lux Academy Angular classes...'
})
export class AppComponent {
}
```

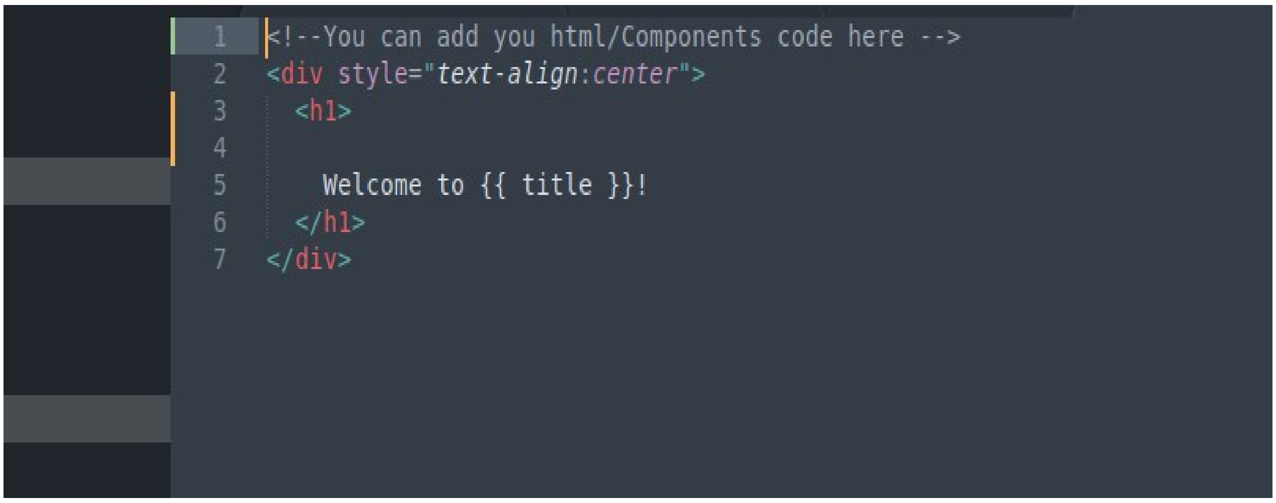
Now, in the below example, we will demonstrate how to use other @Component decorator properties like templateUrls:



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

@Component({
  moduleId: module.id,
  selector: 'app-root',
  templateUrl: './app.component.html'
})
export class AppComponent {
  title = 'Welcome to Lux Academy Angular classes...';
}
```

So, in the above example, we will separate the HTML file for storing the HTML part related to the components. As per the above example, we need to place both the TypeScript file and HTML file in the same location. If we want to place the HTML in a separate folder, then we can use that file in the component decorator using a relative file path. Below is the sample code is written in the app.component.html file:



```
<!--You can add you html/Components code here -->
<div style="text-align:center">
  <h1>

    Welcome to {{ title }}!
  </h1>
</div>
```

All the best, Lux Academy.