Angular Intermediate Training

Slide 1

Hi Everyone,
Welcome to Angular Training.

This is Abhijit Simha R, I am Front End Developer with 5+ years of experience predominantly in Angular.

This is an Intermediate Level Angular Training Session. I hope everyone attending this session has attended my previous session "Angular Beginner Training" last month which covered all the basics of Angular.

In this session we will go through intermediate & a bit of advanced level concepts of Angular with hands-on exercises and also build an Angular Application towards the end.

Let's get started.

Slide 2

I hope everyone has Node setup on their machines and installed Angular.

If Angular is not yet installed please do install now, here are the cmd's to install Angular.

We can check the node version by giving the cmd
node –version

We need to point our npm registry to OPTUM's npm registry to install any npm package, so please give this cmd
npm config set registry https://repo1.uhc.com/artifactory/api/npm/npm-virtual/ --global

Once it is done give this cmd
"npm install -g @angular/cli"

Slide 3

Now, once Angular is installed create a folder on your machine where you want to build the Angular App and open cmd prompt in that location.

Let's create an Angular project by giving the cmd

ng new my-app

No for strict type checking as we are going to explore & learn now.
Yes for routing
Select SCSS for styling

Slide 4

Before we start, let's have recap of Angular Basics.

On a high level this is how Angular's Architecture looks like, Angular application is loaded into the browser through which the API calls are made to render the data from backend

Slide 5

Angular application is a SPA and all the views are displayed in one single page i.e, index.html.

The applications that are contained within one web page are called Single Page Applications (SPA)

This is because a single page application executes the logic in the browser, instead of a server. It does so with JavaScript frameworks that can lift this heavy data on the client-side. JavaScript also enables an SPA to reload only those parts of the app that a user requests for, not the entire app. As a result, SPAs are known to deliver fast and efficient performance.

The pages in SPA are segregated into views and these views take turns and appear on that one page which is index.html and the process of determining the view is managed by Routing. We will look into Routing in a short while.

On the contrary, a multi-page application (MPA) is an app that has more than one page. It works in a traditional way, requiring the app to reload entirely every time a user interacts with it or moves from one view to another view. Here each view is a page.

SPA's are way Faster and Responsive when compared to MPA's.

Slide 6

Every application will have atleast one module & one component which are root module & root component.

<u>Components</u> :

Each component has a template for view & a class for the code associated with the view and the Metadata that identifies an Angular component. We can see the metadata in @component decorator which has 'selector', it is the component tag to be used in other component's html to render this component, 'templateUrl' it contains the reference address of the .html file this is a linked template. We can also define inline template, (Note: use backticks for multiple lines). Next we have 'styleUrls', it contains the reference address of the .scss file (styling file css or scss based on our selection).

A decorator is function that adds metadata to a class, its members or its method arguments.

Class is construct allows us to create a type with properties that define data elements & methods that provide functionality.

All the components are pulled together into an application by defining Angular modules. Modules organize the application.

Every Angular application has atleast one module called root module (app.module.ts) and this root module will have one root component (appComponent) which will be bootstrapped. It is in this root component where the whole Angular application runs.

Root module can have any number of Feature modules, Shared modules.

<u>Modules</u> :

Modules organize the application into cohesive blocks of functionality & organize, modularize our application and promote code reusability. Because a component can exported from one module and used in another module.

Modules provide the environment for resolving pipes & directives into our component's templates. They also selectively aggregate classes from other modules & re-export them to use in convenient module.

Module also has metadata that is defined in @NgModule decorator.

In @NgModule decorator we have 5 arrays,

declarations : Every component, directive, pipe that we create is declared by a module in declaration array which belongs to that module. The declared components, directives, pipes are private by default. We can make accessible by exporting them.

Never re-declare components, pipes, directives that belong to other modules, they should imported in imports arrary.

Imports: This is where other modules can be extended by importing them here. It allows us to import modules which export their components, directives, pipes. Importing a module does give access to all the imports of that module, it only gives access its exports.

Exports: This array allows us to share module's components, directives, pipes with other modules. We can also re-export imported Third part modules & Core modules & any module.

Providers: Angular registers service providers for our application in the providers array of the decorator at the module level or component level (providers can also be in defined in component decorator). Any service i.e, added to this array will be registered at root of the application.

Bootstrap: This array defines the starting point of the application. Every application should bootstrap atleast one component which is the root component. Bootstrap array should only be used in root application module (app module) and should not be used in any other module.

<u>Services</u> :

Service is class with a focused purpose and implements functionality i.e., independent from any component. Services are a great way to share information among classes that don't know each other and provide us the ability to share data or logic across the components.

Services encapsulate external interactions such as data access (API calls).

That is all for the recap of the basic guys.

Let's get a bit deeper into the concepts now.

Slide 7

Angular follows a Singleton design pattern when implementing a service which means there will be only one instance of the service that exists in an application and this instance will be invoked when the service is injected into components through Dependency Injection.

Dependency Injection is coding pattern in which a class receives the instances of Objects it needs (called dependencies) from an external source rather than them itself.

In Angular the external source for providing these Objects (dependencies) is the **Injector** and all the services are & should be registered with Angular Injector. Registering a Service with Angular Injector can be done with the help of a provider. Essentially what a provider does is it returns a service.

Now, for a provider to provide the services registered with it, the provider also in turn has to be registered. Providers can be registered by defining it in the "provideIn" property of @Injectable() decorator's of the service or as part of the component decorator's in the "providers" array property or as part of the module's "providers" array property.

Now that the services are provided, they can be injected into components constructor (this is called as dependency injection) and utilized. Essentially what is happening is that whenever a service is injected into an component's constructor it searches for that inject service class in the providers array of component, if found it will retrieve the service class instance and provide it to the component if not found it will look up further and searches it in the component's parent component and if no found there it will keep on searching till topmost parent arrives and if it does not find it there, then it looks up in the providers array of the component's Module and if does not find there it go the parent module and it goes up till the root module which is the parent module of the whole application and fetches the service instance. If it is not found even here, it will throw a "NullInjectorError".

Now, that's how a service works.

Slide 8

These services can be scoped according to our need.

Root Level: If the provider is registered in the root module i.e., either by providing in “AppModule ’s” providers array or by specifying ‘root’ “provideIn” of Service property of @Injectable decorator (which will be the case mostly these days) then the service will be registered with Angular Injector at the app root level i.e, application level (global level).

Module Level: If the provider is registered in any particular module’s providers array or by specifying Module name in “provideIn” of Service property of @Injectable decorator then the service will be available at that particular module level and will not be available at root level or for any other module.

Component Level: If the provider is registered in a component’s providers array then the service will be available at component level. If the service is already registered with root module, then the Injector will provide the serive with a unique instance for the component.

Limited Provider Scope with Lazy Loaded Modules: If a service is registered globally and a lazy loaded module is using that service then the Injector will provide the singleton instance for all the eagerly loaded modules, however it will provide a unique service instance for the lazy loaded module.

Now, let look into Angular Modularity

Slide 9

Angular Modularity

Modules are a great way to organize an application and extend it with capabilities from external libraries.

JavaScript enables us to use this Modular Design pattern from ES6.

A JavaScript module is an individual file with JavaScript code, usually containing a class or a library of functions for a specific purpose within your application. JavaScript modules let you spread your work across multiple files.

JavaScript modules and NgModules can help you modularize your code, but they are very different. Angular applications rely on both kinds of modules.

The Angular framework itself is loaded as a set of JavaScript modules.

An NgModule is a class marked by the @NgModule decorator. @NgModule takes a metadata object that describes how to compile a component's template and how to create an injector at runtime. It identifies the module's own components, directives, and pipes, making some of them public, through the exports property, so that external components can use them. @NgModule can also add service providers to the application dependency injectors.

I guess all the Module explanation that I gave earlier is making more sense now.

Angular libraries are NgModules, such as FormsModule, HttpClientModule, and RouterModule. Many third-party libraries are available as NgModules such as Material Design, Ionic, and AngularFire2.

Like I mentioned earlier, An Angular application needs at least one module that serves as the root module.
Now, let’s look at some of the frequently used Angular modules.

BrowserModule	@angular/platform-browser	When you want to run your application in a browser
CommonModule	@angular/common	When you want to use NgIf, NgFor
FormsModule	@angular/forms	When you want to build template driven forms (includes NgModel)
ReactiveFormsModule	@angular/forms	When you want to build reactive forms
RouterModule	@angular/router	When you want to use RouterLink, .forRoot(), and .forChild()
HttpClientModule	@angular/common/http	When you want to talk to a server

As we can see Angular distributes all of it's features into various modules and provides us with the ability to import only the modules which we need. Thus it significantly reduces the overall bundle size makes our application a bit mre faster as we are importing the modules only that are needed with the control of when & where.

Slide 10

Now as we develop Angular applications, we can & should segregate our code into appropriate module like feature, shared as per our requirement. Thus following the Modular architecture which is an organizational best practice.

Let's look at some of the types of modules we can segregate our modules into.

- Root Module – App Module which bootstraps your application.
- Core Module – It is the module that contains the base components, services that needed at initial load of the application. Eg: Login Page, Homepage, App Constants, API Constants, Routing Constants (Route Paths).
- Feature Modules – It is organized around a feature, business usecase, widget or user experience.
- Routing Modules – It provides the routing configuration for another Module.
- Shared Modules – It encapsulates a set of components, directives, and pipes available to other modules. Any component, directive, pipe and even built in modules that are being imported and used in two or more components of different modules should be moved to shared modules. We should always keep keen check on the size of the shared module as it will be imported across modules as it may increase the size of the modules it is being imported in. We should try further segregate these Shared into Feature specific shared modules.
- Service Module – It should contain all he utility services such as data access (API calls), data store, component communication, messaging (toaster).

Now, These Modules can be loaded eagerly when the application starts or lazy loaded asynchronously by the router.

Slide 11

Let's see what Routing is all about in Angular.

Like I said earlier, Angular is SPA and all the view are displayed in one page i.e, index.html. Now these views take turns while appearing on index.html which is managed by "**Routing**".

Routing allows us to navigate through all the views by configuring the route for each component that will display a view.
In simple words Routing allows you to move from one part of the application to another part or one View to another View.

The route that is configured is tied to an action or option with routerlink upon which associated route is activated which is the component's view & the browser's url changes then Angular router looks for a route definition matching the "path" segment in the router configuration which is connected with a component that needs to be loaded and once the match is done the component's view is displayed where the router-outlet is specified.

Navigating through Angular Application Routes

- Menu option, link, image, buttons chained with click events activates a route.
- Typing the URL in URL address bar.
- Browser's forward, backward buttons.

Router Module

Router is a separate module in Angular. It is in its own library package, @angular/router. The Router Module provides the necessary service providers and directives for navigating through application views. Router should be registered with its service provider RouterModule in a module's module.ts file.

Router allows us to pass optional parameters to the View, protect the routes from unauthorized users using Guards called Router Guards and allows you to dynamically load the view.

Router Module registers router's service, declares router directives, exposes configured routes

Router
It enables navigation from one component to the next component. We can access the router object and use its methods like navigate() or navigateByUrl(), to navigate to a route

Route
Route tells the Angular Router which view to display when a user clicks a link or pastes a URL into the browser address bar. Every Route consists of a path and a component it is mapped to. The Router object parses and builds the final URL using the Route.

Routes
Routes is an array of Route objects our application supports

RouterOutlet
The routerOutlet is a directive (<router-outlet>) that serves as a placeholder, where the Router should display the view

RouterLink
The RouterLink is a directive that binds the HTML element to a Route. Clicking on the HTML element, which is bound to a RouterLink, will result in navigation to the Route. The RouterLink may contain parameters to be passed to the route's component.

Slide 12

Configuration
We need configure the route in order to use them. This can done by setting the <base href="/"> in index.html which tells the router how to compose the navigation url's. Then we define routes for the view and register the Router Service with Routes, Map HTML Element actions to Route & finally Choose where you want to display the view by placing the routeroutlet tag appropriately in a component where we want to display our views.

Router uses first match win strategy when matching the routes. Hence we should always place more specific routes before the less specific routes.

Let's look at an example Route declaration.

Next we have to connect the route actions to routes, this can be done by rouerlink or calling navigate/navigateByURL on events. Then we need to place router-outlet in host component template where the routes need to be displayed. This will be the base component or root component in most of the cases.

There should be only one active router service in the entire Angular Application.

These routes can have children and can be nested. For child routes we need to use RouterMoudle,forChiled in the route configuration. This will be mostly used in feature modules.

These routed modules can be lazy loaded.

Slide 13

Being a Single Page Application, the Angular applications should not send the URL to the server and should not reload the page, every time user requests for a new page. The URLs are strictly local in Angular Apps. The Angular router navigates to the new component and renders its template and updates the history and URL for the view. All this happens locally in the browser.

There are two ways, by which Angular achieves this. These are called Location Strategies.

HashLocationStrategy
Where URL looks like http://localhost:4200/#/product

The Hash style routing using the anchor tags technique to achieve client-side routing.The anchor tags, when used along with the # allows us to jump to a place, within the web page.

PathLocationStrategy
Where URL looks like http://localhost:4200/product

The introduction of HTML5, now allows browsers to programmatically alter the browser's history through the history object. Using history.pushState() method, we can now programmatically add the browser history entries and change the location without triggering a server page request.

The PathLocationStrategy is the default strategy in Angular application.

To Configure the strategy, we need to add <base href> in the <head> section of root page (index.html) of our application.

Everyone mostly uses PathLocationStrategy.

Slide 14

Let's look into forms now.

What are forms?
Forms are a set inputs that are used to collect the data from the user. Forms can be very simple to very complex. It can contain large no. of input fields, spanning multiple tabs like in any company's job application form or just one or two fields like in a login page. Furthermore, Forms may also contain complex validation logic interdependent on multiple fields.

What a from essentially does is

- Initialize the forms fields and present it to the user
- Capture the data from the user
- Track changes made to the fields
- Validate the inputs
- Display helpful errors to the user

Angular's FormsModule provides all the above services out of the box. It binds the form field to the Angular component class. It tracks changes made to the form fields in order to respond accordingly. Angular forms provide the built-in validators to validate the inputs. You can create your own custom validator. It presents the validation errors to the user. Finally, it encapsulates all the input fields into an object structure when the user submits the form.

Angular supports two design approaches for interactive forms.
Template Driven Forms and Reactive Forms

Template Driven Forms
HTML & Data Binding – Form template in HTML which is bind in component code.

Template-driven approach is the easiest way to build the Angular forms. Here, the logic of the form is placed in the template i.e, (html). It is similar to AngularJS forms implementation. It will have minimal component code as all the logic is placed in the template, thus heavy template code which native HTML validation for Form Validation & two-way binding is used to connect the template to component class. In Simple word, Template-driven forms allows us to create easily without writing any complex javascript code.

Reactive Forms
Model Driven – Form Model & validation in component code.

Reactive Forms are also called as Model Driven Forms. In this approach the logic of the form is defined in the component(.ts) as an object ie., the representation of the form is created in the component class. This form model is then bound to the HTML elements using the special markups provided by ReactiveForms Module. It is more flexivl & can handle complex scenarios. It is easier to respond & perform an on value change. It is easy to add From input element dynamically. It makes unit testing easier and lastly in this approach we will have less HTML code and more TS code as the form model object is created in component class and it does not rely on HTML validation, it has it's own validation methods provided by ReactiveFormsModule.

Slide 15

Both reactive and template-driven forms track value changes between the form input elements that users interact with and the form data in your component model. The two approaches share the same underlying building blocks called Form Building Blocks, but differ in how you create and manage the common form-control instances.

Angular has three Form Building blocks. Let look into them.

First, we have
FormControl – It tracks the value and validation status of an individual form control.

Next, we have
FormGroup – It tracks the values and status for a collection or group of form controls. Any group of input elements can form FormGroup. A form is a FormGroup.

Next, we have
FormArray – It tracks the values and status for an array of form controls.

These building blocks are actually classes that are provided by FormsModule, ReactiveFormModule of Anuglar. Instances of these classes define "FormModel" which is the data structure that represents the HTML from that retains the form state, value, controls and also tracks the formContorls, formgroups.

FormModel is same both reactive and template-driven forms, but the way it is created is different.

In template-driven forms FormModel is creaed based on the form in the html template, where as in reactive forms formModel is created by us.

Now let's build a Template driven Form

First import FormModule in module.ts,
Create a normal html form.
angular automatically converts it into a Template-driven form. This is done by the ngForm directive. ngForm directive automatically detects the <form> tag and automatically binds to it. You do not have to do anything on your part to invoke and bind the ngForm directive.
The ngForm does the following

- Binds itself to the <Form> directive
- Creates a top-level FormGroup instance
- CreatesFormControl instance for each of child control, which has ngModel directive.
- CreatesFormGroup instance for each of the NgModelGroup directive.

We can export the ngForm instance into a local template variable using ngForm as the key (ex: #contactForm="ngForm"). This allows us to access the many properties and methods of ngForm using the template variable contactForm

The FormControl is the basic building block of the Angular Forms. It represents a single input field in an Angular form. Add the ngModel directive to each control.
<input type="text" name="firstname" ngModel>

We use the ngSubmit event, to submit the form data to the component class
<form #contactForm="ngForm" (ngSubmit)="onSubmit(contactForm)">

Now, We need to receive the data in component class from our form. we need to create the onSubmit method in our component class. The submit method receives the reference to the ngForm directive, which we named is as contactForm. The contactForm exposes the value method which returns the form fields as a Json object.

Validation

The Built-in validators use the HTML5 validation attributes like required, minlength, maxlength & pattern. Angular interprets these validation attributes and add the validator functions to FormControl instance.

First, we need to disable browser validator interfering with the Angular validator. To do that we need to add novalidate attribute on <form> element as shown below. it specifies that the form-data (input) should not be validated when submitted.

Required Validation

The required validator returns true only if the form control has non-empty value entered. Let us add this validator to all fields

Minlength Validation

This Validator requires the control value must not have less number of characters than the value specified in the validator.

For Example, minlength validator ensures that the firstname value has at least 10 characters.

Reactive Forms

Build the same from using reactive forms.

HTTP Requests using HTTPClient

Most front-end applications need to communicate with a server over the HTTP protocol, to download or upload data and access other back-end services. Angular provides a client HTTP API for Angular applications,

HttpClient Module allows us to query the Remote API source to get data into our Application.
It is a separate model in Angular and is available under the @angular/common/http package. The following steps show you how to use the HttpClient in an Angular app.

We need to import it into our root module module.ts
inject the HttpClient service as a dependency of an application class,
HTTPClient makes use of observable. Observable help us to manage async data. You can think of Observables as an array of items, which arrive asynchronously over time.

Route Guards

guards help us to secure the route or to perform some actions before navigating into a route or leaving the route. to restrict the user until the user performs specific actions like login. Angular provides the Route Guards for this purpose.

The Angular supports several guards like CanActivate, CanDeactivate, Resolve, CanLoad, and CanActivateChild.

CanActivate

This guard decides if a route can be activated (or component gets used). This guard is useful in the circumstance where the user is not authorized to navigate to the target component. Or the user might not be logged into the system

CanDeactivate

This Guard decides if the user can leave the component (navigate away from the current route). This route is useful in where the user might have some pending changes, which was not saved. The CanDeactivate route allows us to ask user confirmation before leaving the component. You might ask the user if it’s OK to discard pending changes rather than save them.

Resolve

This guard delays the activation of the route until some tasks are complete. You can use the guard to pre-fetch the data from the backend API, before activating the route

CanLoad

The CanLoad Guard prevents the loading of the Lazy Loaded Module. We generally use this guard when we do not want to unauthorized user to be able to even see the source code of the module.
This guard works similar to CanActivate guard with one difference. The CanActivate guard prevents a particular route being accessed. The CanLoad prevents entire lazy loaded module from being downloaded, Hence protecting all the routes within that module.

CanActivateChild

This guard determines whether a child route can be activated. This guard is very similar to CanActivateGuard. We apply this guard to the parent route. The Angular invokes this guard whenever the user tries to navigate to any of its child route. This allows us to check some condition and decide whether to proceed with the navigation or cancel it.

One of the common scenario, where we use Route guards is authentication. We want our App to stop the unauthorized user from accessing the protected route. We achieve this by using the CanActivate guard,