**Angular**:

* Angular is **component-based** model.
* Breaking down page into components, each component knows how to render itself and knows how to deal with user interaction for that component.
* Every Significant portion of a page which can be self sufficient knows what to do with that area can be split and created as component.
* Every component can be assigned a tag or a selector. This selector is what a consumer can use to call and render that component.
* Ex. ***header-section, main-section, side-bar, footer-section*** are components. We can register these components with a name. Registering means saying angular that we have created components and we have created a name. Now angular renders these components (presents) in the UI.
* Each components can have sub-components.

updates

nav

details

summary

main-section

footer-section

side-bar

Header-section

* A tree needs a starting point (**root**) component. Every Angular application has a root which is going to hold main components that needs to display in the page and those main components can have child components and so on.
* Angular project consists of bunch of things that makes a typical **Angular CLI** utility which is useful to kickstart development process to avoid all boiler plate coding by creating barebones of the project and making project ready for us. Useful without having to assemble all the files and start everything yourself.
* **npm install @angular/cli -g -------**->(@group/package) : scoped package name, -g : Install globally
* **ng version :** Shows version
* **ng** is the command to run angular cli
* **ng new project\_name** : To create a new project
* **ng serve:** Angular looks at code base, bundles all modules,creates a simple HTTP Server and hosts the angular project it have. We can access localhost url and load the html and we can make changes to the code and that url is automatically keep updating with latest changes.
* **ng build:** runs Angular CLI, which looks at project and generates HTML, CSS, Javascript into one folder and that is going to be independent of Angular CLI.

**Component:**

* To create a component, we need 3 files, **component\_name.css, component\_name.ts, component\_name.html, component\_name.specs.ts** file is a unit test file contains test cases.
* Every Angular component is mainly a **Typescript** class. As the html and css files as just extra “attachments” the main TypeScript file.
* The way to create a component using angular cli **ng generate component component\_name**. By default, it will be created in app folder. It will create all the 3 files and also updates **app.module.ts** file for the newly created component.
* The way to use a component is to use with the selector.
* Every component has the html tag / selector which is going to render / instantiate the component.
* To place the component in other html page, we can use the selector (like a html tag) of the respective component in that place.
* We can use one component in other components any number of times.
* Steps to create a component manually
  + Create a typescript class which contains functionality and make it export which should implement **OnInit** interface.
  + Register that class as an Angular component. Tell Angular that this is a component class by providing metadata to the class.
  + Each class contains two parts : Class, Registration.

@Component({

selector: 'app-date',

templateUrl: './date.component.html',

styleUrls: ['./date.component.css']

})

* + Angular looks the component.ts file, which is the starting point. Looks these files for rendering as well as styling.
* An annotation is a typescript way of adding meta-data to classes.
* Pass an object with properties (***selector, templateUrl, styleUrls***) to the Component annotation to tell Angular how we want this component to behave.
* Whenever we use the selector in the html, it instantiates the respective class.
* Static portion: html, Dynamic portion : TypeScript / JavaScript.
* The mechanism of having some value in class and showing in view is called as Data Binding.
* Data binding in Angular refers to binding the data in the component instance to the HTML template. Any changes to the data automatically get updated in the view.
* setInterval is a JavaScript API that lets you run a function at regular time intervals.
* {{ }} : **interpolation** : triggers angular to do interpolation which does evaluation of an expression or the member variables of the class.

**Directive:**

* A **Directive** which is used to manipulate DOM, is like an attribute we add to an element
* **\*ngFor=”let item of items”** is a **directive** used to iterate in angular containstwo parts**:** first partis the array that is going to loop over, second part is the loop element / loop item that is going to contain the value of each element as the loop goes through.

<p \*ngFor="let phoneNumber of user.phone">{{phoneNumber}}</p>

* **\*ngIf=”condition expression”** is a **directive** used to check for the condition to execute the following markup to be rendered or not.

<div \*ngIf="user.phone.length > 0 ">

<p>Phone:</p>

<p \*ngFor="let phoneNumber of user.phone">{{phoneNumber}}</p>

</div>

<div \*ngIf="user.phone.length">

<p>Phone:</p>

<p \*ngFor="let phoneNumber of user.phone">{{phoneNumber}}</p>

</div>

* To pass values from a view to a class is by using a custom defined variables in the selector tag of the view and by creating respective data members in the respective class. To tell angular to take that value that was passed to the usage of that component and populate this thing over here. This can be done by using **@Input()** annotation.
* **Input** is a class in **@angular/core**. By using this Input you telling angular that this needs to be pre-populated by angular.
* There may be bunch of properties which can be fed from the view to the class. To identify each we can use the name of the attribute in the **@Input**

@Input('name') userName : string;

@Input('title') title : string;

* Every class has a constructor. After creating the object, angular populates the values. So, we can not use constructor to populate values which we are passing values from view to the class. For this we need to add in **ngOnInit()** method (lifecycle hook)

ngOnInit() {

this.user = {

name : this.userName,

title :this.title,

address :'P. No. 63, Andhra Bank Colony, Asmangadh, Moosarambagh, Hyderabad - 36',

phone : ['9866489944','7893729944']

};

}

* Life cycle Hooks are called as life cycle events. **ngOnChanges(), ngOnInit(), ngDoCheck(), ngOnDestroy(), …**
* In a component, if any life cycle hooks are present, angular automatically executes code in those methods. If none are present, angular just ignores. The methods are available to us using Interfaces. If we don’t use the interface in the class also the method is going to execute. But, if we use the interface then the respective methods should be implemented.
* Instead of accessing each and every property from the view to the class, we can a model class and this can be accessed in the class using @Input

@Input('user') userObj : User;

ngOnInit() {

this.user = {

name : this.userObj.name,

title :this.userObj.designation,

address :this.userObj.address,

phone :this.userObj.phone,

};

}

* To pass user values of the model class from the view, we need to create the model class instance in app component’s constructor.
* To pass a value from the class we need a member variable.
* In order to tell angular to take the instance of the member variable in the view, we need to surround the attribute with **square brackets [ ]**. This tells angular to evaluate the instance and take the member variable that refers to.

<app-address-card [user]="user"></app-address-card>

* In the app component we use the following

export class AppComponent {

title = 'Rajesh';

user : User;

constructor() {

this.user = new User();

this.user.name = "Rajeswara Sarma";

this.user.designation = "Software Developer";

this.user.address = "DSNR";

this.user.phone = ['7893729944','9866489944'];

}

}

* If we do the above process, we need not to set model class values in ngOnInit() method. Because those properties can be directly used in the view.

@Input('user') user : User;

constructor() {

}

ngOnInit() {

}

**Styling:**

* The styles that we apply in our component all the css classes that we apply in your component.css file apply only to the markup inside the component inside. This is the way angular manages styling.
* The [property] selector is used in css to style elements that have that particular property.
* For global styling, we need to put all the css styles in **styles.css** file. Outside the app folder inside src folder.

**Events:**

* The way to handle click events is by telling angular to run a function when the button is clicked. We need to write a function to execute the code for the click event in the class. Syntax for click events is the **word** **click is surrounded by parenthesis**.

<button (click)="toggleCollapse()">Expland / Collapsed</button>

<div \*ngIf="!isCollapsed">

<p class="address">{{user.address}}</p>

<div \*ngIf="user.phone.length > 0 " class="phone">

<p>Phone:</p>

<p \*ngFor="let phoneNumber of user.phone">{{phoneNumber}}</p>

</div>

</div>

toggleCollapse() {

this.isCollapsed = !this.isCollapsed;

}

**To way data binding:**

* Data be a part of component, binded to the view in both ways. View updating the data in the component and the data in the component updating the view back.
* It comes when we are using form data, that is when we update the data in the view.

<input type="text" [value] = "inputText">

* Above is One-way data-binding. Because the value of the text box comes from the component which is evaluated in the view.
* To achieve two-way data-binding, we use a directive **ngModel**. To use this **ngModel**, we use **FormsModule** in **app.modules.ts** (Forms Module comes along with angular but it doesn’t get included in the angular cli project that’s get generated). The syntax to use ngModel in the view is **[(ngModel)]**.
* The **[( )]** syntax is referred to as banana-in-a-box. It helps to remember the order : brackets outside, parenthesis inside.
* **( )** -> view event causes something happen to the component. That is the data flow from the view to the component.
* **[ ]** -> which evaluates that is the data flow from the component to view (Property binding)

**Modules**:

* A module some is kind of container / namespace which contains different components.
* AppComponent is created by default which has been declared in the module.ts file. So, AppComponent is a part of AppModule.
* We can create multiple modules in a project and each module can contain components that is other modules (containers) can contain other components.
* When we create a module, then this module can be used by anybody.
* When we import a module, we can use the components in that module.

@NgModule({

declarations: [

AppComponent

],

imports: [

BrowserModule,

AppRoutingModule

],

providers: [],

bootstrap: [AppComponent]

})

export class AppModule { }

* To create a module we use **ng generate module module\_name**. It will create a module with the module name as the name of the folder inside the app folder which contains ts file in it.

@NgModule({

declarations: [],

imports: [

CommonModule

]

})

export class ViewModule { }

* To create a component inside other module, we use **module\_name/component name**.
* The **module\_name/** prefix to the name in the **ng generate** command actually tells the Angular CLI which folder to create the files in. And so, which module the component needs to be associated with.
* To use component of one module into other module, we do by **importing** the module and also export components of that module by using **exports** in other module.ts file.

import { ViewModule } from './view/view.module';

@NgModule({

declarations: [ViewComponent],

imports: [

CommonModule

],

exports:[

ViewComponent

]

})

export class ViewModule { }

* First the module that needs the module, it needs to **import** module the component is declared in and secondly whatever the module the component is declared in has to **export** the component. The component which we declared in one module is used in another module.
* What is being **imported is the module itself**. We are telling our module to import another module.
* What is being **exported is the component** in the module that contains the component we need to have an exports declaration which says that we are exporting this particular component.

**Types of imports:**

1. TypeScript import

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

import { ViewModule } from './view/view.module';

1. Angular import : Used for Inter module dependencies

imports: [

BrowserModule,

AppRoutingModule,

ViewModule

],

**Service**

* Not all reusable elements are views. We can create some reusable elements which contains functionality / method that needs to be used in multiple places not all three usable functionality come with the view attached.
* If we need to create those kind of business services which doesn’t really have a view we can create a **Service** in Angular.
* Services area also classes like components which contains methods (functionality) that we can reuse across multiple different components.
* To create a service, we use **ng generate service service\_name**. It creates a Service class as ts file with the name of the service in the app folder with test cases file.
* Service class has an **Injectable** Annotation, which tells Angular that this is a Service.

**@Component -> Angular Component**

**@NgModule -> Angular Module**

**@Injectable -> Angular Service**

* When we create a Component then apart from creating all three files, Angular adds an entry to the module in order to assign the component to one of the modules, then this component has to be a part of the module.
* Using services we can create utility methods rather to share business logic that are needed across multiple components.
* If we want to make the service to be a part of the app module then, we need to declare it in **app.module.ts** inside the **@NgModule** annotation and inside the **providers**.
* In **@NgModule**, **declarations** element contains all **Components**, **providers** contains all the **services** that are the part of the module and imports contain all the other modules that this module depends on.

@NgModule({

declarations: [

AppComponent

],

imports: [

BrowserModule,

AppRoutingModule,

ViewModule

],

providers: [

TestService

],

bootstrap: [AppComponent]

})

export class AppModule { }

* The way to access the service is by using something called Dependency Injection.

**Dependency Injection :**

* When we have a class that dependent on another class. You don’t have the class create that instance by itself. What we do instead is have the class declare its dependency and have the dependency injected.
* In Angular, we don’t have to create an instance of a service. Tell Angular you need an instance of a service and Angular hands over an instance of the service to us.
* This can be done by using an argument of respective service in the constructor()

export class AppComponent {

constructor(private svc : TestService) {

svc.printToConsole('Sarma');

}

}

* Angular look at constructor arguments of the component and see if any of these arguments are **Injectable**, they are services. If they are services, Angular is going to find an instance of that service. It is going to create if it doesn’t find any one and that is going to pass that into that constructor and that component is going to be created.
* To use a service, First we need to create a service. Second, use this service as a private constructor argument of the component wherever required.

**Service Injection Context:**

* If a service in the outer module, it can be available for all the components in the module. But it is also available to the inner module (which was imported in the outer module).
* Unlike components are restricted to the module, Services are not restricted to the module that they are declared in. In the case of providers, Angular creates a common area for creating services is called as **share service space**.
* The shared service space is referred as **Dependency Injection context**
* Each module can have many services and these services are shared in a common area.

export class ViewModule {

constructor(private svc: TestService) {

svc.printToConsole('Inside View Module')

}

}

* Any of a component that makes dependency Injection which declares its dependency on any other services can get the instance of that service.
* Services are created in a shared service space anyway, then we can declare then in any module. To understand this, we need to understand lazy loading.
* Angular has a feature of **lazy loading** which **effects service Injection context**. So, whenever we are providing services especially in a lazy loaded applications, we need to be careful where to provide it.

**Rest Api:**

* Typically it make sense to import the http client module (**HttpClientModule**) in the root.

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

* All the services and providers in the module are now the part of injection context where we use it wherever we need.
* In order to inject it we use it as a constructor argument.

export class AppComponent {

constructor(private svc : TestService, private http: HttpClient) {

svc.printToConsole('Rajeswara Sarma');

}

}

* In Angular, request is Observable not as a promise which is powerful than promises. We are not going to get an object as a response back.
* Whatever the code we are going to execute then we need to write that code in subscribe method. That is when we use subscribe to a request then request operation is done the subscribe will execute.
* To get a response from the observable, we use this response from the subscribe method.

ngOnInit() : void {

let obs = this.http.get('http://localhost:8091/user\_info/Sarma');

obs.subscribe((response) => console.log(response));

}

* Accessing properties of undefined object returns an error.
* In order to avoid this, we can use **?** before accessing the property. That is whenever we are accessing property on our object then there would be risk that this object could be null then to avoid this follow the object instance with **?**.

<p>Login Name : {{response?.login}}</p>

<p>Full Name : {{response?.full\_name}}</p>

* Instead we use **\*ngIf=”response”** to a div to avoid to use **?** to each and every view element.

<div \*ngIf="response">

<p>Login Name : {{response.login}}</p>

<p>Full Name : {{response.full\_name}}</p>

<p>Repo Count : {{response.public\_repos}}</p>

<p>Follower Count : {{response.followers}}</p>

</div>

**Building a Project:**

* Use **ng build --prod** on the project to build a project in production mode. If we wont use in production mode, there will be warnings
* It will create dist folder in the project.
* Download http-server globally using : **npm install http-server -g**
* After installing http-server, we have to run it by specifying directory we wish to host. **http-server dist/second-project**.
* Angular is also going to do **AOT** (Ahead of Time Compilation).
* For every build there Angular will create minified files with new hash value in the file names except index.html.

**Routing:**

* In a single page application, when the user clicks on something a new view needs to be displayed with a full page refresh. It’s the same page but the view that’s displayed to the user dynamically changes by the execution of javascript.
* Javascript somehow removes the DOM elements of view1 and inserts the DOM elements of view2.
* This switching between views in Angular application happens with a concept called **Routing**.
* Routes in a single page app are not purley managed by flags or switches in a component. Routing is used in Angular by 2 ways.
* **URL based Routing**: Routing is controlled by Urls that is navigation is controlled by URLs. We need to be able to expose Urls in Angular application that takes the users directly to those views.
* **Component based Routing**: Each view is managed by its own component That is each view component knows to render that view. That component may render the whole view itself or it may include other components which may be root node of that component tree starts with that component. In this way we can have each view can have a component tree and each component working together with view knows how to render that view and each such view has a root component.
* In Angular first define what routes are there in the application to provide the functionality we are looking for. For each of those routes we configure the route url that triggers the route and we map it to the route component that’s responsible for handling the view in that route and then provide this configuration to the angular routing framework.
* Then Angular take care of the rest. Once the user gets a path that’s map to a route, it will remove the view that’s currently being displayed at the time and then it will find an instantiate the new component that’s mapped to the view that needs to show in the right view is shown to the user.
* As the user goes from view to view the right component route component is pulled up and then the component tree is rendered for that view.

**Steps for Routing:**

* Define route URLs
* Create Angular components for each view (one for each route). Each view is rendered by a single component or tree of components.
* Configure Angular to map route URLs to components.

**Routing Project:**

* If you enable routing, the Angular CLI Project will have those libraries included.
* **ng new project\_name --route**: To create a routing project.
* It will create a project with **AppRoutingModule** has been imported in **app.module.ts** and also creates **app-routing.module.ts** in the project.

import { NgModule } from '@angular/core';

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

const routes: Routes = [];

@NgModule({

imports: [RouterModule.forRoot(routes)],

exports: [RouterModule]

})

export class AppRoutingModule { }

* **const routes: Routes = []**; or **const routes: Route[] = [];** is an array of Route which contains bunch of mappings and we pass that configuration to Angular by calling **RouterModule.forRoot(routes)**.
* Telling Angular to configure those routes and whenever those routes are accessed in your page Angular is load those components.

const routes: Route[] = [

{path: 'home', component: HomeComponent},

{path: 'settings', component: SettingsComponent},

];

* Routing Array is an array of Objects. Each object in the Routing Array specifies a mapping URL to a component.
* In order to provide a window to put the component in the view, we need to have router-outlet in the app.component.html

<router-outlet>

</router-outlet>

* If we access the url specified in the routes then that respective view will be injected into this **routing-outlet**.
* Urls: **http://localhost:4200/home, http://localhost:4200/settings**
* If we don’t specify any route, then after hosting the application, no page will be displayed. So, to make a default route we can a route that specify the empty path to a component.

const routes: Route[] = [

{path: '', component : HomeComponent},

{path: 'home', component: HomeComponent},

{path: 'settings', component: SettingsComponent},

];

* From the above, we can infer that the **path to component is not one-to-one** and for multiple URLs (paths) map to a same component. It is also accomplished that specifying a component if there is no path route configuration
* But to make a path to redirect to another route we can use **redirectTo : ‘/home’** which contains **/** in the path. But in case of path, we don’t have the **/** because that will be automatically appended to the end.

const routes: Route[] = [

{path: '', redirectTo:'/home', pathMatch: 'full'},

{path: 'home', component: HomeComponent, pathMatch: 'prefix'},

{path: 'settings', component: SettingsComponent},

];

* **pathMatch : ‘full’** says the matching path should be completely matched.
* **pathMatch : ‘prefix’** matches the portion of the URL and the rest can be propagated.
* If we want to load a component to be load by Angular when the uses access the URL which is not matched with any of the paths mentioned in the route configuration, we need to configure wild cards \*\* : which matches anything. It retains the same URL in the address bar after routing to PageNotFoundComponent also, which is not an error

{path: '\*\*', component: PageNotFoundComponent},