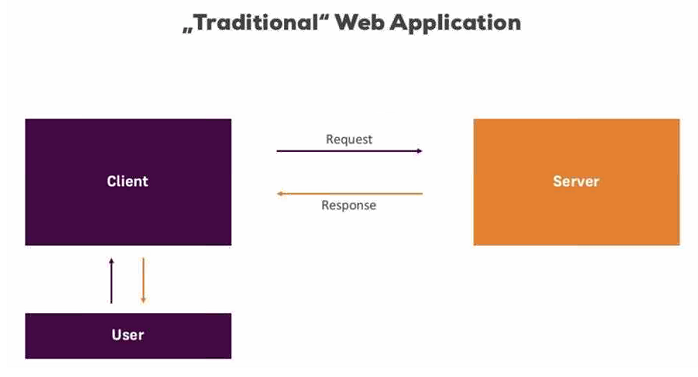
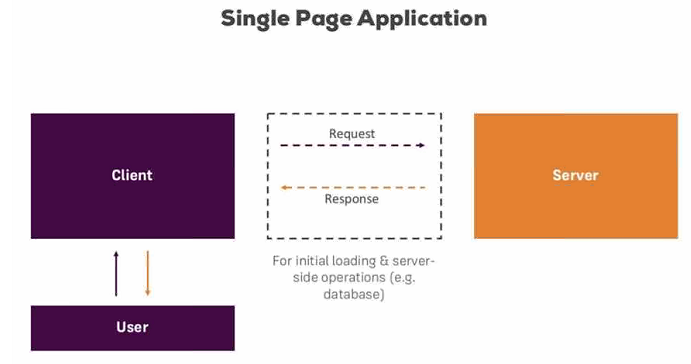
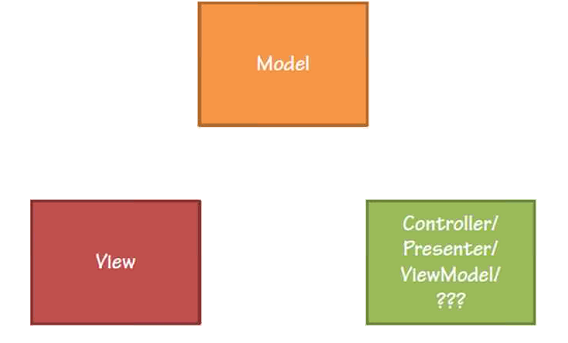
**Introduction:**

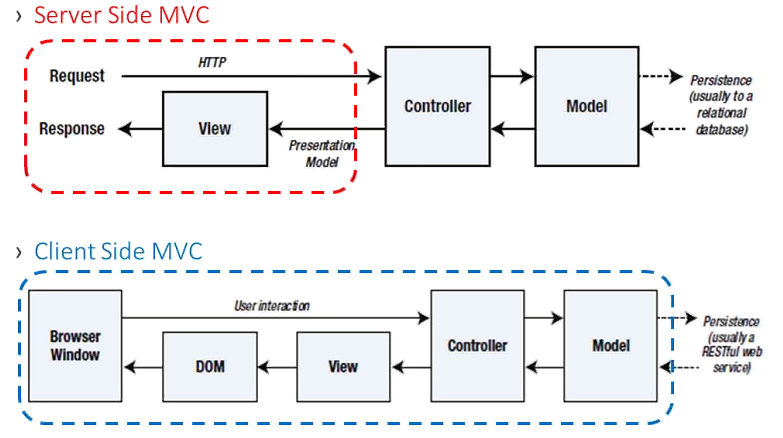
Angular is a JavaScript Framework which allows us to create Reactive Single Page Application.



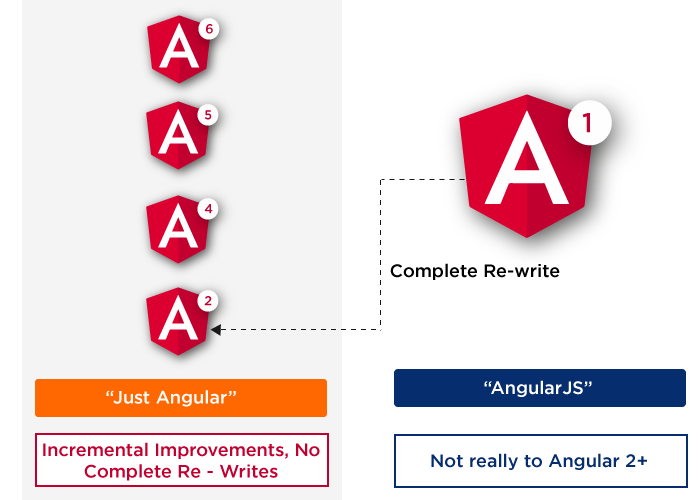


Angular has Model View Whatever structure





**Angular Versions**



**Environment Setup**

* Install the Angular CLI by using the following command.

***npm install -g @angular/cli***

* Once you installed just use the below cmd for the verification

***ng -v***

* It will give you the installed version details

If you installed Angular Already means, uninstall to avoid version conflicts

***npm uninstall -g*** ***angular-cli @angular/cli***

***npm cache clean --force***

* Navigate to where you want to create your Angular project by using cd cmd.
* Run the following cmd to create your 1st project.

***ng new your-project-name***

* Once your project setup, run the below cmd

***ng serve -o***

* Now you can run your project in (By default the port is 4200)

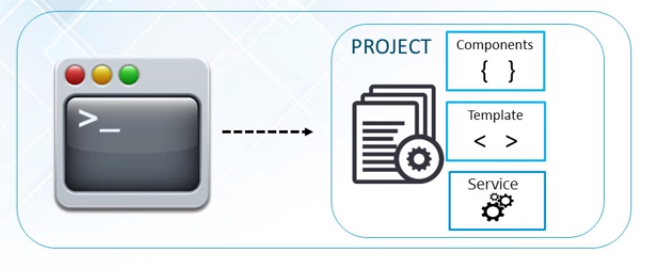
[***http://localhost:4200/***](http://localhost:4200/)

* If you want to change the port, run the following command

***ng serve --port 4201 -o***

First let's understand why should we use Angular CLI and what problems it solves.

CLI stands for Command Line Interface.



1.Create a separate application folder and add the package definition file ( ie. package.json) and other configuration files.

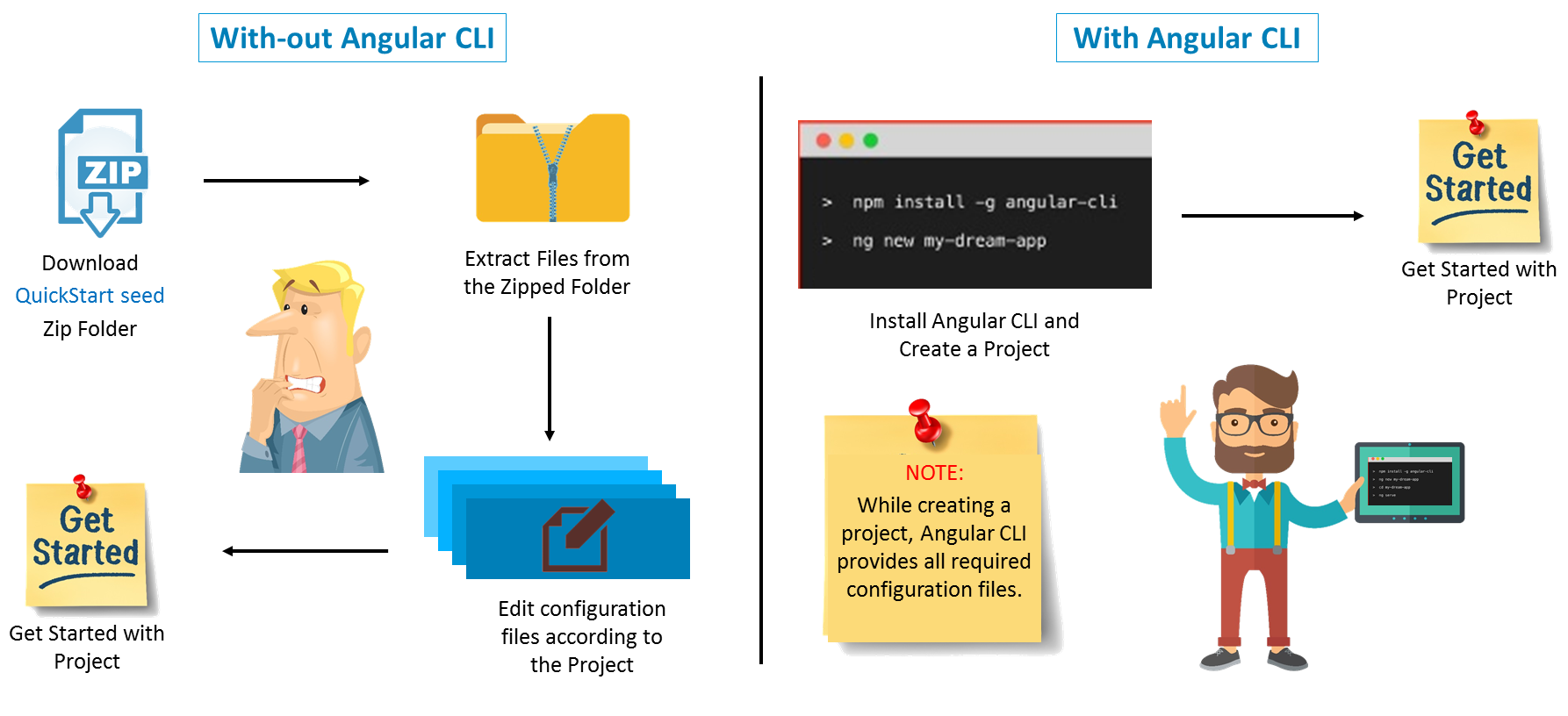
2. Install the packages using NPM

3. Setup the environment.

4. Provides required files from angular program.

5. Create index.html which hosts our application.

6. Reduces the development effort



**Features of Angular**

1. **High performance**

It gives high performance, offline, and zero-step installation. You can get 10\* performance.

**2. Single language for both platforms**

You can build native mobile apps with strategies using Ionic Framework, NativeScript, and React Native.

**3. Code Generation**

Angular giving you all the benefits of hand-written code with the productivity of a framework by using Angular CLI.

**4. Universal**

You can use any technology with Angular for serving the application like node.js, .NET, PHP and other servers.

**5. Code splitting**

Angular apps load quickly with the new Component Router, which delivers automatic code-splitting, so users only load code required to render the view they request.

**6. Templates**

Quickly create UI views with simple and powerful template syntax.

**7. Angular CLI**

Command line tools: You can easily and quickly start building components, adding components, testing them, and then, instantly deploy them using Angular CLI.

**8. IDE**

Get intelligent code completion, instant errors, and other feedback in popular editors and IDEs like Microsoft’s VS Code.

**Structure of Angular Project**

Inside the generated folder, you’ll find the following top-level folders:

* **e2e**: includes end-to-end tests.
* **node\_modules**: all the third-party libraries that our project is dependent upon.
* **src**: the actual source code of our Angular application.

99.9% of the time you’ll be working with the files inside the src folder. But let’s quickly overview the other files we have in this project:

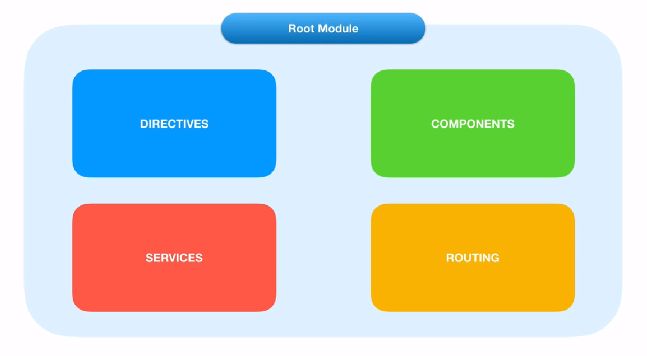
* **package.json**: a standard file for Node-based projects. It contains metadata about our project, such as its name, version as well as the list of its dependencies.
* **karma.conf.js**: Karma is a test runner for JavaScript applications. This file contains some configuration for Karma. We rarely need to modify this file.
* **tsconfig.json**: includes setting for the TypeScript compiler. Again, we hardly, if ever, need to modify this file.
* **tslint.json**: includes the settings for TSLint which is a popular tool for linting TypeScript code.

**Main Building Blocks of Angular**

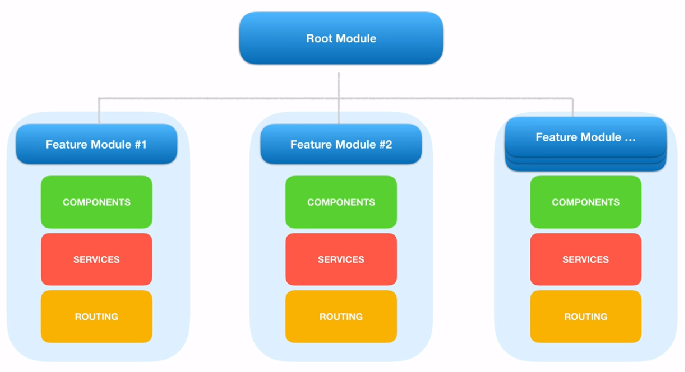
1. Modules
2. Components
3. Templates
4. Metadata
5. Data binding
6. Directives
7. Services
8. Dependency Injection.

**Modules**

* Every Angular app contains at least one Angular module, i.e. the **root module**.
* Generally, it is named **as AppModule**.
* We can create multiple Modules if needed.
* Any angular module is a class with **@NgModule** decorator.
* Encapsulation of different similar functionalities.



**Importing other modules in Root Module**



**Module Decorators:**

Decorators are basically used for attaching metadata to classes so that, it knows the configuration of those classes and how they should work.

NgModule is a decorator function that takes metadata object whose properties describe the module.

The properties are,

* **declarations**: The classes that are related to views and it belong to this module.
* **exports**: The classes that should be accessible to the components of other modules. (A root module generally doesn’t export it’s class because as root module is the one which imports other modules & components to use them.)
* **imports**: Modules whose classes are needed by the component of this module.
* **providers**: Services present in one of the modules which is to be used in the other modules or components. Once a service is included in the providers it becomes accessible in all parts of that application
* **bootstrap**: The *root component* which is the main view of the application. This root module only has this property and it indicates the component that is to be bootstrapped.

**Angular libraries**

* Angular gives us a collection of JavaScript modules (library modules) which provide various functionalities.
* Each Angular library has **@angular** prefix, like @angular/core, @angular/compiler, @angular/compiler-cli, @angular/http, @angular/router.
* You can install them using the **npm** package manager and import parts of them with JavaScript import statements.

|  |
| --- |
| import { Component } from '@angular/core'; |

* In the above example, Angular’s Component decorator is imported from the @angular/core library.

**Components**

* A component controls one or more section on the screen called a **view**.
* For example, if we build shopping cart Application, we can have components like App Component (the bootstrapped component), list products, product description, add to cart, update cart, etc.,
* Component fetch and update data from **services**. Transforms the DOM using **Directives** and Redirecting the user to another component by using **Routing**.
* Inside the component, you define a component’s presentation logic i.e. how does it support the view—inside a class.
* Every app has a main component which is bootstrapped inside the main module, i.e AppComponent.

|  |
| --- |
| import { Component } from '@angular/core';   @Component({ selector:'app-root', templateUrl:'./app.component.html', styleUrls: ['./app.component.css'] })   export class AppComponent{ title = 'app works!'; } |

## **Metadata:**

Metadata tells Angular how to process a class.

|  |
| --- |
| import { Component } from '@angular/core';   @Component({ selector:'app-root', templateUrl:'./app.component.html', styleUrls: ['./app.component.css'] })  ng generate component componentName |

* Here is the ***@Component*** decorator, which identifies the class immediately below it as a component class.
* The ***@Component*** decorator takes the required configuration object which Angular needs to create and present the component and its view.

The most important configurations of @Component decorator are,

* **selector**: Selector tells Angular to create and insert an instance of this component where it finds ***<product-desc>*** tag. For example, if an app’s HTML contains ***<product-desc></product-desc>***, then Angular inserts an instance of the Product Description view between those tags.
* **templateUrl**: It contains the path of this component’s HTML template.
* **providers**: An array of **dependency injection** providers for services that the component requires. This is one way to tell Angular that the component’s constructor requires a **ProductService** to get the list of products to display.

The template, metadata, and component together describe a view.

## **An Angular Component in Action**

* app.component.css
* app.component.html
* app.component.spec.ts
* app.component.ts
* app.module.ts
* **A CSS file**: where we define all the styles for that component. These styles will only be scoped to this component and will not leak to the outside.
* **An HTML file**: contains the markup to render in the DOM.
* **A spec file**: includes the unit tests.
* **A TypeScript file**: where we define the state (the data to display) and behavior (logic) of our component.

## **Create Component through CLI**

|  |
| --- |
| ng generate component componentName  ng g c componentName |

## **Install Bootstrap npm**

|  |
| --- |
| npm install bootstrap@3 --save |

After successful Installation, make necessary changes in angular.json.

***angular.json => architect => build => style[];***

**Use your selector as an Attribute:**

To use your selector as a Attribute, simply make changes in selector property as

|  |
| --- |
| selector : '[attributeName]' |

Now in your html just use it as Attribute for an element

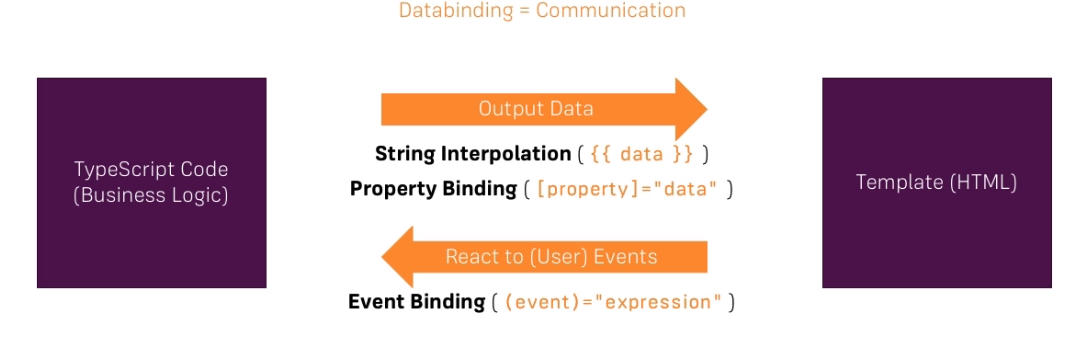
|  |
| --- |
| <div attribute></div> |

**Use your selector as a Class:**

|  |
| --- |
| selector : '.className' |

## **Data Binding**

Which means Projection of the Model or Communication.



The combination of both is called Two way data binding.

|  |
| --- |
| ( [(ngModel)]="data" ) |

**Class Binding:**

Syntax:

|  |
| --- |
| [class.className]="propertyValue"; |

If the propertyValue becomes true the specific class will be apply or else no.

**Style Binding**

Syntax:

|  |
| --- |
| [style.stylePropertyName]="propertyValue ? ‘value1’ : ‘value2’" |

**Event Filtering:**

Syntatx:

|  |
| --- |
| <input type="text" (keyup.enter)="OnKeyUp($event)"> |

If you want to invoke a function only for inputs means just use ***(input)*** or ***(ngModelChange)***

## **Directives**

Directives are Instructions to the DOM.

There are three kinds of directives in Angular:

1. **Components** - directives with a template.
2. **Structural directives** - change the DOM layout by adding and removing DOM elements. (leading with **\***).
3. **Attribute directives** - change the appearance or behavior of an element.

## **List of Directives in Angular**

1. \*ngIf

**Difference Between ngIf and hidden.**

**ngIf :** It just remove the element and re-attach the element based on the condition.

**Hidden :** It just hides and shows the attached element based on the condition.

1. \*ngFor

By default \*ngFor has the following local variables,

**index**: number: The index of the current item in the iterable.

**first**: boolean: True when the item is the first item in the iterable.

**last**: boolean: True when the item is the last item in the iterable.

**even**: boolean: True when the item has an even index in the iterable.

**odd**: boolean: True when the item has an odd index in the iterable.

Syntax:

|  |
| --- |
| <li \*ngFor="let user of users; index as i; first as isFirst"> |

1. ngClass
2. ngStyle
3. ngSwitch

## **Creating Custom Directive**

**Angular custom attribute**

It is created to change appearance and behavior of HTML element.

Find the steps to create custom attribute directive.

1. Create a class decorated with **@Directive()**.

2. Assign the attribute directive name using selector metadata of **@Directive()** decorator enclosed with bracket [] .

3. Use **ElementRef** class to access DOM to change host element appearance.

4. Use **@Input()** decorator to accept user input in our custom directive.

5. Use **@HostListener()** decorator to listen events in custom attribute directive.

6. Configure custom attribute directive class in application module in the declarations metadata of **@NgModule** decorator.

Angular allows us to create Custom directives.

Syntax:

|  |
| --- |
| ng g d directiveName |

**HostListener - decorator**

Listen to the event occurs on the element and act accordingly.

This is a function decorator that accepts an event name as an argument. When that event gets fired on the host element it calls the associated function.

So if we add this function to our directive class:

|  |
| --- |
| @HostListener('mouseover') onHover() {  console.log("hover"); } |

**Angular Custom Structural Directive:**

Structural directive is used to change the DOM layout by adding and removing DOM elements.

Find the steps to create custom structural directive.

1. Create a class decorated with @Directive().

2. Assign the structural directive name using selector metadata of @Directive() decorator enclosed with bracket [] .

3. Create a setter method decorated with @Input(). We need to take care that the method *name should be same as directive name*.

4. Configure custom structural directive class in application module in the declarations metadata of @NgModule decorator.

To create custom structural directive we need to use TemplateRef and ViewContainerRef that will help to change the DOM layout.

**TemplateRef** : It represents an embedded template that can be used to instantiate embedded views.

**ViewContainerRef**: It represents a container where one or more views can be attached.

**Pipes**

Angular Pipe takes in data as input and transforms it to a desired output before the view. Angular comes with some inbuilt pipes as follows,

* Lowercase
* Uppercase
* Titlecase
* Slice
* Json
* Number
* Percent
* Currency
* Date

Pre-defined format options

**'short'**: equivalent to 'M/d/yy, h:mm a' (6/15/15, 9:03 AM).

**'medium'**: equivalent to 'MMM d, y, h:mm:ss a' (Jun 15, 2015, 9:03:01 AM).

**'long'**: equivalent to 'MMMM d, y, h:mm:ss a z' (June 15, 2015 at 9:03:01 AM GMT+1).

**'full'**: equivalent to 'EEEE, MMMM d, y, h:mm:ss a zzzz' (Monday, June 15, 2015 at 9:03:01 AM GMT+01:00).

**'shortDate'**: equivalent to 'M/d/yy' (6/15/15).

**'mediumDate'**: equivalent to 'MMM d, y' (Jun 15, 2015).

**'longDate'**: equivalent to 'MMMM d, y' (June 15, 2015).

**'fullDate'**: equivalent to 'EEEE, MMMM d, y' (Monday, June 15, 2015).

**'shortTime'**: equivalent to 'h:mm a' (9:03 AM).

**'mediumTime'**: equivalent to 'h:mm:ss a' (9:03:01 AM).

**'longTime'**: equivalent to 'h:mm:ss a z' (9:03:01 AM GMT+1).

**'fullTime'**: equivalent to 'h:mm:ss a zzzz' (9:03:01 AM GMT+01:00).

**Custom Pipe**

* Every pipe is decorated with ***@Pipe*** where we define the name of our custom pipe. Every pipe will implement ***PipeTransform*** interface.
* This interface provides ***transform()*** method and we have to override it in our custom pipe class.
* ***transform()*** method will decide the input types, number of arguments and its types and output type of our custom pipe.

We perform the following steps to create a custom pipe.

Step 1: Create a pipe by using ***ng g p pipeName***

Step 2: Decorate the class using @Pipe.

Step 3: Implement PipeTransform interface.

Step 4: Override transform() method.

Step 5: Configure the class in application module with @NgModule.

Step 6: Ready to use our custom pipe anywhere in application.

On the basis of change detection, angular provides two types of pipes.

**Pure pipe**: This will run only for pure changes in component properties.

**Impure pipe**: This will run for any type of changes in component properties.

By default, All the pipes becomes pure pipe.

**Components Interaction**

Pass data from parent to child with input binding by using @Input decorator.

**@Input**

Defines input variable in component communication. It is used to communicate from parent to child component using **property binding**.

* As the name implies it is used to Inputting the data.
* Enables the component to accept Input from the parent component.
* Use the @Input decorator.

**@Output**

Defines output variable in component communication. It is used to communicate from child to parent component using **custom event binding**.

**EventEmitter:**

* ***EventEmitter*** is a class in angular framework. It has **emit**() method that emits custom events.
* We can use **EventEmitter** in custom event binding.
* To create a custom event we need to create an instance of ***EventEmitter*** annotated by ***@Output()***.
* We can receiving emitting custom function arguments by ***$event***.
* To achieve it first we need to import it in our component file as given below

|  |
| --- |
| import {Component, EventEmitter, Input, Output} from '@angular/core'; |

* And then initialize it using @Output decorator as follows,

|  |
| --- |
| @Output() customEventname = new EventEmitter(); |

* Using emit() method of EventEmitter class we can emits parent component event.

|  |
| --- |
| this.customEventname.emit(); |

**View Encapsulation**

View encapsulation defines whether the template and styles defined within the component can affect the whole application or vice versa. Angular provides three encapsulation strategies:

* **Emulated** (default) - styles from main CSS (style.css) propagate to the component. Styles defined in this component's @Component decorator are scoped to this component only.
* **Native** - styles from main CSS do not propagate to the component. Styles defined in this component's @Component decorator are scoped to this component only.
* **None** - styles from the component propagate back to the main HTML and therefore are visible to all components on the page.

**ng-content**

We want the header and footer content to be fixed, but we also want to allow a user to add dynamic content to the body section.

Transclusion is a way to let you define a fixed view template, and at the same time allow you to define a slot for dynamic content by using ***<ng-content>*** tag.

***<ng-content>*** accepts a select attribute, which allow us to sort of elements, attributes classes.

|  |
| --- |
| <!-- add the select attribute to ng-content -->  <**ng-content** select="[attributeName]"></**ng-content**> |

|  |
| --- |
| <!-- select matched class names only --> ... <**ng-content** select=".className"></**ng-content**> ... |

|  |
| --- |
| <!-- select matched elements names only --> ... <**ng-content** select="elementName"></**ng-content**> ... |

|  |
| --- |
| <!-- select matched multiple classes --> ... <**ng-content** select=".class1,.class2"></**ng-content**> ... |

**ng-container directive**

We can’t use more than one structural directive in a single element. In order to avoid having to create that extra div, we can instead use ng-container directive:

|  |
| --- |
| <**ng-container** \*ngIf="lessons">  <**div** class="lesson" \*ngFor="let lesson of lessons">  <**div** class="lesson-detail">  {{lesson | json}}  </**div**>  </**div**> </**ng-container**> |

**Angular Component Lifecycle Hooks**

A component in Angular has a life-cycle, a number of different phases it goes through from start to end.

We can hook into those different phases to get some works done on the particular phase.

**Constructor:**

This is invoked when Angular creates a component calling new on the class.

**ngOnChanges -** OnChanges**:**

Invoked event **ngOnChanges** every time there is a change in one of the input properties [inputProperty] of the component.

**ngOnInit -** OnInit**: Called once the component Initialized**

The ngOnInit method of a component is called directly after the constructor and after the ngOnChange is triggered for the first time. It is the perfect place for initialisation work. Invoke only once.

**ngDoCheck -** DoCheck**: Called during every change detection runs.**

This is fired each time anything that can trigger change detection has fired (e.g. click handlers, http requests, route changes, etc…). This lifecycle hook is mostly used for debug purposes;

**ngAfterContentInit -** AfterContentInit **: Called after content has been projected into view**

Invoked after Angular performs any content projection into the components view. Invoke only once.

**ngAfterContentChecked -** AfterContentChecked: called after every check of component (ngDoCheck) content.

**ngAfterViewInit -** AfterViewInit: **Called after component’s view (and its child view) has been initialized.**

Called after ngAfterContentInit when the component’s view has been initialised. Invoked when the component’s view has been fully initialized. Invoke only once.

**ngAfterViewChecked -** AfterViewChecked

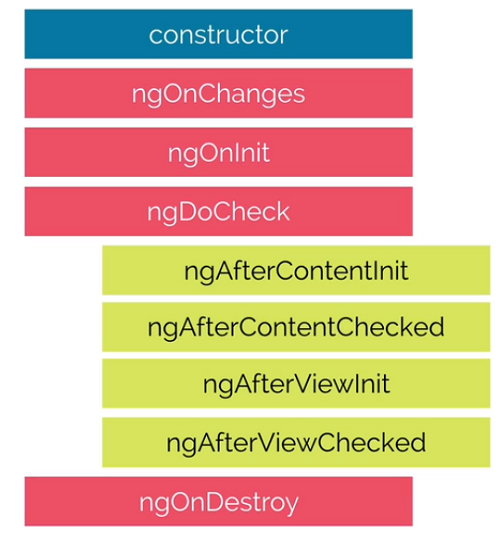
Invoked each time the view of the given component has been checked by the change detection mechanism of Angular. called after every check of component (ngAfterContentChecked) content.

***ngDoCheck => ngAfterContentChecked => ngAfterViewChecked***

**ngOnDestroy -** OnDestroy

This method will be invoked just before Angular destroys the component.

The hooks are executed in this order:



**Template Driven Forms**

* In Template driven Form Approach, everything is defined in the template.
* In template driven we use directives to create the model.
* The directives we need to build template driven forms are in the FormsModule.

**Form Setup:**

* Import FormsModule and add it to our NgModule as an import

|  |
| --- |
| import {FormsModule} from '@angular/forms'; |

* One of the directives pulled in via the FormsModule is called NgForm.
* So just by adding FormsModule to our NgModule imports our template form is already associated with an instance of the NgForm directive.
* This instance of ngForm is hidden but we can expose it with a local template reference variable attached to the form element like,

|  |
| --- |
| <form #f="ngForm"> ... </form> |

* Now we can use the variable **f** in our template and it will point to our instance of the ngForm directive.
* To create Form control, we need to do two things to each template.

1. Add the **ngModel** directive
2. Add the **name** attribute.

|  |
| --- |
| <input name="foo" ngModel> |

**Angular Form States**

|  |  |  |
| --- | --- | --- |
| **States** | **Data Type** | **Description** |
| valid | Boolean | Returns true if the form/input element has been valid |
| invalid | Boolean | Returns true if the form/input element has been Invalid |
| touched | Boolean | Returns true if the input element has been touched |
| untouched | Boolean | Returns true if the input element yet not touched |
| dirty | Boolean | Returns true if the input element or form has been modified |
| pristine | Boolean | Returns true if the input element or form yet not modified (original state) |
| submitted | Boolean | Returns true if the form has been submitted |
| errors | Object | Returns an object with existing errors in the input element or form. The key names or error names and the values always true. |

**Routing**

Routing Makes your application as SPA. To use Routing in our application, we have to follow the following steps,

1. **Import RouterModule and Routes**

|  |
| --- |
| import { RouterModule, Routes } from '@angular/router'; |

* RouterModule is a separate module in angular that provides required services and directives to use routing and navigation in angular application.
* Routes defines an array of roots that map a path to a component.

2. **Create Array of Routes**

|  |
| --- |
| const routes: Routes = [  { path: 'pathName', component: componentName },  { path: '', redirectTo: '/manage-book ', pathMatch: 'full' } ] |

* The path property describes the URL this route will handle.
* The component property is the name of the component we want to display when the URL in the browser matches this path.

3. **Using RouterModule.forRoot()**

|  |
| --- |
| imports: [ RouterModule.forRoot(routes) ] |

Now we need to import RouterModule.forRoot(routes) using imports metadata of @NgModule. Here argument routes is our constant that we have defined above as array of Routes.

4. **RouterLink and RouterLinkActive**

|  |
| --- |
| <**a** routerLink="/users" routerLinkActive="active-link">User list</**a**> |

RouterLink is a directive that is used to bind a route with clickable HTML element. RouterLinkActive is a directive that is used to add or remove CSS classes.

5. **RouterOutlet**

|  |
| --- |
| <router-outlet></router-outlet> |

RouterOutlet is a directive that is used as <router-outlet>. The role of <router-outlet> is to mark where the router displays a view.

**RouterLink as a property Binding**

By using RouterLink as a property Binding, we can easily configure the complex links.

<**a** [routerLink]="['/login', 'admin']">Sign Up</**a**>

**Navigating to other links programmatically**

To Navigate to other pages through programmatically, we need to follow the below steps,

1. Import Router class from ‘@angular/router’
2. Create dependency injection on the current class’s constructor function.

|  |
| --- |
| constructor(private router : Router) |

3. By using the router object we can navigate to next link like,

|  |
| --- |
| this.router.navigate([‘/home’]); |

**Passing Parameter to Routes**

Sometimes our url may have some parameters like this

|  |
| --- |
| product/2 |

The product number will be different for each products. So we can’t write route for each and every product. Here is the **Dynamic Routing**.

We need to handle Dynamic route for the above url.

|  |
| --- |
| {path: 'product/:pid', component: ProductComponent} |

Here the :pid will receive the url value dynamically.

**Fetching Route Params**

To fetch the route params in our class file we need to follow the below steps,

1. Import ActivatedRoute class from ‘@angular/router’

The ActivatedRoute class have many inbuilt methods which is related to currently activated route.

2. Create DI for imported ActivatedRoute class.

|  |
| --- |
| constructor(private route : ActivatedRoute) |

3. Now we can get the route params through route object.

|  |
| --- |
| this.route.snapshot.params[‘pid’] |

Or

|  |
| --- |
| this.route.params.subscribe((params)=> {  params['pid'] }); |

The **pid** should be same as which we used in dynamic routing **(:pid)**

**Unsubscribe the above subscribe**

Unsubscribe the subscription is very important, if not the subscription will be active and always in memory.

We need to unsubscribe the subscription while our current component gets destroyed.

So implement OnDestroy interface in the current class and the interface method is

ngOnDestroy.

**Http**

Angular 6 **HttpClient** that use for consuming RESTful API from the remote server.

To enable Http service in our Angular Application, we need to follow the below steps,

1. **Setup and Configure Angular 6 HttpClient**

Register the HttpClientModule in our root module.

|  |
| --- |
| import { HttpClientModule } from '@angular/common/http'; |

Add that module in `**@NgModule**` imports.

|  |
| --- |
| imports: [  BrowserModule,  HttpClientModule ], |

That it's, now you can use the Angular 6 HttpClient in your application.

**2. Create object for HttpClient through DI on the required classes or services.**

|  |
| --- |
| import { HttpClient } from '@angular/common/http';  constructor(private http: HttpClient) { } |