Future of Web Development → Component based Development → ANGULAR

Introduction

– A JavaScript framework to create reactive Single Page Applications (SPAs) which will look similar to Mobile Application

How "Single Page Applications"? – Single HTML file & many Java Script files that changes the DOM (HTML) dynamically.

JavaScript Files

1. *inline.bundle.js*
2. *polyfills.bundle.js*
3. *styles.bundle.js*
4. *vendor.bundle.js*
5. *main.bundle.js*

Versions

There are several languages that we can use to develop Angular applications. To name a few, we have

- ECMAScript 5
- ECMAScript 6 (also called ES 2015)
- TypeScript etc.

Type**S**cript is the most popular language. **A**ngular, is built using TypeScript

Ang**ular** **CLI** – Angular Command Line Interface

- Best and Fastest way to create/develop Angular App – ng new <project_Name>
- Compile, Bundles the Files & Optimizes the Code to run in browser – ng build
    - **How ng build works internally** → **Angular CLI** internally calls **Webpack** (one of bundling tools)
        - **Webpack** internally Calls npm → to manage dependencies (project and framework related)
        - **Webpack** internally Calls **Typescript's Transpiler** → to convert .ts to .js
        - optimize the code.
        - bundles the Files
        - Places the output files in provided location

Project Setup – Steps

- Install NodeJS
- Open cmd in Admin mode and run below commands in sequence
    - npm install –g @angular/cli@latest
      ---- Downloads latest version of Angular CLI from repository and install it globally in our system
    - ng new <project_name>
      ---- Creates new Angular project with required configuration – configuration → used to compile, bundle & deploy the project
    - ng serve
      ---- Serves our Angular App for the request

Type**S**cript is a **superset** of **J**ava**S**cript – provides ***strong*** **TYPE**ing – type **check**ing at **compile time**

Basics

Setting- Up Bootstrap for Styling

- Create new project – ng new BootstrapSetUp
- Download & Install Bootstrap – npm install –save bootstrap@3
- Open **angular.json** - Angular CLI config file
  Go to styles array

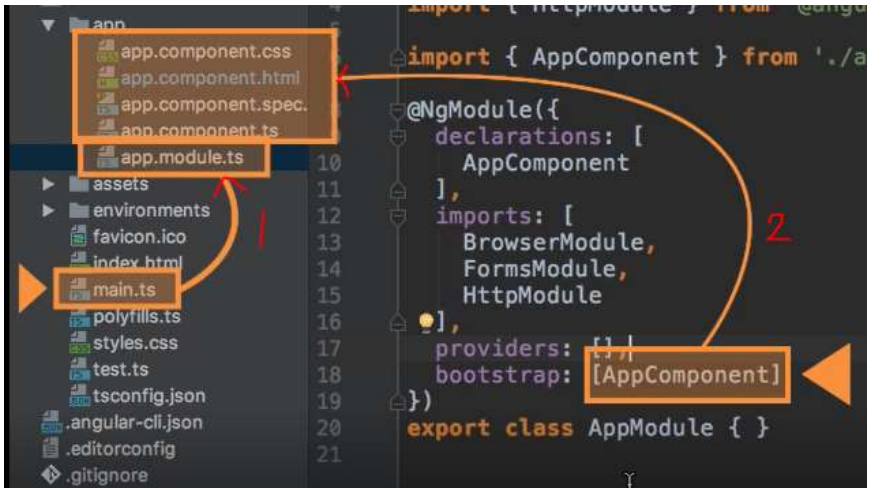
```
"styles": [
  "src/styles.css"
],
```

  Add bootstrap style file path & do save the file

```
"styles": [
  "node_modules/bootstrap/dist/css/bootstrap.min.css",
  "src/styles.css"
],
```

- To load the application with updated configuration – ng serve
- Hit the application from Browser and go to developer tools, then Elements
  You can see below style tag added for newly added bootstrap style class

How an Angular App gets Loaded and Started



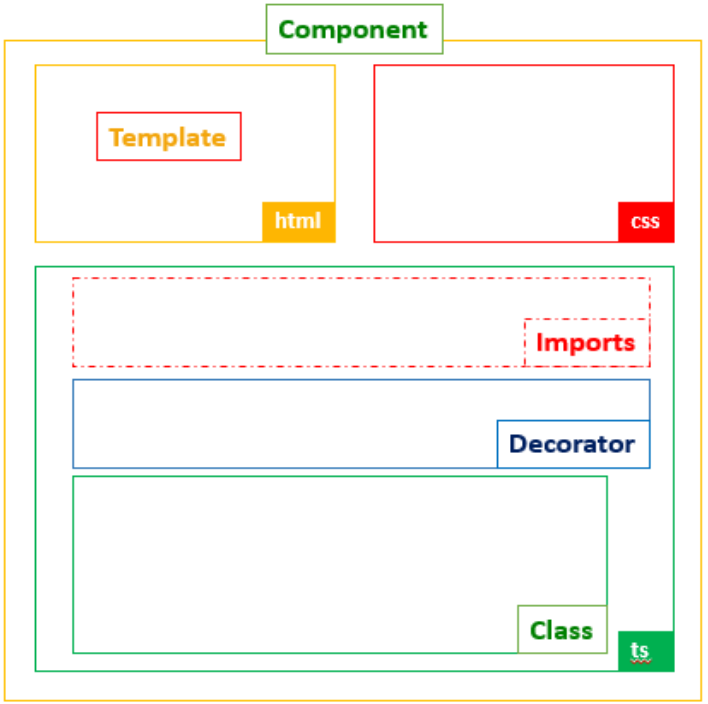
- Visit localhost:4200 from browser
- Then **index.html** contains bunch of JavaScript files will be served by server

```
<!doctype html>
<html>
<head>
  <meta charset="utf-8">
  <title>Angular - The Complete Guide</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>Loading...</app-root>
<script type="text/javascript" src="inline.bundle.js"></script><script type="text/javascript" src="polyfills.bundle.js"></script><script
type="text/javascript" src="styles.bundle.js"></script><script type="text/javascript" src="vendor.bundle.js"></script><script type="text/javascript"
src="main.bundle.js"></script></body>
</html>
```

- Then the scripts get executed immediatly
- Then the Angular App gets the information about module to bootstrap (to start with) and then components to render
- The Angular App then parses the components and replaces the custom tags with respective component code to render the page

Angular in the end is a **JS framework**, changing you **DOM('HTML')** at **Runtime**



Components are Important

- **Components** are **Building Blocks** of **Angular App**
- Components can be **Reusable**. So that
  - Application can be easily maintainable
  - Reflect/Re-use same business logic & styles in many places of Application

Creating a New Component

AppComponent - A root component i.e., Angular starts parsing/reading other components from this component

app.module.ts - bootstrap: [AppComponent] → ***AppComponent*** **is the class name**

index.html - <app-root> ....</app-root> → ***app-root*** **is the selector name of AppComponent class**

A **Component** is basically a ***typescript class*** where **Angular** uses it as blue print ***to instantiate the Objects***

Syntax:

Folder: <***component_Name***> inside **root** folder (i.e., app)

File: <***component_Name***>.component.ts

```
import {Component} from '@angular/core';
@Component ({
        selector: 'app-<component_name>',
        templateUrl:'. /<component_name>. component.html'
})
export class <Component_Name>Component {
}
```

→ **Metadata required for Angular during runtime** *(External Template File)*

→ **Typescript**

@angular/core → is a **.ts** file Name & this file consists of core functionalities of Angular (ex:- @Component)

@Component → Decorator & it accepts JavaScript object {} as configuration

→ (Decorator Syntax is from Typescript & @Component is implemented in Angular using Typescript Decorator Syntax)

{Component} → Class Name

selector → consist of tag name which will be used in other components to refer this particular component

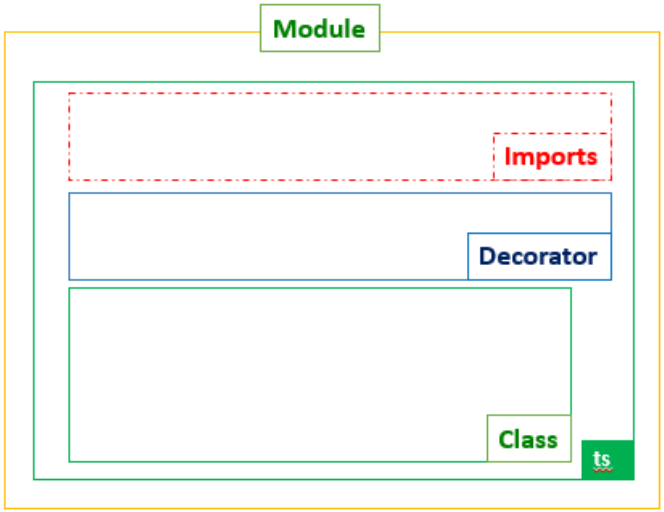
templateUrl → consist of relative path of HTML file whose code is used as replacement of selector tag at runtime

→ **(relative path : ./** <***component_Name***>.component.html)

Understanding the Role of AppModule and Component Declaration

Angular uses

- **Components – to build Web Pages**
- **Modules – to bundle different pieces/components/functionalities of the module (here - app) into packages & gives info to the Angular which features the module have to use**



Small project – Single Module (here- AppModule) ------- Big project – Multiple Modules

Module

- Is a empty Typescript class like component class
- Default Module - export class AppModule {}
    - File Name – **app.module.ts**
    - Folder Name - **app**
- It has **@NgModule** decorator on it
    - Is one of decorators from ***@angular/core*** and accepts java script object **{}**
    - This decorator tells Angular this class is a **Module**
    - **Java Script Object Properties of *@NgModule* Decorator**
        - **declarations: [] –** used to register the Components (AppComponent & Custom Component) that will be used in this Application – so that Angular scans & parses this component, otherwise not
        - **imports: [] –** allows us to use **other modules in this module** (in case of multi module projects/Applications)
            - **Some built-in Angular module**
                - **BrowserModule - @angular/platform-browser –** this module gives all the base functionalities to start our application
                - **FormsModule - @angular/forms**
                - **HttpModule - @angular/http**
        - **providers: []**
        - **bootstrap: [] –** tells the Angular which component should be used (to start parsing from) to start the whole start application and also used same component's selector in index.html file

Using Custom Components

1. Add/register **custom component's class name** in **declarations: []** of ***@NgModule*** decorator of **module's** class (**here-** AppModule {} class)
2. Add **custom component's selector** in ***app.component.html*** or its ***sub component's html*** file but not in **index.html**

Creating a New Component → Using **Angular CLI** (i.e., using ng command)

- Open terminal & enter - **ng generate component** <*component_Name*> or **ng *g c*** <*component_Name*>
    - This will create files/folders with below structure

        <*component_Name*> folder in app folder
        - <*component_Name*>.component.**css**
        - <*component_Name*>.component.**html**
        - ~~<*component_Name*>.component.**spec.ts** - Used for Testing Purpose~~ → **ng *g c*** <*component_Name*> ***--spec false***
        - <*component_Name*>.component.**ts**
            - *constructor() {}*

- *ngOnInit() {}*
- Also ***Updates*** **custom component's class name** (i.e., *<Component_Name>*) in **declarations: []** of ***@NgModule*** decorator of **module's** class (**here-** AppModule {} class)

Nesting Components **– Adding custom component's selector in other custom component's html**

Component Templates

```
import {Component} from '@angular/core';
@Component ({
    selector: 'app-<component_name>',
    templateUrl: './<component_name>.component.html' → external Template
    template: 'sub/child component's selector tag' → inline Template (to define html code as String (single/multi line) in typescript code)
})
export class <Component_Name>Component {
}
```

Note:- **Either templateUrl or template should be present but not both**

Single Quotes → Single Line String ('.....')

Back Ticks → Multi Line Strings (`.....

.....`)

Component Styles

```
import {Component} from '@angular/core';
@Component ({
    selector: 'app-<component_name>',
    template: 'sub/child component's selector tag',
    styleUrls: ['./<component_name>.component.css'] → Array of external css files
    styles: ['.....'] → Array of inline styles(single/multi line String)

})
export class <Component_Name>Component {
}
```

Note :- **Either styleUrls or styles should be present but not both**

Component Selector

```
@Component ({
    selector: 'app-<component_name>', → Should be Unique in our project and/or 3RD party Framework
    template: 'sub/child component's selector tag',
```

styleUrls: [‘. /<component_name>. component.css’]
})
export class <***Component_Name***>Component {
}

Types in Selectors - Syntax

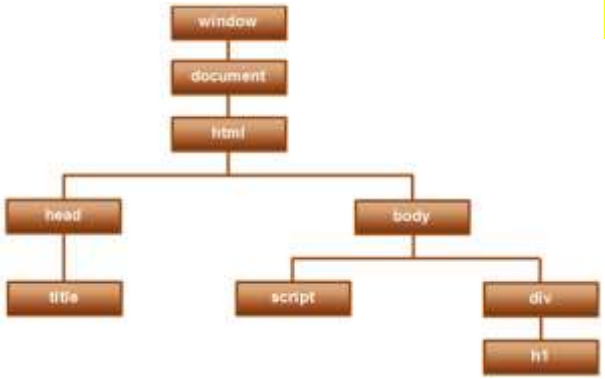
- *Element* - **selector: ‘app-<component_name>’**
- *Attribute* - **selector: ‘[app-<component_name>]’,**
- *Class* - **selector: ‘.app-<component_name>’,**
- *Id & sudeo – not supported by angular*

Data Binding = *for* Communication */for* Dynamic Content

***Communication** b/w **Typescript** (**Business Logic**) and **Template** (**HTML**)*

Types

- Output Data (*Typescript* → ***HTML***)
  - String Interpolation – **{{<expression>}}**
    - <expression> - which will be finally resolved to String
      - Component’s class (to which this html template belongs to) **property name - {{propertyName}}**
      - **Call to** Component’s (to which this html template belongs to) class **method - {{methodName ()}}**
      - String with single Quotes (constant) **- {{ ‘String’ }}**
      - **<expression> - cannot be multiline String/block code (if/for)**
      - Can be **Ternary Operator - {{ <expression>? ‘Something’: ‘Something-Else’}}**
      - Boolean, number are casted to String
  - Property Binding – **[DOM Element’s property] = ”<expression>”**
    - Here <expression> can be Component’s class (to which this html template belongs to) **property name or method call**
      Ex :- [disabled]=”propertyName” – button
      <button [disabled]='isDisabled'>Click me</button> → looks like we are binding to the Button's disabled attribute. This is not true.
      We are actually binding to the disabled property of the button object.



Property Binding is all about binding to DOM object properties and not HTML element attributes.

- HTML Attribute vs **DOM Property**
  - When a browser loads a web page, the browser creates a Document Object Model (DOM) of that page.
  - Difference between HTML Element’s Attribute and DOM property
    - Attributes are defined by HTML, where as properties are defined by the DOM.
    - Attributes initialize DOM properties. Once the initialization complete, the attributes job is done.
    - Property values can change, where as attribute values can't.

- React to (User) Events (***HTML*** ➔ *Typescript*)
    - Event Binding – **(HTML Element's event) = "<expression>"**

      Here <expression> can be Component's class (to which this html template belongs to) **method call**

      Can be code to execute on **event**

      Ex :- (input)="methodName($event)" – input

      (click)="methodName()" – button

      $event → Data emitted with that data, can be passed to method as argument from HTML template
- Two-Way Binding (combination of both Output Data & React to Events) - **[(ngModel)] = "Property Name of Component"**

  ngModel – one of the built-in directives

  ngModel – updates input's element value

Directives – Instructions in the DOM – mostly used in Attributes of HTML elements

- Example: Using selector of our component in other component HTML, instructing angular to add template & business logic of our component in other component
- Basically Directives are implemented using **@Directive Decorator** (directive **without** a template) like **@Component Decorator** (directive **with** a template) in **.ts** file
- Selector would be **attribute Selector** in Directive

```
@Directive ({
        selector: '[app<Directive_Name>]'
})
export class < Directive_Name>Directive {
}
```

- **Built-in Directives**

***ngIf="<expression>"**

- Structural Directive
- **Add/Remove** Element from DOM
- <expression> can be propertyName, methodName or expression which will be resolved to true or false

<ngTemplate #<someName>>

- Component Directive
- Used to Mark Place in the DOM
- **#<someName> - resolver/marker**

***ngIf="<expression>; else <someName>"**

[ngStyle]="{<expression>}" - Allows you to specify the **properties of style attribute of HTML Element**

- Attribute Directive
- **{<expression>} – {<styleProperty>:'StringConstant' } or { <styleProperty>:<methodName>()}**
  Ex;- {backgroundColor:'red' } or { backgroundColor:<methodName>()}

[ngClass]="{<expression>}" - Add/Remove css classes based on condition

- Attribute Directive
- **{<expression>} – {<className>:'<Boolean expression>' }**

***ngFor="let element of list"**

- Structural Directive
- **Add** Element**s** to DOM
- Getting current **index** of Iteration - ***ngFor="let element of list; let i=index"**

Note: Star (*) before directive indicates that it is a **Structural Directive** (which changes the structure of the DOM)

Square Bracket **[]** indicates that we want bind the directive to the property of the HTML Element

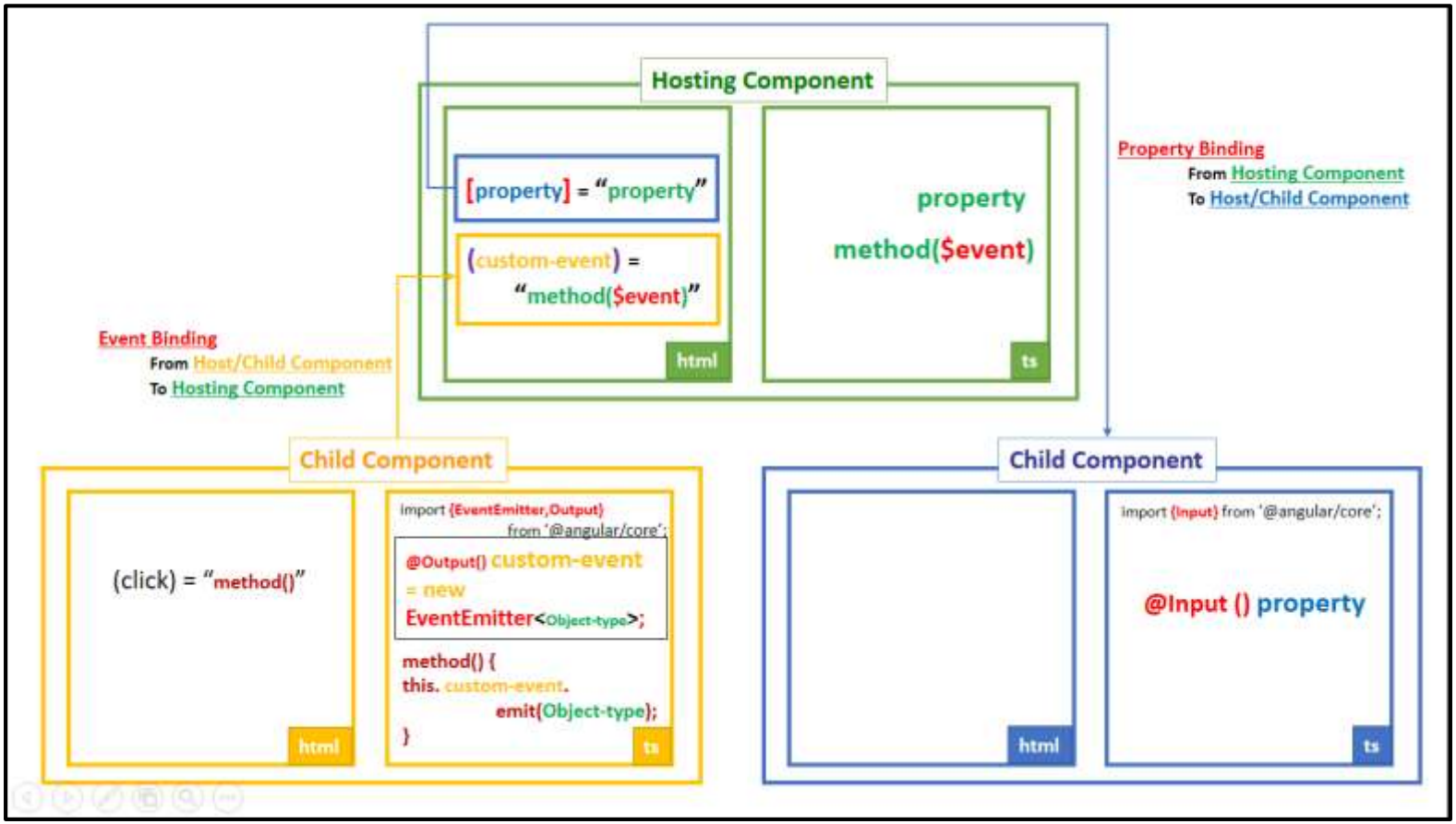
Unlike **Structural directives**, **Attribute directives** don't add or remove elements. They only change the element they were placed on

Components & Databinding

Property & Event Binding – can be used

- not only on – **HTML Elements** using **Native Properties & Events**
  - **[HTML Element's property] = "<expression>" → Property Binding**
  - **(HTML Element's event) = "<expression>" → Event Binding**
- but also on below – also to emit our own/custom events
  - Directives using **Custom Properties & Events**
  - Components using **Custom Properties & Events**

Component Communication – using Property & Event Binding



By default, the **properties** are only accessible inside the components (.ts & .html files of particular component)

- To access the property from **hosting** Component

- **@Input() /@Input('alias')** – Property Decorator – To **assign** a value **from** **hosting** component – passing data from one component to other component
- To emit the event to **hosting** Component
  - **@Output() /@Output('alias')** - Property Decorator – To **emit** the value **to** **hosting** component's property - emitting custom event from one component to other component

Note:- When **'alias'** is used then property name won't exposed to outside world but only **'alias'**

This Approach of Component Communication is **not best**, as to two child components can't communicate directly

View Encapsulation

- Angular assigns same unique **attribute** to all elements in each component & appends same unique attribute selector to that component's styles – so that Angular enforces the styles of one component won't **affect** other component – **default behaviour by Angular**
- To **override** above **default behaviour** of **View Encapsulation**

```
@Component ({
        selector: 'app-<component_name>',
        templateUrl:  '. /<component_name>. component.html',
        styleUrls: ['. /<component_name>. component.css'],
        encapsulation: ViewEncapsulation.None
})
export class <Component_Name>Component {
}
```

  - **View Encapsulation Modes**
    - **ViewEncapsulation.Emulate - DEFAULT**
    - **ViewEncapsulation.None –** Won't assigns/generates unique attribute for that component & component's styles, so that style defined in this component will be applied globally to all components
    - **ViewEncapsulation.Native –** Uses shadow DOM technology (same as **Emulate**) **–** but only in supported browsers

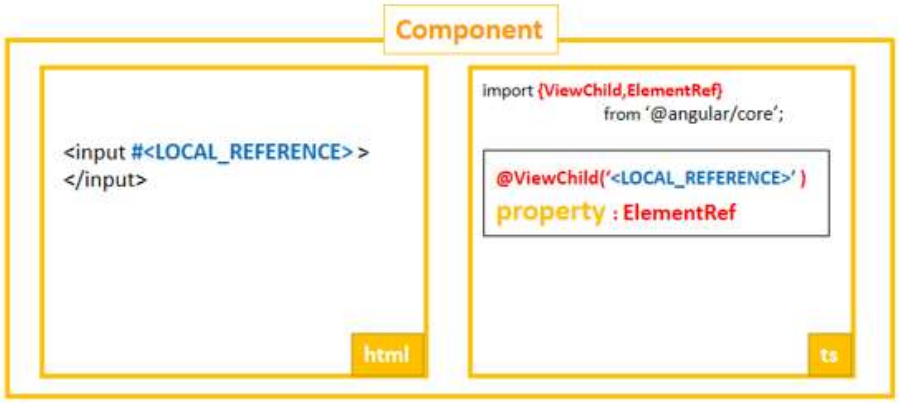
Local Reference in HTML Template

- Local Reference can be applied to all elements in the Template
- <element **#<LOCAL_REFERENCE>**>
- **#<LOCAL_REFERENCE>** holds the reference to the element
- **<LOCAL_REFERENCE>** can be used anywhere in the belong Component's Template (HTML) **but not in Typescript Code**
  ex:- **<button (click)="methodName(<LOCAL_REFERENCE>)"/> - To access element and its properties from Component's Typescript class**

@ViewChild(): ElementRef

- To access the elements (using **#<LOCAL_REFERENCE>** on element) of Component's HTML Template from corresponding Typescript **without passing it from HTML Template**

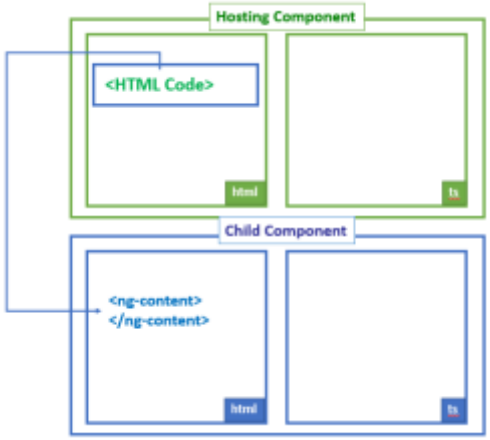
- @ViewChild('<LOCAL_REFERENCE>', {static:false})
- @ViewChild*(<Component_Name>,*{static:false}*)* – gives the first occurred element from root template



ElementRef.nativeElement = target Element

<ng-content> - Directive – To project HTML Content from Hosting Component to Hosted Component

This directive allows us to **hooks the HTML Code** present in b/w corresponding component's selector tag of **Hosting Component** to **Hosted Component's HTML Template**

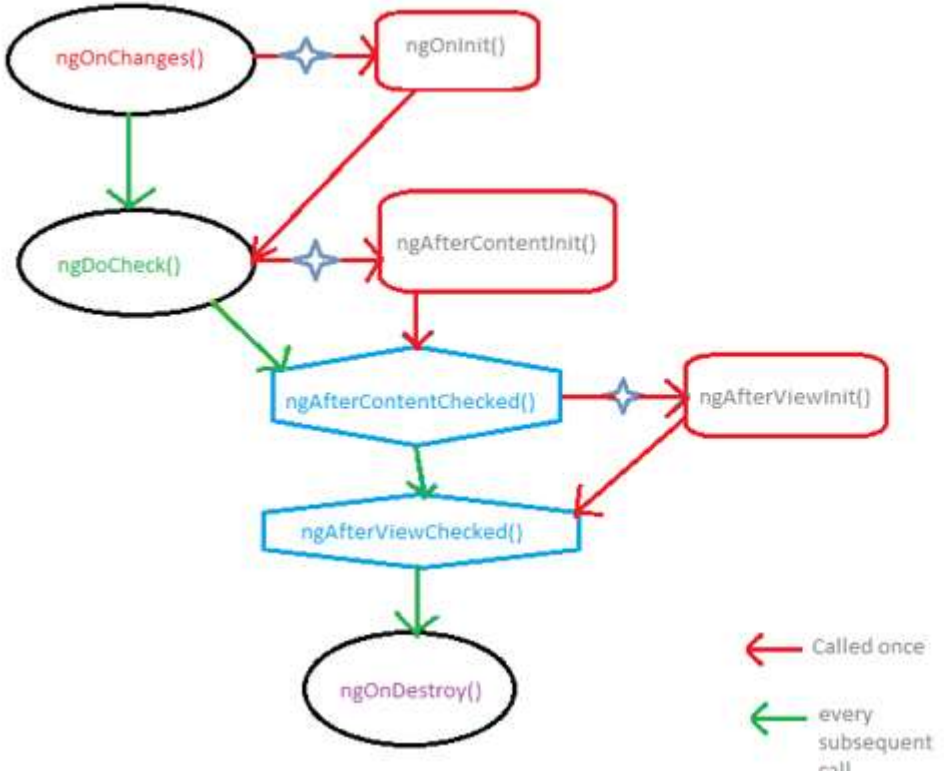


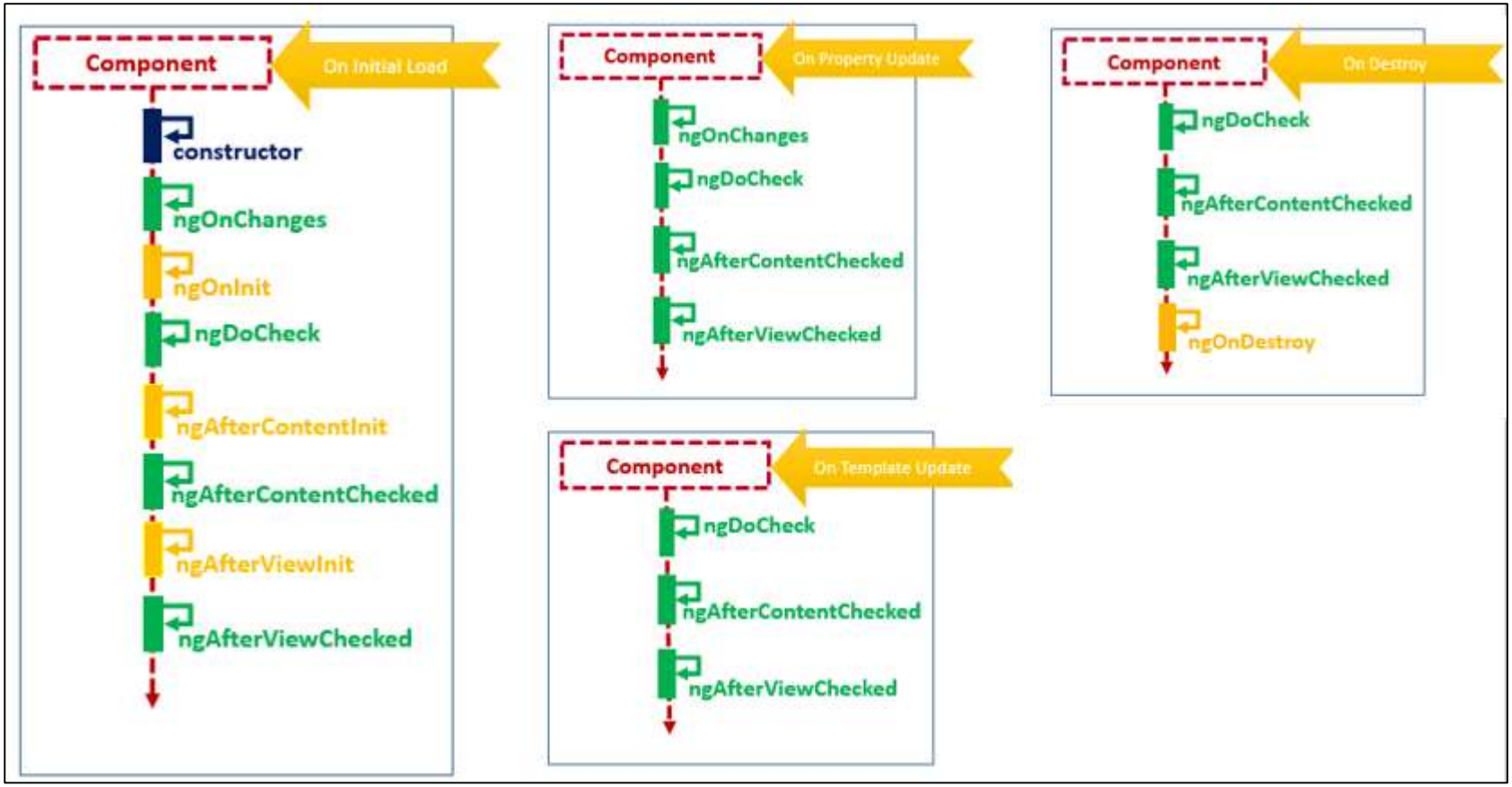
- @ContentChild('<LOCAL_REFERENCE>', {static:false}) **<variableName>**: ElementRef

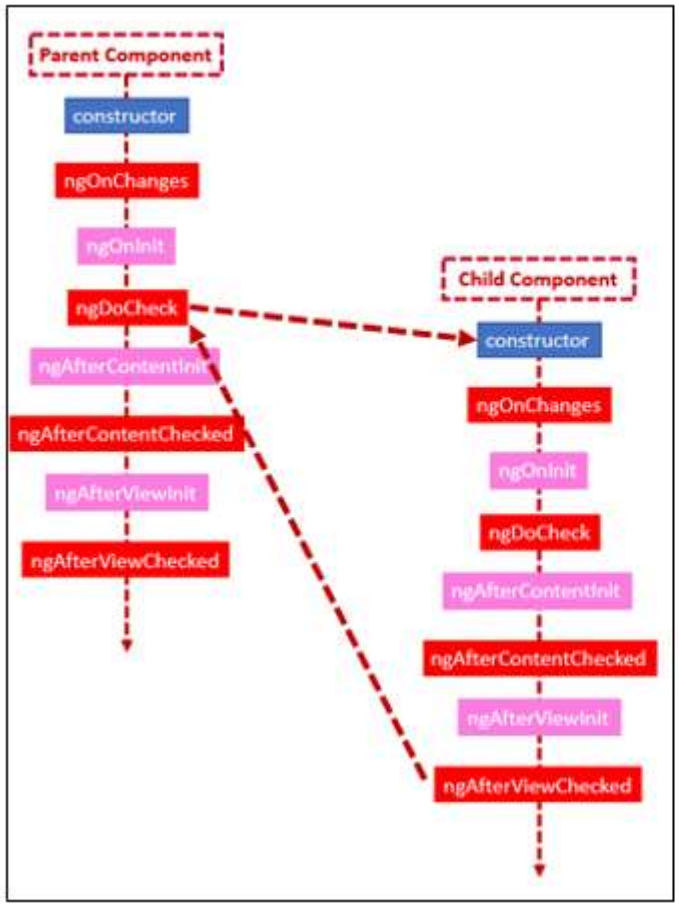
Component Life-Cycle

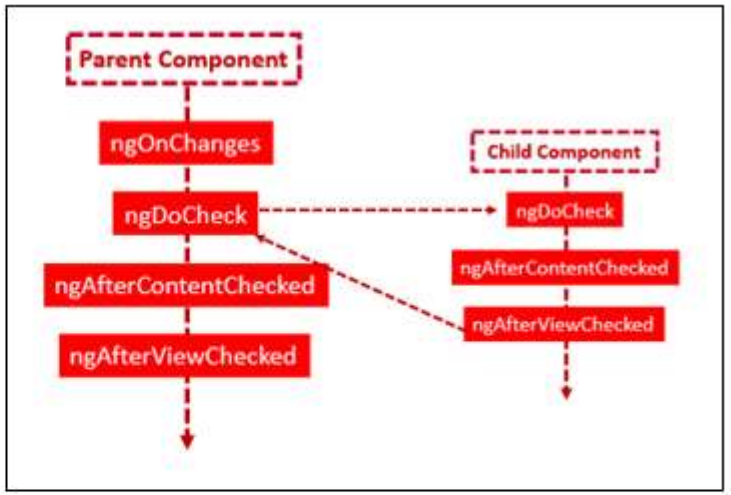
- When Angular finds a new selector tag, it instantiate the corresponding component and add it into the DOM

- Angular goes through a different phases in instantiation process
- During each phase it will execute some methods if present in that corresponding component




Component
On Initial Load
constructor
ngOnChanges
ngOnInit
ngDoCheck
ngAfterContentInit
ngAfterContentChecked
ngAfterViewInit
ngAfterViewChecked
Component
On Property Update
ngOnChanges
ngDoCheck
ngAfterContentChecked
ngAfterViewChecked
Component
On Destroy
ngDoCheck
ngAfterContentChecked
ngAfterViewChecked
ngOnDestroy
Component
On Template Update
ngDoCheck
ngAfterContentChecked
ngAfterViewChecked





- So that it gives us chance to hook and execute our code by adding to that particular method

Custom Directive

- Create a class with <filename>.directive.ts
- Add @Directive annotation on class and give selector as **[attributeSelector]**
- Inject **ElementRef** to constructor – to access the hosting html Element
- Add Directive Class Name in declarations arrays of @NgModule
- But modifying the hosting html element using ElementRef is not good practice. Instead use Renderer2
- So inject **Renderer2** along with **ElementRef** in the constructor
- Updating Hosting HTML Element Style – this.< **Renderer2 instance** >.setStyle(this.< ElementRef Instance>.nativeElement,'<styleProperty>','<value>')
- Listening to Hosting HTML Element Events

  @HostListener('<eventName>') <methodName>(<variableName>:Event) { }

e.g., @HostListener('mouseover') <methodName>(<variableName>:Event) { }

- Reading & Updaing to Hosting HTML Element Properties instead of using **Renderer2**
  @HostBinding('<HTML Element Property>') <variableName>:string;
  e.g.,@HostBinding('style.backgroundColor') <variableName>:string;
- Directive with Property – use - **@Input <PropertyName>**
  e.g., <p <CustomDirective> [<PropertyName>]=" '<String Value>' ">
- **Structural Directive**
  - **Structural Directive contains * before directive name**
  - Resolving Structural Directive by Angular – Behind the Scenes
    - Whenever angular finds * in Dire ctiveName – then Angular wraps that hosting element and childs in **<ng-template [<DirectiveName]>** before rendering
    - The ng-template is a element that won't render anything itself but render internal elements based on condition
- **Custom Structural Directive**
  - Create Directive .ts file
  - Add **@Input set <DirectiveName>(<variable>:<type>)** {<Code Here to Render Element in View>} method – this gets executes whenever property changes
  - Inject **TemplateRef<**any**>** - Generic Type
  - Inject **ViewContainerRef**
    - Here **TemplateRef - is for what to render, ViewContainerRef – is for where to render**
  - this.<**ViewContainerRef_Instance>.createEmbeddedView(**this.< **TemplateRef_Instance>) – to render Element in View**
  - this.<**ViewContainerRef_Instance>.clear() – to clear Element in View**
- **In-build Directive**
  - [ngStyle]
  - [ngClass]
  - *ngIf
  - *ngFor
  - [ngSwitch] - *ngSwitchCase, *ngSwitchDefault

Services Dependency Injection →*(new **instance** will be created by Angular & **provide** that instance where ever required)*

Services

- A Normal Typescript Class (no directives attached)
- To Implement Tasks that are to be centralized (Ex:- Logging)
- To manage Data Storage to communicate b/w Components
- File Name = ***<Name>*.service.ts**

Hierarchical Injector

- If you provide Service at **AppModule level** then same instance of Service is available to entire Application (All **Components & other Services**)
  @NgModule({ providers:***[<ServiceName>]***})
- If you provide Service at **AppComponent level** then same instance of Service is available to entire Application (All Components, **but not to Services**)
  @Component({ providers:***[<ServiceName>]***})
- If you provide Service at **Component level** then that Component & its child Component receive same instance of Service
  @Component({ providers:***[<ServiceName>]***})

Put **@Injectable** on service class (**receiving service** class)to inject other services to another service & mention service class name in Constructor

Cross Component Communication

- Add <Event Emitter variable> of type EventEmitter in Service
- Emit event by accessing < Event Emitter variable> of Service from one Component
- Subscribe to event by accessing < Event Emitter variable> of Service in Other Component

Routing using Router

- Create a **variable** of type **Routes** array
  Const <variableName>: Routes =[]
- Configuring Routes
  Const <variableName>: Routes =[{**path**:'<pathName>',**component**:<componentName>}]
  - <pathName> can be empty i.e., ('')
- Add **RouterModule.forRoot(**<Routes Array variableName>**)** in **imports** Array of **@NgModule**
- **Path Strategy** → default
- **Location Strategy - RouterModule.forRoot(**<Routes Array variableName>, {useHash:true} **)**
- Add <router-outlet></router-outlet> in component's template to render currently selected route's component
- Navigating with Router Link → Add **routerLink** directive on html element i.e.,
  - <a **routerLink** ="***/<path>***">
  - <a **routerLink** ="***/<path>***">
  - <a [**routerLink]** ="['***/<path>'***]">
- Navigation paths
  - Without leading slash - <a **routerLink** ="***<path>***"> or <a **routerLink** ="**./*<path>***"> - Relative Path & gets appended to current path
  - With leading slash - <a **routerLink** ="***/<path>***"> - Absolute Path & gets appended to IP address of server
  - With leading dots - slash - <a **routerLink** ="**../*<path>***"> - Relative Path & gets appended to existing URL by going/removing one path
- Styling Active Routes – **routerLinkActive="<value>" [routerLinkActiveOptions]="{exact=true}"**– can be on route hosting element or on hosting element's wrapper element
- Navigating Programmaticaly

- Inject **Router** to component
- this.<router_instance>.navigate([' <AbsolutePath> '])

  Note:- The **routerLink directive (since it sits in loaded template)** knows currently loaded path but not **navigate** method

  To make navigate method know about the currently loaded path
  - Inject **ActivatedRoute** to component
  - this.<router_instance>.navigate([' <AbsolutePath> '],{ relativeTo:this.<actived Route_instance> })

Configuring Route Params

- Configuring Routes to pass Params to Components

  Const <variableName>: Routes =[{**path**:'<pathName>/**:<paramName>**',**component**:<componentName>}]
- Accessing Parameters from Routes in the loaded component
  - Inject **ActivatedRoute** to component
  - this.<actived Route_instance>.snapshot.params['**paramName>']**
- Passing Params from **routerLink**
  - <a [**routerLink]** ="['***/<path>','<paramValue>']***">

    Note: Angular won't reload/re-instantiate same component when we try to load component from the same component using routerLink
- To read the params from reloaded path – Programatically using Observable
  - this.<actived Route_instance>.params.subscribe((<params>:Params)=> { <params>['<paramName>'] })
- Passing Static Data to Routes

  Const <variableName>: Routes =[{**path**:'<pathName>',**component**:<componentName>, data:{<key>:'<value>'}]
  - this.<actived Route_instance>.data.subscribe((<data>:Data)=> { <data>['<key>'] })
- Passing Params & QueryParam from **routerLink**
  - <a [**routerLink]** ="['***/<path>','<paramValue>']***" [**queryParams**]="{<key>: '<value>' } **fragment**="<fragementValue>">
- Passing Params & QueryParam Programatically
  - Inject **ActivatedRoute** to component
  - this.<router_instance>.navigate([' <AbsolutePath> '],{ queryParams: {<key> : "<value>" }, fragement:"<fragementValue>")
- Configuring Nested/Child Routes

  Const <variableName>: Routes =[{**path**:'<pathName>',**component**:<ParentComponentName>, children: [

  {**path**:'<pathName>',**component**:<ChildComponentName>}

  ] }]
  - Add **<router-outlet></router-outlet>** in <ParentComponentName> to render <ChildComponentName> in <ParentComponentName>
- Preserving QueryParams
  - this.<activatedRoute_instance>.navigate([' <RelativePath> '],{ relativeTo: this.<route_instance>,***queryParamsHandling***:'merge | preserve'})
- Redirecting Routes

Const <variableName>: Routes =[{**path**:'<pathName>',**redirectTo: <otherPathName>**} ,{**path**:' **<otherPathName>**',**component**:<componentName>}]

- **Default** Route

  Const <variableName>: Routes =[{**path**:'******',**component**:<componentName>}]

Note: In Routes Array order is very important and **default route** should be the **last** route

Configuring Route Gaurds

- Protecting Routes using **CanActivate**
  - Create Service .ts file
  - Implement **CanActivate** Interface
  - Add **canActive** method with **ActivatedRouteSnaphot** & **RouterStateSnapshot** as parameters
  - **canActive** method returns **Observable<Boolean>**| **Promise<Boolean>** | **Boolean**
  - Const <variableName>: Routes =[**path**:' **<PathName>**',**component**:<componentName>, **canActivate:[<ServiceName>]** }]

    Note:- Angular executes **canActive** method of **<ServiceName>** **before routing/rendering** **<componentName>** component
- Protecting Child Routes using **CanActivateChild**
  - Create Service .ts file
  - Implement **CanActivateChild** Interface
  - Add **canActiveChild** method with **ActivatedRouteSnaphot** & **RouterStateSnapshot** as parameters
  - **canActiveChild** method returns **Observable<Boolean>**| **Promise<Boolean>** | **Boolean**
  - Const <variableName>: Routes =[**path**:' **<PathName>**',**component**:<componentName>, **canActivateChild:[<ServiceName>]** }]

    Note:- Angular executes **canActiveChild** method of **<ServiceName>** before routing/rendering **Child Component of** **<componentName>** component
- Projecting Changing Route from One to Other using **CanDeActivate**
  - Create Service .ts file
  - Implement **CanDeactivate<T>** Interface – is a generic type
  - Add **canDeactive** method with **Component**, **ActivatedRouteSnaphot** , **RouterStateSnapshot** & **RouterStateSnapshot** (do define where to navigate next) as parameters
  - **canDeactive** method returns **Observable<Boolean>**| **Promise<Boolean>** | **Boolean**
  - Const <variableName>: Routes =[**path**:' **<PathName>**',**component**:<componentName>, **canDeactivate:[<ServiceName>]** }]

    Note:- Angular executes **canDeactive** method of **<ServiceName>** **before leaving** **<componentName>** component
- Passing Data from Route to Component Dynamically before rendering the component
  - Create Service .ts file
  - Implement **Resolve<T>** Interface – is a generic type
  - Add **resolve** method with **ActivatedRouteSnaphot** & **RouterStateSnapshot**
  - **resolve** method returns **Observable<T>**| **Promise<T>** | **T**
  - Const <variableName>: Routes =[**path**:' **<PathName>**/:<param>',**component**:<componentName>, **resolve:{<key>: <ServiceName>** }]

- this.<actividRoute_instance>.data.subscribe((<data>:Data)=> { <data>[‘<key>’] })

Passing data between components in angular

Passing data from Parent Component to Child Component

Input Properties

Passing data from Child Component to Parent Component

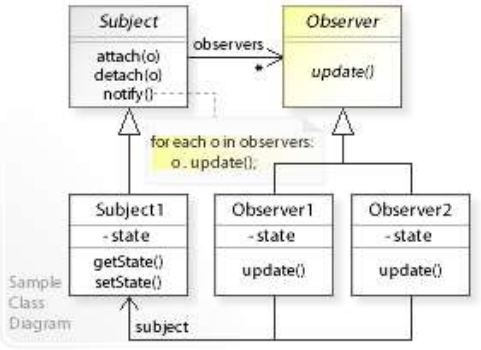
Output Properties
Template Reference Variables

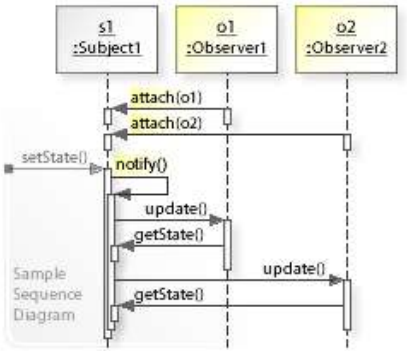
Passing data from Component to Component (No parent child relation)

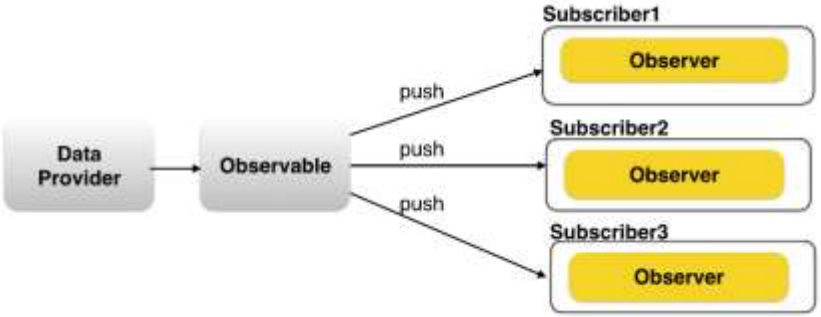
Angular Service
Required Route Parameters
Optional Route Parameters
Query Parameters

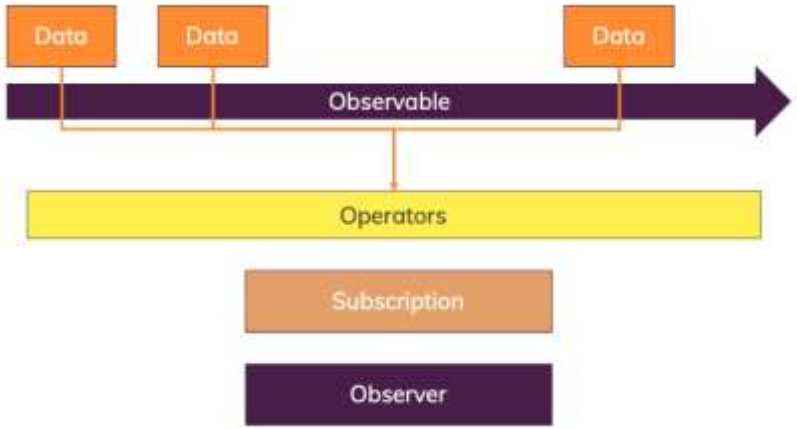
Observable – An object which we import to Angular from 3rd party Library – RxJS

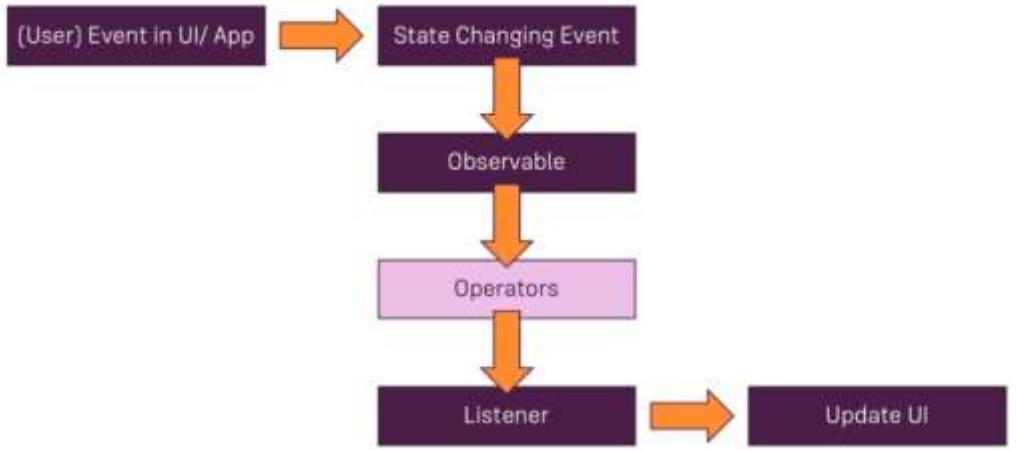
- Can be thought as Data Source - User Inputs (Events),Http Requests, Triggered in the Code etc.,
- Follows Observable Pattern







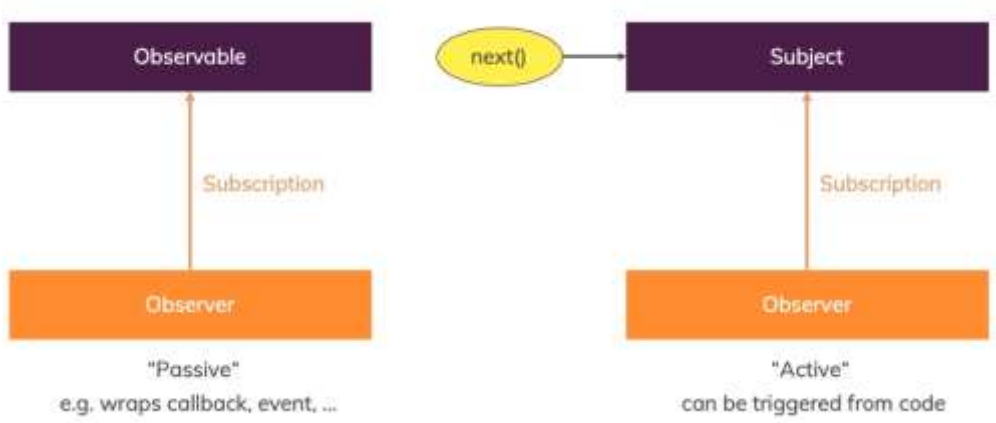




- Consists of Observable & Observer
  - Observable – Emits events or data (based on data sources) in the form data packages
  - Observer – This is our code to handle **Data** received from Observable
    - Data can be
      - Data
      - Error
      - Completion
- Subscribing to Observable → ***<Observable_instance>.***subscribe(***<data_name> => <callback_function_on_data>, <error_name> => <callback_function_on_error>, () => <callback_function_on_complete>***)
- subscribe function of ***<Observable_instance>*** Returns <**Subscription_instance**>
- Un-Subscribing from Observable → <**Subscription_instance**>.***un***subscribe()

Un-Subscribing is taken care by Angular by default for the Observables provided by Angular

- Creating Custom Observable – Syntax → **const <observable_name> = Observable.create(<observer_name> => { <code here> and *<observer_name>*.next(<data to be emitted>) | *<observer_name>*.error(new Error('<error message>')) | *<observer_name>*.complete() })**
    - *Observable automatically stops(**dies**) **emitting data** after **error** & **completed***
- **Operators** – to transform the data that is being emitted from observable to subscriber
  <Observable_instance>.pipe(***<pre-defined_operator>*** (***<data_emitted_by_observable>*** =>{ ***<code_here>*** return ***<transformed_data>;*** }))
  .subscribe(***<data_name>*** => ***<callback_function_on_data>***, ***<error_name>*** => ***<callback_function_on_error>***, () => ***<callback_function_on_complete>***)
    - Pipe method takes unlimited amount RxJS operators as arguments and executes one after the other in sequence
    - Predefined operators map, filter etc.,
- **Subject –** it is similar to EventEmitter<>()



- **<filename>.service.ts**
  <variable_name> **=new Subject<T>();**
- **<filename>.component1.ts**
  this.*<injected_service_name>*.*<variable_name>* .next(<data to be emitted>);
- **<filename>.component2.ts**
  this.*<injected_service_name>*.*<variable_name>* .subscribe(<emitted_data_name> => {<code here>});
  Mostly used in **Cross Component Communication**

Promises vs observables

Promise	Observable
Emits a single value	Emits multiple values over a period of time
Not Lazy	Lazy. An Observable is not called until we subscribe to the Observable
Cannot be cancelled	Can be cancelled using the unsubscribe() method
	Observable provides operators like map, forEach, filter, reduce, retry, retryWhen etc.

Forms

- Single Angular is Single Page Application – we won't submit the Form to server in Angular, instead it is handled by Angular using HttpService
- Angular provides JavaScript Object of the Form to access values and to validate

- Angular offers 2 Approaches to handle forms
  - Template-Driven Forms
    - Just create/implement HTML code, then the Angular infers Form object from that HTML Code (DOM) automatically
  - Reactive Forms
    - Create Form Objects and HTML Code manually & relate them manually to have greater controller on each DOM Form Element
- Template-Driven Forms
  - Import **FormsModule** to **imports** property of **@NgModule** director of **<filename>.module.ts** file
  - **<filename>.component.html**

    <form **(ngSubmit)** ="***<method_Name>***(***<resolvername>***)" #***<resolvername>***="ngForm">
    <input **ngModel** name="***<field_Name>***"></input>
    </form>
    Don't use html's <input type="submit"> as this will submit entire form to server
  - **<filename>.component.ts**

    Instead of **(ngSubmit)** directive on <form> tag

We can directly get from references using @ViewChild as below
@ViewChild('*<resolvername given in html>*') <variable_name>: **NgForm**;

Validations

- Validators used in Template Driven Form are Angular Directives
- **<filename>.component.html**

```
<form (ngSubmit) ="<method_Name>(<resolvername>)" #<resolvername>="ngForm">
<input ngModel name="<field_Name>" required></input>
</form>
```

- Angular updates **valid** property of **NgForm Javascript Object** to true if Form is valid otherwise false
- **NgForm** is a wrapper around the **FormGroup**
- **FromGroup** holds the group of **FormControl**s
- The valid property available at NgForm level & Per Control level too
- Also angular adds some classes (**ng-dirty, ng-touched, ng-untouched, ng-valid, ng-invalid,ng-pristine, ng-pending**) to html element of form

Grouping ngModel Controls

- **<filename>.component.html**

```
<form (ngSubmit) ="<method_Name>(<resolvername>)" #<resolvername>="ngForm">
    <div ngModelGroup="<group_keyName>">
        <input ngModel name="<field_Name>"></input>
    </div>
</form>
```

Getting access to ngModelGroup JavaScript object in HTML

- **<filename>.component.html**

```
<form (ngSubmit) ="<method_Name>(<resolvername>)" #<resolvername>="ngForm">
    <div ngModelGroup="<group_keyName>" #<ref-name>="ngModelGroup">
        <input ngModel name="<field_Name>"></input>
    </div>
</form>
```

- **S**etting of all Form Fields with Data → this.<formName>.**setValue**(<Form JavaScript Object with values>)
- **S**etting only empty Form Fields with Data → this.<formName>.**patchValue**(<Form JavaScript Object with values>)
- **G**etting of Form Fields values → <variableName> = this.<formName>.**value**.<fieldname>
- **Res**etting of all Form Fields→ this.<formName>.**reset()**

- Reactive Forms
  - **<filename>.component.ts**
    - Create variable as **<variable_name>: FormGroup;**

- import { FormGroup, FormControl,Validators,FormArray} from '@angular/forms';

- **<filename>.module.ts**
  - import { ReactiveFormsModule} from '@angular/forms';
  - Add ReactiveFormsModule **to** imports array of @NgModule **Directive**
  - Creating Empty Form
    - **<filename>.component.ts**
      - **<form_name> = new FormGroup({})**
  - Creating Form Controls in Form Group
    - **<filename>.component.ts**
      - **<form_name> = new FormGroup({**
        '<field_name>' : new FromControl()

        })
      - The FormControl takes below three **optional** (?) parameters
        - Initial State(Default Value)| null(empty)
        - Validator Function|Validator Function Array,
        - Async Validator Function|Async Validator Function Array

  - Attaching FormGroup of .ts to .html file
    - Add property binding as – <form **[formGroup]**=" **<form_name>**">
  - Attaching FormGroup & FormControlls of .ts to .html file
    - Add property binding as –
      <form **[formGroup]**=" **<form_name>**">
      <input **formControlName="<field_name>"**>
      Or
      <form **[formGroup]**=" **<form_name>**">
      <input **[formControlName]="'<field_name>'"**>
  - Submitting the Form
    <form [formGroup]=" <form_name**>**" **(ngSubmit)="*<method_name>*()**">
  - Adding validations to Form
    - **<filename>.component.ts**
      - **<form_name> = new FormGroup({**
        '<field_name>' : new FromControl(null, Validators.<validatorName>)

        })

        - **<form_name> = new FormGroup({**
          '<field_name>' : new FromControl(null, [Validators.<validatorName>, Validators.<validatorName2>])

          })
- Displaying validations error messages

  <span *ngIf="!**<form_name>.get('<field_name>').valid && <form_name>.get('<field_name>').touched**"> Some Message Here
- Grouping Form Controlls
- Array of Form Controlls
  - **<filename>.component.ts**
    - **<form_name> = new FormGroup({**
      '<field_name>' : new FromControl()
      '<field_name>': new FormArray([])
      })
  - **FormArray takes below**
    - **Controlls Array**
    - **Validator Function**
    - **Async Validator Function**
  - **<filename>.component.html**

    <div formArrayName="<field_name>">
- Creating Custom Validators
  - Validator is just a function – that will get executed when the control is changed
    - **<validator_name>(<controlName>:FormControl): {[<errorCodeType>:string]:boolean} {**

      If <validation_condition> {

      return {'<errorCode>':true}

      }

      return null;

      }
- Creating **Async** Custom Validators

  <validator_name>(<controlName>:FormControl): Promise<any>|Observable<any> {

  const <variableName>=new Promise<any>((resolve,reject)=> {

  If <validation_condition> {

  resolve({'<errorCode>':true});

  }

  resolve(null);

```
        });
        Return <variableName>;
    }
```

- ValueChanges – observable
  - this.**<form_variable>**.**valueChanges**.subscribe(**<callback_function>**)
  - this.**<form_variable>**.**<formControl>**.**valueChanges**.subscribe(**<callback_function>**)
- statusChanges – observable
  - this.**<form_variable>**. **statusChanges**.subscribe(**<callback_function>**)
  - this.**<form_variable>**.**<formControl>**. **statusChanges**.subscribe(**<callback_function>**)
- setValue – sets values in all given fields
  - this.**<form_variable>**.**setValue** (**<Form JavaScript Object with values to set>**)
  - this.**<form_variable>**.**<formControl>**.**setValue** (**<Form JavaScript Object with values to set>**)
- patchValue – sets values in given fields, only if they are empty
  - this.**<form_variable>**.**patchValue** (**<Form JavaScript Object with values to set>**)
  - this.**<form_variable>**.**<formControl>**. **patchValue** (**<Form JavaScript Object with values to set>**)
- The following are some of the useful properties provided by the **AbstractControl** class
  - value
  - errors
  - valid
  - invalid
  - dirty
  - pristine
  - touched
  - untouched
- **FormControl** instance tracks the value and state of the **individual html element** it is associated with
  FormGroup instance tracks the value and state of all the form controls in it's **group**
- To access a FormControl in a FormGroup, we can use one of the following 2 ways.
  <form_group_name>.**controls**.**<form_control_name>**.value
  <form_group_name>.**get**(**<form_control_name>**).value
- AbstractControl also provides the following methods.
  - setValidators()
  - clearValidators()
  - updateValueAndValidity()
  - setValue()
  - patchValue()

- Reset()

- **Formbuilder**
  - create instances of a FormControl, FormGroup, or FormArray.
  - It reduces the amount of code we have to write to build complex reactive forms.
  - The FormBuilder service has three methods:
    - control() - Construct a new FormControl instance
    - group() - Construct a new FormGroup instance
    - array() - Construct a new FormArray instance

Pipes - Used to transform the Output – ng g pipe **<PipeName>**

https://angular.io/api

- **<filename>.component.html**
  - **{{ <variableName> | <pipeName> : <firstArgument> : <secondArgument> }}**

- Chaining multiple Pipes
  - Pipes are parse from left to right
  - Output of left Pipe is Input to next Pipe
  - **<filename>.component.html**
    - **{{ <variableName> | <pipeName1> : <firstArgument> : <secondArgument> | <pipeName2> : <firstArgument> : <secondArgument> }}1**

- Custom Pipes
  - **<filename>.pipe.ts**

```
import { PipeTransform } from "@angular/core";
@Pipe({
        name: <PipeNameToBeUsedInHTMLFile>
        })
export class <CustomFilterPipeName> implements PipeTransform {
        transform(<value>:any) {
                return <transformedValue>;
        }
}
```

- Add pipe to **declarations:[]** arrays of **NgModule** directive of **<filename>.**module.ts
- Pure & Impure Pipe -- this pipe implementation used in case of Filtering

```
@Pipe({
    name: <PipeNameToBeUsedInHTMLFile>,
    pure: false
```

```
})
```

- Async Pipe {{ **<observable or promise variable name>** | async }}

Making Http Requests

- Backend Interaction using Http
- Connecting Angular to DB (SQL | NoSQL)– to store & fetch data
  - Without storing credentials of DB in Angular App
    - Angular App ←**Http Request | Http Response** → Server (API – REST) → DB (SQL |NoSQL)
- Http Request
  - URL (API End Point) → /post/**param**
  - Http Verb – GET, PUT, POST,PATCH
  - Headers (MetaData) – e.g.:- {"Content-Type" :"application/json"}
  - Body – e.g.:- {title:"Data"}
- Sending Post Request
  - **<filename>.module.ts**

```
Import {HttpClientModule} from '@angular/common/http';
@NgModule ({
        Imports:[ HttpClientModule ]
})
Export class <ModuleName> {}
```

  - **<filename>.component.ts**

```
Import {HttpClient} from '@angular/common/http';
@Component ({
        })
Export class <ComponentName> {
        Constructor(private http: HttpClient) {}
        <methodName>() {
                this.http.post('<URL>',<dataToBeInBody>).subcribe(responseData => {});
        }
}
```

Note:- Angular HttpClient converts our JavaScript Oobject to JSON automatically

- Angular uses Observables – HttpRequest are managed by Observable in Angular
- Angular wraps HttpRequest in Observable, if no one subscribing to it then **No Request** is being sent
- Here post method of HttpClient returns the Response in Observable

- GET Request

- **<filename>.component.ts**

```
Import {HttpClient} from '@angular/common/http';
@Component ({
    })
Export class <ComponentName> {
    Constructor(private http: HttpClient) {}
    <methodName>() {
        this.http.get('<URL>').subcribe(responseData => {});
    }
}
```

- RxJS Operator to transform data
  - **<filename>.component.ts**

```
Import {HttpClient} from '@angular/common/http';
Import {map} from 'rxjs/operators';

@Component ({
    })
Export class <ComponentName> {
    Constructor(private http: HttpClient) {}
    <methodName>() {
        this.http.get('<URL>').pipe(map(responseData =>{})).subcribe(responseData => {});
    }
}
```

- Using Types
  - **<filename>.component.ts**

```
Import {HttpClient} from '@angular/common/http';
Import {map} from 'rxjs/operators';

@Component ({
    })
Export class <ComponentName> {
    Constructor(private http: HttpClient) {}
    <methodName>() {
        this.http.get<{ [<someName>:keyType]:valueType}>('<URL>').pipe(map(responseData
=>{})).subcribe(responseData => {});
```

```
        }
    }
```

- **<filename>.component.ts**

```
Import {HttpClient} from '@angular/common/http';
Import {map} from 'rxjs/operators';

@Component ({
    })
Export class <ComponentName> {
    Constructor(private http: HttpClient) {}
    <methodName>() {
        this.http.post<{ [<someName>:keyType]:valueType}>('<URL>',<dataToBeInBody>).pipe(map(responseData
=>{})).subcribe(responseData => {});
    }
}
```

- Showing a loading Indicator
- DELETE Request
  - **<filename>.component.ts**

```
Import {HttpClient} from '@angular/common/http';
@Component ({
    })
Export class <ComponentName> {
    Constructor(private http: HttpClient) {}
    <methodName>() {
        this.http.delete('<URL>').subcribe(responseData => {});
    }
}
```

- Handling Errors – Bug in Program, Server is Offline, Server Error, Not Authenticated
  - **<filename>.component.ts**

```
Import {HttpClient} from '@angular/common/http';
@Component ({
    })
Export class <ComponentName> {
    Constructor(private http: HttpClient) {}
    <methodName>() {
```

this.http.get('<URL>').subcribe(responseData => {}, **error => {<variableName>=error.message}**);

}

}

- Using Subjects to Handle Errors
- Using **catchError** Operator to Handle Errors

Import {catchError} from 'rxjs/operators';

Import {throwError} from 'rxjs';

<methodName>() {

this.http.get('<URL>').catchError(**<errorResponseVariableName>**=>{return throwError(<**errorResponseVariableName>**)});

}

Note – **throwError** yields/produces Observable by wrapping error

- Setting Headers

Import {HttpHeaders} from '@angular/common/http';

<methodName>() {

this.http.get('<URL>',{

<headers>:new HttpHeaders({

<key_value_pairs> })

})

}

- Query Params

Import {HttpParams} from '@angular/common/http';

<methodName>() {

this.http.get('<URL>',{

<params>:new HttpParams().set('**<key>**':'**<value>**')

})

}

Note:- HttpParams is an immutable object

- Observing the response type → observe: 'response' | 'body' (default –converts to json by default) | 'events' (event.type – HttpEventType)
- Operator – tap – used to perform operation on given data but the modify the given data
- responseType: 'text'|'json'
- **Request Interceptors**
  - **<filename_interceptor>.service.ts**

Import {HttpInterceptor, **HttpHandler**} from '@angular/common/http';

@Component ({

})

```
Export class <ComponentName> implements HttpInterceptor {
        intercept(req:HttpRequest<any>,next: HttpHandler) {
                return next.handle(req);
        }

}
```

Here next:HttpHandler is a function which will be called after executing intercept method to let req:HttpRequest to continue its journey

- **<filename >.module.ts**

```
@NgModule({
        providers:[{provide:HTTP_INTERCEPTORS, useClass:<ClassNameToBeExecuted>,multi:true}]
        })
```

Here **HTTP_INTERCEPTORS** is a token to which we provide classes to execute and these classes will execute one after the other
Here **multi:true** is used to inform angular that we multiple classes to execute with given token other wise angular will override the class

- **Response Interceptors**
  - **<filename_interceptor>.service.ts**

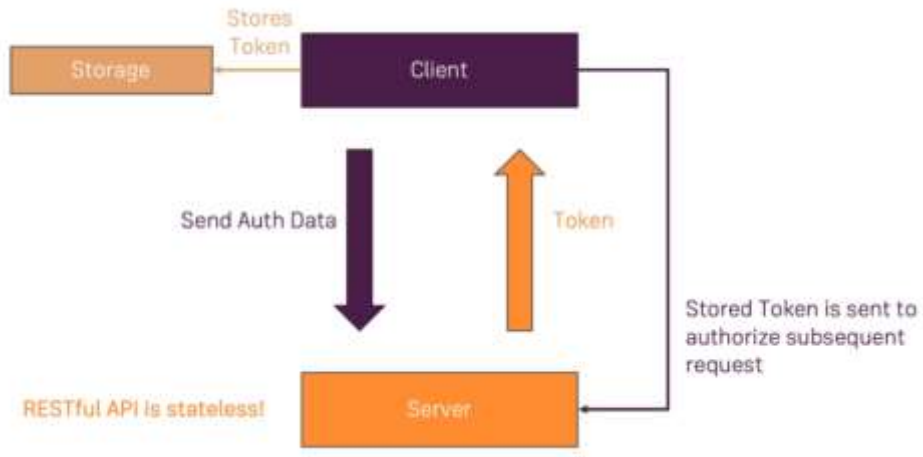
```
Import {HttpInterceptor, HttpHandler} from '@angular/common/http';
@Component ({
        })
Export class <ComponentName> implements HttpInterceptor {
        intercept(req:HttpRequest<any>,next: HttpHandler) {
                return next.handle(req).pipe();
        }

}
```

Here we can use same interceptor for Response too, since ***handle*** method returns observable –so we use ***pipe*** to modify the response

Authentication

- How it works



*Token contains encoded information (encoded by server) + private key, which can be only understood by server

- **BehaviourSubject – Observable – used to get all the data available before subscription**
- **take – operator – used with BehaviousSubject Observable**
- **Casting String to Integer → +<StringValue>=<IntegerValue>**
- **exhaustMap – operator – used replace wrapped Observable with its Observable**
- **HasOwnProperty**
- **SpreadOperator (…) → to clone JSON object →{…<Object_Instance_name>}**
- **Unknown key & value names but known types – { [<someName>:keyType]:valueType}**
- **Persisting State using browser localStorage – localStorage.setItem("key","value");**
- **AuthGuard**
  - **auth.guard.ts**

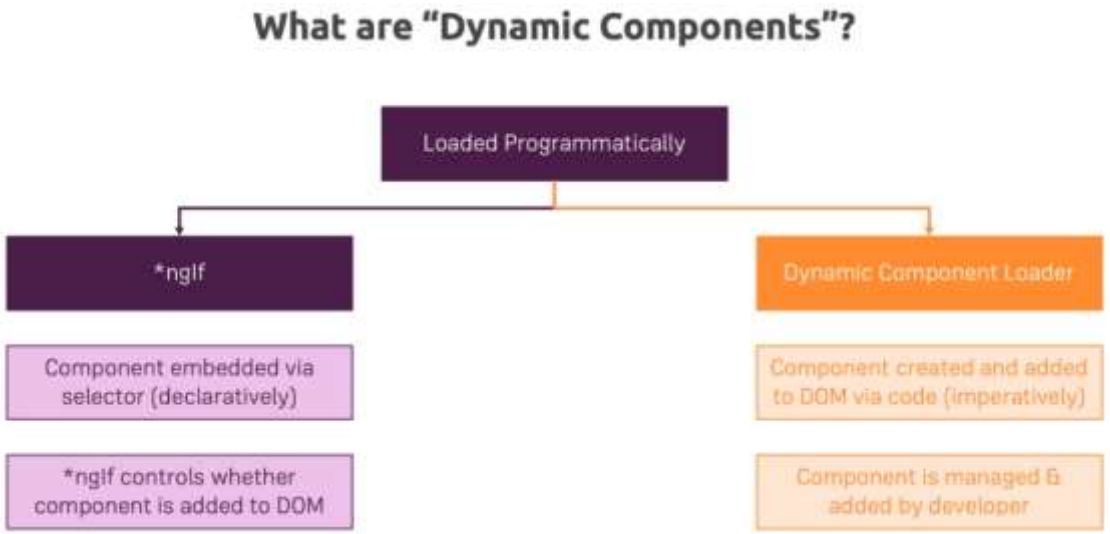
```
Import {CanActivate} from '@angular/router';
Import { Injectable } from '@angular/core';

@Injectable({providedIn:'root'})
```

```
Export class AuthGuard implements CanActivate {
    canActivate(route:ActivatedRouteSnapShot,router:RouterStateSnapShot):Boolean|Promise<Boolean>|Observable<Boolean> {
        }
}
```

- **<filename>.module.ts**
  - **Add canActivate:[AuthGuard]** Param to json objects of **Routes** array

Dynamic Components



- Using Dynamic Component Loader
  - Inject ***ComponentFactoryResolver*** into constructor
  - this.<**componentFactoryResolverInstance**>.resolveComponentFactory(<**ComponentName**>) – **this will return componentFactory Instance**
  - Create a attribute selector Directive & inject **ViewContainerRef** as public
  - Add <ng-template> in build directive in html – in which we want to load component dynamically
  - Add created attribute selector in that <ng-template>
  - Add @ViewChild("DirectiveClassName", {static:false}) <variableName>: DirectiveClassName; in html's component class
  - Get Directive's ViewContainerRef Instance as - <variableName>.<viewContainerRef>;
  - And call clear method on that instance - <variableName>.<viewContainerRef>.clear(); - to remove the component's template from hosted component's template
  - <variableName>.<viewContainerRef>.createComponent();
  - Behind the scenes of Angular – while creating Components
    - If Angular finds any component's selector in any html templates – then Angular checks declarations array of @NgModule and creates that component

- If Angular finds any component name in Routes Array – then Angular checks **declarations** array of @NgModule and creates that component
- Hence Angular does not reach out to declarations arrays to load & instantiate Components
- Add component class name (components to be loaded dynamically) in **entryComponents** array of @NgModule – to make ready Angular to load and create component when prompted Programatically

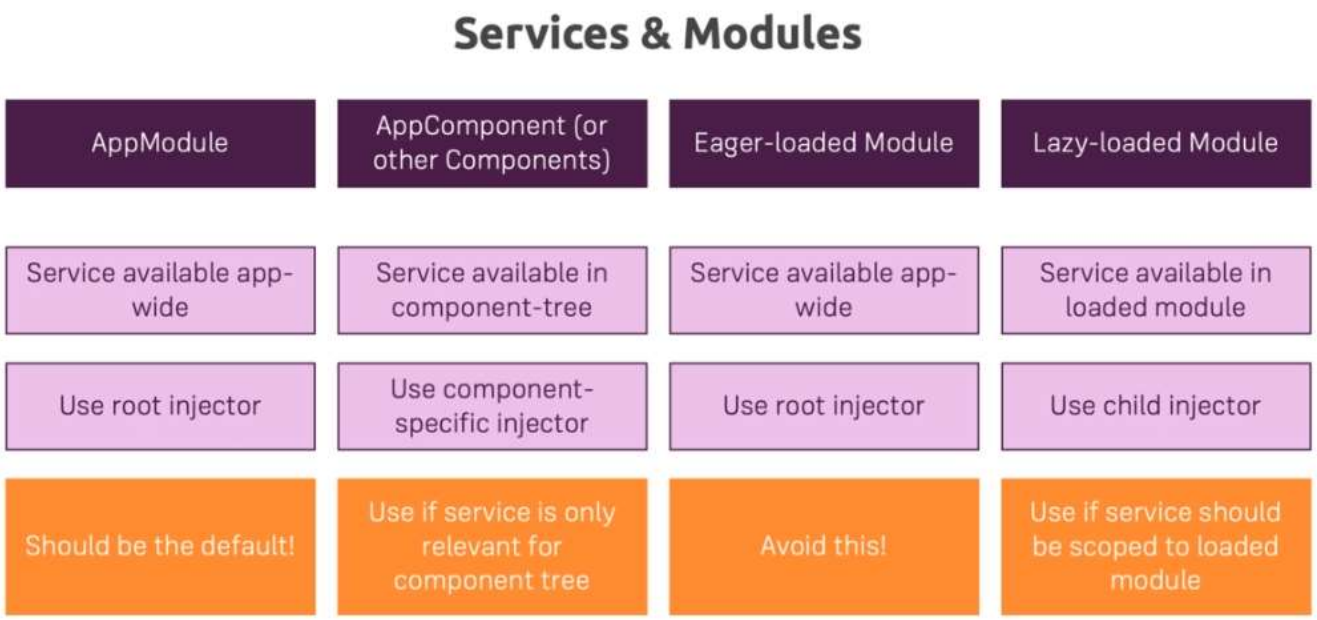
  - Inputting value to **dynamically loaded component** - <variableName>.<viewContainerRef>.**instance**.<variableName>=<Value>
  - Subscribing to Event Emitted from **dynamically loaded component** - <variableName>.<viewContainerRef>.**instance**. .<variableName>.subscribe({})

Modules & Optimizations

- Module is way of packaging/bundling angular app building blocks (Components, Directives, Services)

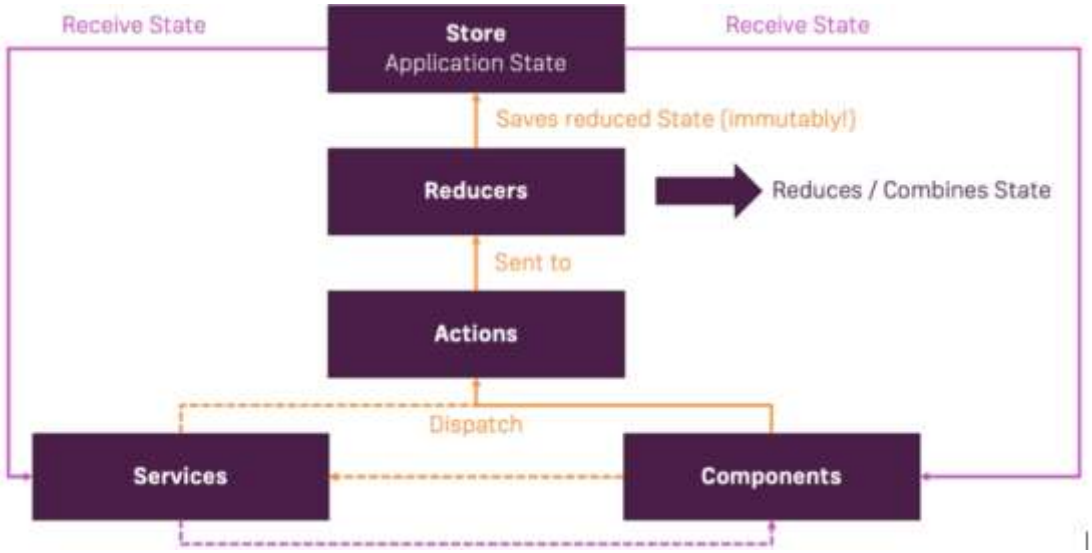
Add {**path**:'<pathName>', **loadChildren**:<Module_Path>**#**<Module_Name>} to Routes Array of **parent** Module

Add {**path**:' ', **component**:<Component_Name>} to Routes Arrays of **child** Module



NgRx – An Angular Implementation of Redux pattern for State (Data) Management that is used by Application

Redux Pattern -



- Angular provides Injectable Services to Application State in any part of Application
- All the stored state is managed as one large Observable.

Debugging

Angular Error Messages → Developer Tools – Console (Only for **Error Messages** but not for Logical Errors)

Debugging Code in the Browser Using Sourcemaps → Developer Tools – Sources – main.bundle.js (for both **Error Messages** & **logical Errors**)

- **JavaScript Files** supports **SourceMaps**
- **SourceMaps** are additions which we will added by **Angular CLI** to the **bundles** (JavaScript Files) → This allows the **browser** to **map JavaScript Code** to **Typescript Code**
- Instead of accessing Typescript files from JavaScript – We can directly access our files
    - Developer Tools – Sources –webpack:// - dot (.) folder – src folder – app folder (here you can find all our typescript files directly)

Angular Augury – A tool to Debug Angular App

- Steps
    - https://augury.rangle.io/
    - It is chrome extension

- Developer Tools – Augury
- Reload Our App

Note: - In this tool we can see – Components, Routes, Injector Graph etc.,