Angular 8 : Online Class

Class – 4

By: Sahosoft Solutions

Presented by : Chandan Kumar

Components

Components are the most basic user interface building block of an Angular application. An angular application contains a tree of angular components.
It is a subset of directives and always associated with a template. Unlike other directives, you can only create one instance of a component per element in a template.

A component is nothing more than a simple TypeScript class in which you can create your own methods and properties according to your needs, which is used to bind with a UI (HTML page or cshtml page) of our application.
The most important feature of any Angular application is the component that controls View or the template that we use. In general, we write all the application logic for the view that is assigned to this component.

Therefore, an Angular application is just a tree of such Components, when each Component is processed, it recursively processes its children Components. At the root of this tree is the top level component, the main component.
When we bootstrap an Angular application, we tell the browser to render that top level root Component which renders its child component and so on.

Components

APP.COMPONENT.TS

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

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

Let's take a look at each of these 3 definition sections in detail.

1. Component Imports

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

Components

The way you make a class component is by importing the component member from the @angular / core library: some components may have dozens of imports based on the component's needs.

2. The Component Decorator

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

The next section is what is known as a component decorator. The @Component is using the component that was imported from the above import line. This is what makes this class a component.
Within the component, it has a variety of configuration properties that help define this component.

• **selector:** this is the name of the label to which the component is applied. For example: <app-root> Loading ... </ app-root> in index.html.

• **templateUrl & styleUrls:** these define the HTML template and style sheets associated with this component. You can also use the template and style properties to define inline HTML and CSS.

There are other properties that can be defined within the component decorator based on the component's needs. We will see in the next chapter.

Components

The Component Class

```
export class AppComponent {
  title = 'app';
}
```

Lastly, we have the core of the component, that is the component class.
This is where the various properties and methods are defined. All properties and methods defined here are accessible from the template. Likewise, events occurring in the template are accessible within the component.
Become a component when we add @Component metadata or we can say that when you decorate the simple class with the @Component attribute, it becomes the component.
To make this class an angular component, we need to decorate the above class using the @Component decorator that is present in the @ angular / core library.

Note: The normal TypeScript class will become a component class once decorated with @component decorator.

Components

Finally, a component must belong to an NgModule to be available for another component or application. To specify that a component is a member of an NgModule, it must be included in the declaration field of that @NgModulemetadata (NgModule).

When we created a new project using the angular-cli command, the above files were created by default. The file structure has the application component and consists of the following files-

- app.component.css
- app.component.html
- app.component.spec.ts
- app.component.ts
- app.module.ts

Components

If you open the app.module.ts file, you have some imported libraries and also a declarative to which the Appcomponent is assigned in the following way:

APP.MODULE.TS

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';

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

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

As per above code the declarations include the AppComponent variable, which we have already imported. This becomes the parent component.

Components

Now, angular-cli has a command to create your component. However, the app component that is created by default will always remain the parent component and the next components created will form the child components. Let's create your corner component.
create a components using the following command in the integrated terminal:

ng g c demo
• **ng** calls the angular CLI
• **g** is the abbreviation of generate (note, you can use the whole word generate if you wish)
• **c** is the abbreviation of component (note, you can use the whole word component if you wish).
component is the type of element that is going to be generated (others include directive, pipeline, service, class, security, interface, enumeration and module)
• and then the **name of the component**
When you run the above command on the command line, you will receive the following output

Components

Now, if we check the structure of the file, we will get the new demo folder created in the src/app folder. The following files are created in the demo folder:

- demo.component.css: the css file is created for the new component.
- demo.component.html: the html file was created.
- demo.component.spec.ts: can be used for unit tests.
- demo.component.ts: here we can define the module, properties, etc.

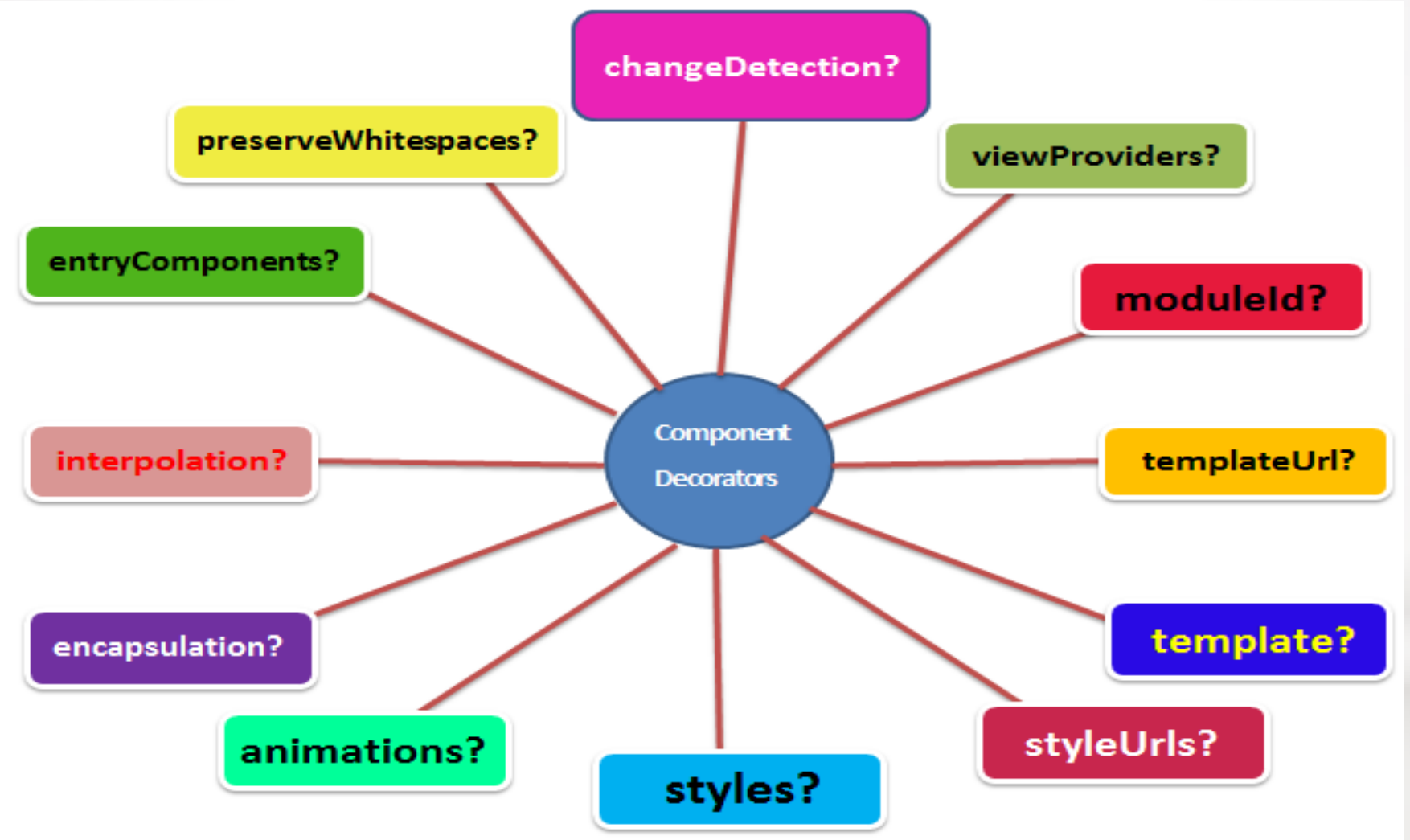
The changes are added to the app.module.ts file as follows:

Brief of @Component Decorators

@component decorator provides additional metadata that determines how to process, instantiate and use the component at runtime (the decorators in Typescript are like annotations in Java or attributes in C #) component Decorator accepts the required configuration object that requires information to create and display the component in real time.

```
@Component({
  changeDetection?: ChangeDetectionStrategy
  viewProviders?: Provider[]
  moduleId?: string
  templateUrl?: string
  template?: string
  styleUrls?: string[]
  styles?: string[]
  animations?: any[]
  encapsulation?: ViewEncapsulation
  interpolation?: [string, string]
  entryComponents?: Array<Type<any> | any[]>
  preserveWhitespaces?: boolean
  // inherited from core/Directive
  selector?: string
  inputs?: string[]
  outputs?: string[]
  providers?: Provider[]
  exportAs?: string
  queries?: {...}
  host?: {...}
  jit?: boolean
})
```

Brief of @Component Decorators



Components

- **changeDetection -** change the detection strategy used by this component
- **viewProviders -** list of providers available for this component and the view of their children.

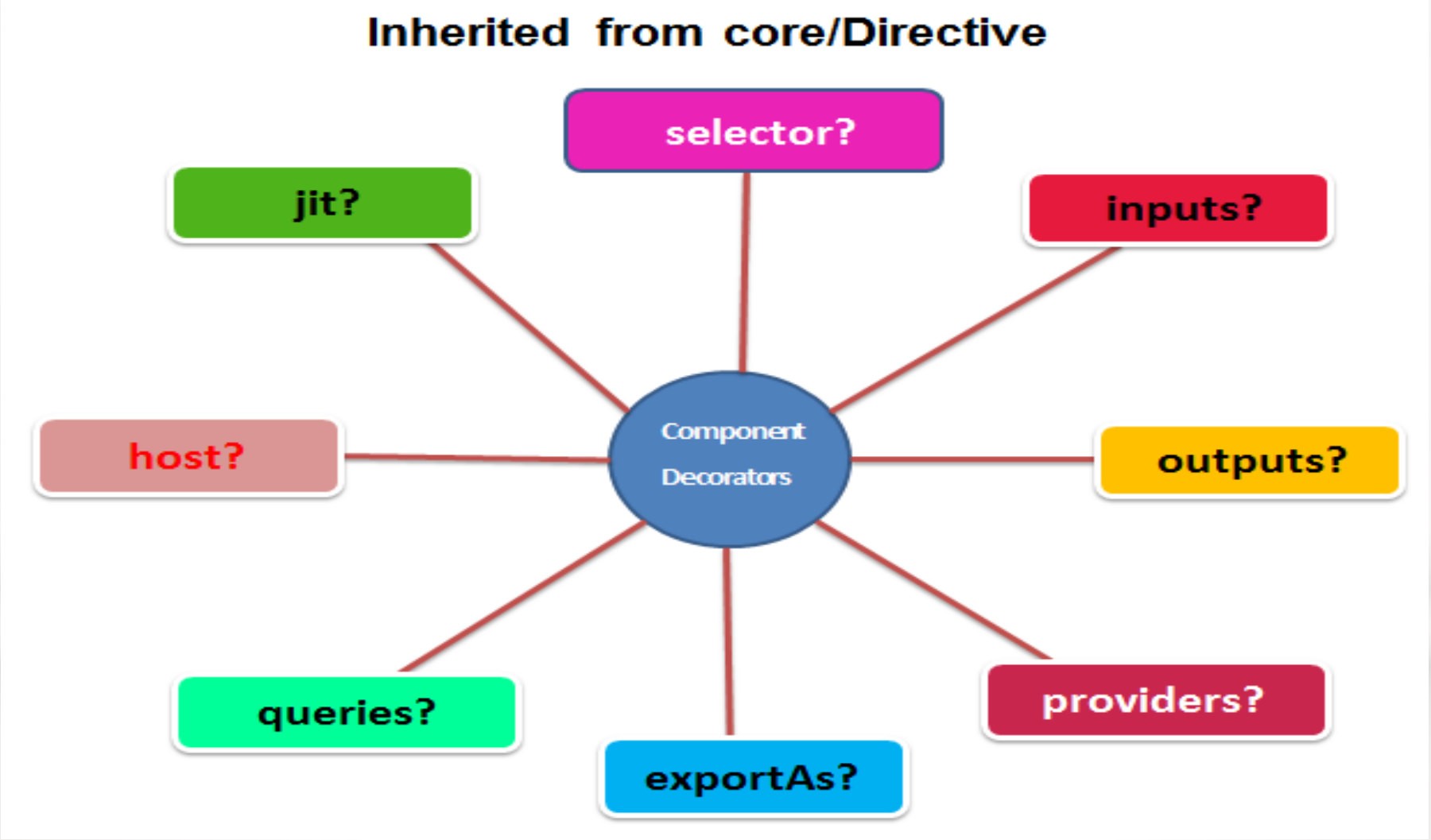
- **moduleId -** Module ID ES / CommonJS of the file in which this component is defined.
- **templateUrl -** url in an external file that contains a template for the view.

- **template -** template defined inline template for the view.
- **styleUrls -** url list for style sheets that will be applied to the view of this component.
- **styles -** styles defined online that will be applied to the view of this component.
- **animations -** animation's list of this component.

- **encapsulation -** strategy of style encapsulation used by this component.
- **interpolation -** custom interpolation markers used in the template of this component.

- **entryComponents -** entryComponents is the list of components that are dynamically inserted into the view of this component.
- **preserveWhitespaces -** Using this property, we can remove all whitespaces from the template.

Components


Inherited from core/Directive
selector?
jit?
inputs?
host?
Component
Decorators
outputs?
queries?
exportAs?
providers?

Components

- **selector -** css selector which identifies this component in a template.
- **inputs -** it is property within one component (child component) to receive a value from another component (parent component).

- **outputs -** it is property of a component to send data from one component (child component) to calling component (parent component).
- **providers -** Providers are usually singleton objects (an instance), to which other objects have access through dependency injection (DI).

- **exportAs -** name under which the component instance is exported to a template.
- **queries -** allows you to configure queries that can be inserted into the component.

- **host -** Map of class properties to host element links for events, properties, and attributes.
- **jit -** if true, the AOT compiler will ignore this directive/component and will therefore always be compiled using JIT.
- .

template & templateUrl

We know that the @Component decorator functions take an object and this object contains many properties. So we will learn about the properties of the template & templateUrl in this article.
We can render our HTML code within the @Component decorator in two ways.

Inline Templates
The Inline templates are specified directly in the component decorator. In this, we will find the HTML inside the TypeScript file. This can be implemented using the "template" property.
The literal values of the template with back-ticks marks allow strings on multiple lines. It means that if you have HTML on more than one line, you should use backticks instead of single or double quotes, as shown below.

External Template
The External templates define HTML in a separate file and refer to this file in templateUrl. means. In this, we will find a separate HTML file instead of finding the HTML inside the TS file. Here, the TypeScript file contains the path to that HTML file with the help of the "templateUrl" property.

.

template & templateUrl

We know that the @Component decorator functions take an object and this object contains many properties. So we will learn about the properties of the template & templateUrl in this article.
We can render our HTML code within the @Component decorator in two ways.

Inline Templates
The Inline templates are specified directly in the component decorator. In this, we will find the HTML inside the TypeScript file. This can be implemented using the "template" property.
The literal values of the template with back-ticks marks allow strings on multiple lines. It means that if you have HTML on more than one line, you should use backticks instead of single or double quotes, as shown below.

External Template
The External templates define HTML in a separate file and refer to this file in templateUrl. means. In this, we will find a separate HTML file instead of finding the HTML inside the TS file. Here, the TypeScript file contains the path to that HTML file with the help of the "templateUrl" property.

.

Styles and StyleUrls

We know that the decorator functions of @Component take object and this object contains many properties. We can represent our styles, ie the CSS code within the @Component decorator in two ways.

Inline Styles

The Inline Style are specified directly in the component decorator. In this, you will find the CSS within the TypeScript file. This can be implemented using the "styles" property.

External Styles

The External styles define CSS in a separate file and refer to this file in styleUrl.means In this, we will find a separate CSS file instead of finding a CSS within the TypeScript file. Here, the TypeScript file contains the path to that style sheet file with the help of the "stylesUrls" property..

preserveWhitespaces

We know that the decorator functions of @Component take object and this object contains many properties. Using this property, we can remove all whitespaces from the template. it takes a Boolean value, that is:

If it is **false,** it will remove all whitespace from the compiled template.
If it is **true,** it will not remove whitespace from the compiled template.

preserveWhitespaces With true
Use the following code in app.component.ts file.

preserveWhitespaces

We know that the decorator functions of @Component take object and this object contains many properties. Using this property, we can remove all whitespaces from the template. it takes a Boolean value, that is:

If it is **false,** it will remove all whitespace from the compiled template.
If it is **true,** it will not remove whitespace from the compiled template.

preserveWhitespaces With true
Use the following code in app.component.ts file.

viewProvider

We know that the decorator functions of @Component take object and this object contains many properties.

The viewProviders property allows us to make providers available only for the component’s view.
When we want to use a class in our component that is defined outside the @Component () decorator function, then, first of all, we need to inject this class into our component, and we can achieve this with the help of the "viewProvider" property of a component.

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

class MyProvider {
  constructor() {
    console.log("provider property");
  }
  VarMyProvider = "VarMyProvider";
}

class MyProvider1 {
  VarMyProvider1 = "VarMyProvider1";
  constructor() {
  }
  getString(name) {
    console.log("provider property1" + name);
  }
}

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
  viewProviders: [MyProvider, MyProvider1]
})
export class AppComponent {
  constructor(public obj: MyProvider, public obj1: MyProvider1) {
    obj1.getString(" SahosoftTutorials.com");
    console.log(obj.VarMyProvider);
    console.log(obj1.VarMyProvider1);
  }

  title = 'app';
}
```