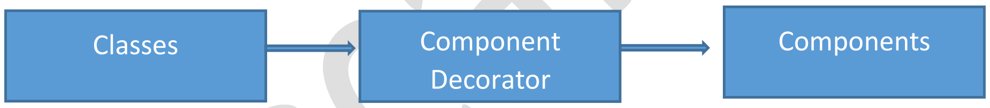
Agenda

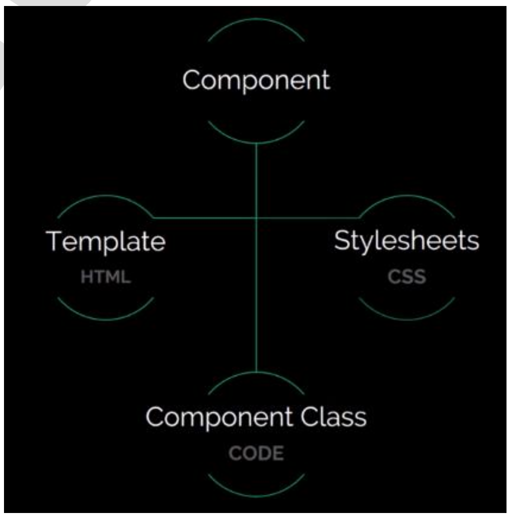
- What are Components?
- Understanding Components life cycle hooks
- Creating and using components
- Dynamic components using **ngComponentOutlet**

What are Components?

- **Components** are the most basic building block of an UI in an Angular application.
- An **Angular application is a tree of Angular components**. Angular **components are a subset of directives.**
- Unlike directives, components **always have a template** and **only one component can be instantiated** per an element in a template.
- A component must belong to an **NgModule** in order for it to be usable by another component or application. To specify that a component is a member of an **NgModule**, you should list it in the **declarations** field of that **NgModule**.
- In addition to the metadata configuration specified via the Component decorator, components can control their runtime behaviour by implementing various **Life-Cycle hooks**.
- **Angular 4 components** are simply **classes** that are designated as a component with the help of a **component decorator**.



- Every component has a defined **template** which can communicate with the **code** defined in the **component class**.



Components are separated into three different sections

1. **Imports:** Components are imported from the predefined libraries of Angular, for building components the basic library we use is **'@angular/core'.** Some components may import more than one library based on the needs and we can **import services** that may our component make use of it using **Dependency Injection (DI)**.

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

2. **Component Decorator: @Component** used is from the above import. A component can control its runtime behaviour by implementing various **Configuration Properties.** Mostly we use **selector, template, templateUrl, styles, stylesUrls.** As per the needs we can use various properties. Let's have a look at the other properties of component decorator.

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

Properties:

- **animations** - list of animations of this component
- **changeDetection** - change detection strategy used by this component
- **encapsulation** - style encapsulation strategy used by this component
- **entryComponents** - list of components that are dynamically inserted into the view of this component
- **exportAs** - name under which the component instance is exported in a template
- **host** - map of class property to host element bindings for events, properties and attributes
- **inputs** - list of class property names to data-bind as component inputs
- **interpolation** - custom interpolation markers used in this component's template
- **moduleId** - ES/CommonJS module id of the file in which this component is defined
- **outputs** - list of class property names that expose output events that others can subscribe to
- **providers** - list of providers available to this component and its children
- **queries** - configure queries that can be injected into the component
- **selector** - css selector that identifies this component in a template
- **styleUrls** - list of urls to stylesheets to be applied to this component's view
- **styles** - inline-defined styles to be applied to this component's view
- **template** - inline-defined template for the view
- **templateUrl** - url to an external file containing a template for the view
- **viewProviders** - list of providers available to this component and its view children

3. **Component Class:** It is the core of the component. This is the place where we **define properties and methods** which are accessible from the **template**. Similarly **events** that are used with in the template are **accessible with in component.**

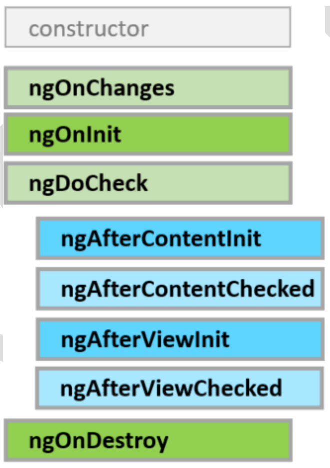
```
export class AppModule {
   //your properties, methods and events.
}
```

Understanding Components Life Cycle Hooks

A component has as life cycle managed by angular.

Angular creates it, renders it, creates and renders its children, checks it when it's data-bound properties change, and destroys it before removing it from the DOM.

- Angular offers **lifecycle hooks** that provide visibility into these key life moments and the ability to act when they occur.
- A directive has the same set of lifecycle hooks, minus the hooks that are specific to component content and views.
- Whereas the developers can tap into key moments of that life cycle by implementing one or more life cycle hook interfaces in the **@angular/core** library.



After creating a component angular calls the life cycle hook methods in the following sequence.

- **ngOnChanges** - called when an input binding value changes
- **ngOnInit** - after the first **ngOnChanges**
- **ngDoCheck** - after every run of change detection
- **ngAfterContentInit** - after component content initialized
- **ngAfterContentChecked** - after every check of component content
- **ngAfterViewInit** - after component's view(s) are initialized
- **ngAfterViewChecked** - after every check of a component's view(s)
- **ngOnDestroy** - just before the component is destroyed

Ex: Component Life-cycle

File: **app.component.ts**

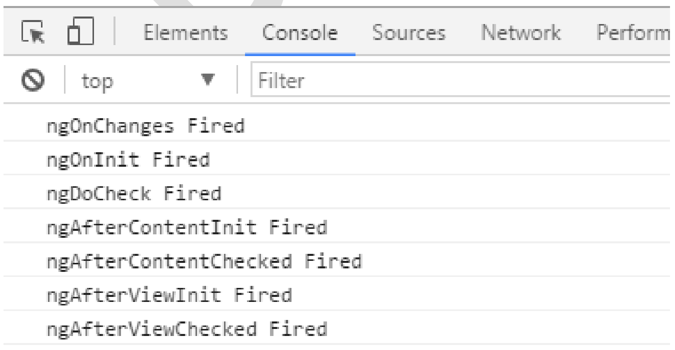
```
import {
Component,Input,OnInit,OnChanges,OnDestroy,DoCheck,AfterContentInit,AfterContentChecked,AfterViewInit
,AfterViewChecked } from '@angular/core';
@Component({
 selector: 'my-first-component',
 template:`<Student [name]="myname"></Student><br/> <input type="text" [(ngModel)]="myname">`
})
export class AppComponent {
   myname = "Sandeep Soni";
}
//Creating a component
@Component({
   selector: 'Student',
   template: `<h3>Hello {{name}}</h3>`,
})
export class Student implements OnInit, OnDestroy, OnChanges, DoCheck, AfterContentChecked,
AfterContentInit, AfterViewInit, AfterViewChecked {
   @Input() name: any;
   getAlert (studName:string) {
      alert("Hello " + studName);
   }
   ngOnChanges() {
      console.log("ngOnChanges Fired");
   }
   ngOnInit() {
      console.log("ngOnInit Fired");
   }
   ngOnDestroy() {
      console.log("ngOnDestroy Fired");
      alert("destroy");
   }
   ngDoCheck() {
      console.log("ngDoCheck Fired");
   }
   ngAfterContentInit() {
    console.log("ngAfterContentInit Fired");
  }
  ngAfterContentChecked() {
    console.log("ngAfterContentChecked Fired");
  }
  ngAfterViewInit() {
    console.log("ngAfterViewInit Fired");
  }
  ngAfterViewChecked() {
    console.log("ngAfterViewChecked Fired");
  }
}
```

File: **app.module.ts**

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule } from '@angular/forms';
import { AppComponent, Student, LifeCycle } from './app1.component';  //importing app.component module
@NgModule({
  imports: [BrowserModule, FormsModule],
  declarations: [AppComponent, Student, LifeCycle],
  bootstrap: [AppComponent]
})
export class AppModule {  }
```

This is how it shows the output. It's not mandatory that imported interfaces to be in the same order. Here **ngOnChanges** will get fired when there is any change in the textbox.

Output:



```
ngOnChanges Fired
ngOnInit Fired
ngDoCheck Fired
ngAfterContentInit Fired
ngAfterContentChecked Fired
ngAfterViewInit Fired
ngAfterViewChecked Fired
```

Dynamic Components using **ngComponentOutlet**

- **NgComponentOutlet** is a new directive that provides a declarative approach for creating dynamic components.
- **Ng-container** directive is a logical container that can be used to group nodes but is not rendered in the DOM tree. **<ng-container>** is rendered as HTML comment. **Ng-container** is mostly used when we want to group elements on a condition based **(i.e., using *ng-if).**

Syntax:

Simple

```
<ng-container * ngComponentOutlet="componentTypeExpression" > </ng-container>
```

Customized Injector/Content

```
<ng-container * ngComponentOutlet="componentTypeExpression;
                    injector: injectorExpression;
                    content: contentNodesExpression; ">
  < /ng-container>
```

Customized ngModuleFactory

```
<ng-container * ngComponentOutlet="componentTypeExpression;
          ngModuleFactory: moduleFactory; ">
  < /ng-container>
```

Ex: simple example using **ngComponentOutlet**

File: **app.component.ts**

```
@Component({ selector: 'hello-world', template: 'Hello World!' })
export class HelloWorld {
}
@Component({
  selector: 'ng-component-outlet-simple-example',
  template: `<ng-container *ngComponentOutlet="HelloWorld"></ng-container>`
})
export class NgTemplateOutletSimpleExample {
  // This field is necessary to expose HelloWorld to the template.
  HelloWorld = HelloWorld;
}
```

File: **app.module.ts**

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

import { HelloWorld, NgTemplateOutletSimpleExample } from './app1.component';  //importing
app.component module

@NgModule({
  imports: [BrowserModule, FormsModule],
  declarations: [HelloWorld, NgTemplateOutletSimpleExample],
  bootstrap: [NgTemplateOutletSimpleExample],
  entryComponents: [HelloWorld]
})
export class AppModule {  }
```

File: **Index.html**

```
<body>
  <ng-component-outlet-simple-example>Loading...</ng-component-outlet-simple-example>
</body>
```

- Here we are using **ngContainer** with **ngComponentOutlet** that takes as an Input reference to the component. And you need to add your dynamic components to **entryComponents** section of the module.
- Components which are added in **@NgModule.bootstrap** are added automatically in **entryComponents** automatically, and the components which are reference in router configuration are also added automatically in declarations section automatically by angular. But when your app is **bootstrapping or dynamically loading the components** you must add to it **entryComponents** explicitly.

You can control the component creation process by using the following optional attributes:

- **ngComponentOutletInjector:** Optional custom Injector that will be used as parent for the Component. Defaults to the injector of the current view container.
- **ngComponentOutletProviders:** Optional injectable objects (Provider) that are visible to the component.
- **ngComponentOutletContent:** Optional list of projectable nodes to insert into the content section of the component, if exists.
- ngComponentOutletNgModuleFactory: Optional module factory to allow dynamically loading other module, then load a component from that module.

Ex: **Using Customized Injector/ Content**

File: **app.component.ts**

```
@Injectable()
class Employee {
    EmpName = "SandeepSoni";
}
@Component({
    selector: 'complete-component',
    template: `EmpDetails: <ng-content></ng-content> {{ emp.EmpName }}`
})
export class EmployeeComponent {
    constructor(public emp: Employee) { }
}
@Component({
    selector: 'ng-component-outlet-complete-example',
    template: `
    <ng-container *ngComponentOutlet="EmpComponent;
                                injector: myInjector;
                                content: myContent"></ng-container>`
})
export class NgTemplateOutletCompleteExample {
    // This field is necessary to expose CompleteComponent to the template.
    EmpComponent = EmployeeComponent;
    myInjector: Injector;
    myContent = [[document.createTextNode('Hello')]];
    constructor(injector: Injector) {
        this.myInjector = ReflectiveInjector.resolveAndCreate([Employee], injector);
    }
}
```

- Here we use **<ng-content>** directive for transclusion, **transclusion** will **transfer and includes** the content HTML mark up in the destination mark up.

Ex: Using **NgModuleFactory**

- **Compiler** is a service which is used for running the angular **compiler during runtime** and each **@NgModule** will provide an own compiler to its injector.
- **compileModuleSync(moduleType)**

```
compileModuleSync(moduleType: Type<T>) : NgModuleFactory<T>
```

Compiles the given **NgModule** and all of its components. All templates of the components listed in **entryComponents** have to be inlined.

File: **app.component.ts**

```
@Component({ selector: 'other-module-component', template: `Other Module Component!` })
class OtherModuleComponent {
}
@Component({
  selector: 'ng-component-outlet-other-module-example',
  template: `
  <ng-container *ngComponentOutlet="OtherModuleComponent;
                            ngModuleFactory: myModule;"></ng-container>`
})
export class NgTemplateOutletOtherModuleExample {
  // This field is necessary to expose OtherModuleComponent to the template.
  OtherModuleComponent = OtherModuleComponent;
  myModule: NgModuleFactory<any>;

  constructor(compiler: Compiler) { this.myModule = compiler.compileModuleSync(OtherModule); }
}
// This is the Other Module
@NgModule({
  imports: [CommonModule],
  declarations: [OtherModuleComponent],
  entryComponents: [OtherModuleComponent]
})
export class OtherModule {
}
```

File: **app.module.ts**

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

import { NgTemplateOutletOtherModuleExample } from './app1.component';  //importing app.component
module
@NgModule({
  imports: [BrowserModule, FormsModule],
  declarations: [NgTemplateOutletOtherModuleExample],
  bootstrap: [NgTemplateOutletOtherModuleExample],
  entryComponents: []
})
```

```
export class AppModule {  }
```