Angular

SummitWorks™
GLOBAL SOLUTION ARCHITECTS

Authorized & published by Summitworks Technologies Inc

Agenda

- Pipes
  - Introduction of Pipes
  - Built-in pipes
  - Custom pipes
- Angular router
  - Router Module
  - Component of RouterModule
  - Configuring Routes
  - Passing data to routes
  - Child routes
  - Guarding routes
  - Developing a SPA with multiple router outlets
  - Splitting an app into modules
  - Lazy-loading modules
- Working with forms
  - Overview of HTML forms
  - Template-driven forms
  - Reactive Forms

Pipes

- A pipe is a template element that allows you to transform a value into a desired output. A pipe is specified by adding the vertical bar (|) and the pipe name right after the value to be transformed.
- **Syntax of Pipe**

```
Expression | pipeOperator[:pipeArguments]
```

  - **Expression** : is the expression, which you want to transform
  - **|** : is the Pipe Character
  - **pipeOperator** : name of the Pipe
  - **pipeArguments**: arguments to the Pipe

Built-in Pipes

- There are some built-in pipes provided by Angular
- **DatePipe**
  - The Date pipe formats the date according to locale rules. The syntax of the date pipe is as shown below

```
date_expression | date[:format]
```

```
{{toDate | date:'medium'}}
```

  - **date_expression** is a date object or a number
  - **date** is the name of the pipe
  - **format** is the date and time format string which indicates the format in which date/time components are displayed.

Built-in Pipes Cont..

- **UpperCasePipe & LowerCasePipe**
  - As the name suggests, these pipes transform the string to Uppercase or lowercase.

```
`<p>Unformatted :{{msg}} </p>
<p>Uppercase :{{msg | uppercase}} </p>
<p>Lowercase :{{msg | lowercase}} </p>`
```

- **DecimalPipe / NumberPipe**
  - The Decimal Pipe is used to Format a number as Text. This pipe will format the number according to locale rules.

```
number_expression | number[:digitInfo]
```

```
`<p> Unformatted :{{num}}</p>
<p> Formatted :{{num | number}}</p>
<p> Formatted :{{num | number:'3.1-2'}}</p>
<p> Formatted :{{num | number:'7.1-5'}} </p>`
```

Custom Pipes

- It’s easy to create custom pipes to use in your templates to modify interpolated values
- It Import Pipe and PipeTransform and Use @Pipe decorator.

```
import {Pipe, PipeTransform} from '@angular/core';
```

- In your customPipe class implement PipeTransform interface.
- The PipeTransform interface has only one method known as the ***transform***. This method takes the value being piped as the first argument and number of optional arguments of any type. It returns the final transformed data(type any).

Custom Pipe Cont..

- We decorate the class with a @pipe decorator. **@pipe decorator** is what tells Angular that the class is a Pipe.

```
@pipe({
    name: 'tempConverter'
})
```

- Before using our Angular custom pipe, we need to tell our component, where to find it. This done by first by importing it and then including it in declarations array of the AppModule.

```
import {TempConverterPipe} from './temp-convertor.pipe';

@NgModule({
    declarations: [AppComponent,TempConverterPipe],
    imports: [BrowserModule,FormsModule,HttpModule],
    bootstrap: [AppComponent]
})
export class AppModule { }
```

Custom Pipe Example

```
export class TempConverterPipe implements PipeTransform {
    transform(value: number, unit: string) {
        if(value && !isNaN(value)) {
            if (unit === 'C') {
                var temperature = (value - 32) /1.8 ;
                return temperature.toFixed(2);
            } else if (unit === 'F'){
                var temperature = (value * 1.8 ) + 32
                return temperature.toFixed(2);
            }
        }
        return;
    }
}
```

```
<div class="row">
    <h3>Fahrenheit to Celsius </h3>
    <p> Fahrenheit : <input type="text" [(ngModel)]="Fahrenheit"/>
            Celsius : {{Fahrenheit | tempConverter:'C'}} </p>
</div>
```

Angular Routing

- The Angular Router enables navigation from one view to the next as users perform application tasks.
- We can build the single page application(SPA) without ever changing the URL.
- Routing allows you to:
  - Maintain the state of the application
  - Implement modular applications
  - Dynamically load the view
  - Implement the application based on the roles (certain roles have access to certain URLs)
  - Pass optional parameters to the View

The Angular Router Module

- The 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.
- **Components of Router Module:**
  - **Router :**The Angular Router is an object that enables navigation from one component to the next component as users perform application tasks like clicking on menus links, buttons or clicking on back/forward button on the browser. We can access the router object and use its methods like ***navigate()*** or ***navigateByUrl()***, to navigate to a route

Components of Router Module

- **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.

- **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.

Components of Router Module Cont..

- **Routes:** Routes is an array of Route objects our application supports.
- **ActivatedRoute:** The ActivatedRoute is an object that represents the currently activated route associated with the loaded Component.
- **RouterState:** The current state of the router including a tree of the currently activated routes with convenience methods for traversing the route tree.
- **RouteLink Parameters array:** The Parameters or arguments to the Route. It is an array which you can bind to RouterLink directive or pass it as an argument to the Router.navigate method.

Configuring Routes

- **Base URL Tag**
  - Most routing applications should add a <base> element to the index.html as the first child in the <head> tag to tell the router how to compose navigation URLs.

```
<base href="/">
```

- **Route Definition Object**
  - The Routes type is an array of routes that defines the routing for the application. This is where we can set up the expected paths, the components.
  - Each route can have different attributes; some of the common attributes are:
    - **path** - URL to be shown in the browser when application is on the specific routes
    - **component** - component to be rendered when the application is on the specific route
    - **redirectTo** - redirect route if needed; each route can have either component or redirect attribute defined in the route (covered later in this chapter)
    - **pathMatch** -optional property that defaults to 'prefix'; determines whether to match full URLs or just the beginning.
    - **children** - array of route definitions objects representing the child routes of this route (covered later in this chapter).

Configuring Routes Cont..

- **Defining Links Between Routes**
  - Add links to routes using the RouterLink directive.

```
<li><a [routerLink]="['product']">Product</a></li>
```

  - Alternatively, you can navigate to a route by calling the navigate function on the router

```
this._router.navigate(['product']
```

- **Defining Routes**

```
export const appRoutes: Routes = [
  { path: 'home', component: HomeComponent },
  { path: 'contact', component: ContactComponent },
  { path: 'product', component: ProductComponent },
  { path: '', redirectTo: 'home', pathMatch: 'full' },
  { path: '**', component: ErrorComponent }
];
```

Configuring Routes Cont..

- **Default Route**

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

  - The path is empty, indicates the default route. The default route is redirected to the home path using the RedirectTo argument.

- **Wild Card Route**

```
{ path: '**', component: ErrorComponent }
```

  - The “**” matches every URL. The Router will display the ErrorComponent.
  - Note that the order of the route is important. The Routes are matched in the order they are defined. Since the wildcard route (**) matches every URL and should be placed last.

Configuring Routes Cont..

- **Register the Routes**
  - Routes are registered in root module of the application. I.e. app.module.ts

```
import { appRoutes } from './app.routes';

@NgModule({
  declarations: [
    AppComponent,HomeComponent,ContactComponent,
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpModule,
    RouterModule.forRoot(appRoutes)
ategy */
    /*RouterModule.forRoot(appRoutes, { useHash
egy */
  ],
  providers: [ProductService],
  bootstrap: [AppComponent]
})
```

  - **forRoot method** is used, when you want to provide the service and also want to configure the service at the same time

Configuring Routes Cont..

- **Display the component using Router-outlet**
  - Finally, we need to tell Angular where to display the Component. This is done using the **Router-outlet directive**
  - The RouterOutlet is a directive which tells the Angular where in our page we want to display the view. We do not have to import the routerOutlet and routerLink directives. These directives are imported when we imported RouterModule in our app.module

```
<router-outlet></router-outlet>
```

Passing data to routes

- Route parameters allow us to pass values in our url to our component so we can dynamically change our view content.
- Let's think we are creating an application that displays a product list. When the user clicks on a product in the list, we want to display a page showing the detailed information about that product. To do this you must:
  - add a route parameter ID
  - link the route to the parameter
  - add the ActivatedRoute service that reads the parameter.

Defining Route Parameters

- The route for the component that displays the details for a specific product would need a route parameter for the ID of that product.
- We can define parameter by adding forward slash followed colon and a placeholder (id) as shown below

```
{ path: 'product/:id', component: ProductDetailComponent }
```

- If you have more than one parameter, then you can extend it by adding one more forward slash followed colon and a placeholder

```
{ path: 'product/:id/:id1/:id2', component: ProductDetailComponent }
```

Linking Route Parameters

- The route for the component that displays the details for a specific product would need a route parameter for the ID of that product.
- We can define parameter by adding forward slash followed colon and a placeholder (id) as shown below.
- We, now need to provide both path and the route parameter routerLink directive.
- This is done by adding the productID as the second element to the routerLink parameters array as shown below.

```
<a [routerLink]="['/Product', product.productID]">{{product.name}} </a>
```

Extracting parameters from ActivatedRoute

- Finally, our component needs to extract the route parameter from the URL.
- This is done via the **ActivatedRoute** service from angular/router module to get the parameter value.
- **ActviatedRoute**
  - The **ActivatedRoute** is a service, which keeps track of the currently activated route associated with the loaded Component.
  - The ActivatedRoute class has a params property which is an array of the parameter values, indexed by name. We can use the params array to get the parameter value.
  - To use ActivatedRoute, we need to import it in our component.

ActivatedRoute Cont..

- The ActivatedRoute service has a great deal of useful information including:
- **url:** This property returns an array of Url Segment objects, each of which describes a single segment in the URL that matched the current route.
  - **params:** This property returns a Params object, which describes the URL parameters, indexed by name.
  - **queryParams:** This property returns a Params object, which describes the URL query parameters, indexed by name.
  - **snapshot:** The initial snapshot of this route.
  - **data:** An Observable that contains the data object provided for the route
  - **component:** The component of the route. It's a constant.
  - **routeConfig:** The route configuration used for the route that contains the origin path.
  - **parent: An** ActivatedRoute that contains the information from the parent route when using child routes.
  - **firstChild:** contains the first ActivatedRoute in the list of child routes.
  - **children:** contains all the child routes activated under the current route.

ActivatedRoute Cont..

- To use ActivatedRoute, we need to import it in our component

```
import { ActivatedRoute } from '@angular/router';
```

- Then inject it into the component using dependency injection

```
constructor(private _Activatedroute:ActivatedRoute)
```

- There are two ways in which you can use the ActivatedRoute to get the parameter value
  - **Using Snapshot**
  - **Using observable**

ActivatedRoute Cont..

- **Using Snapshot**
  - The snapshot property returns initial value of the route. You can then access the params array, to access the value of the id.
  - Use this option, if you only need the initial value.

```
this.id=this._Activatedroute.snapshot.params['id'];
```

- **Using observable**
  - You can retrieve the value of id by subscribing to the params observable property of the activateRoute.
  - Use this option if you expect the value of the parameter to change over time.

```
_Activatedroute.params.subscribe(params => { this.id = params['id']; });
```

ActivatedRoute Cont..

- **Why use observable**
  - We usually retrieve the value of the parameter in the **ngOninit** life cycle hook, when the component initialised.
  - When the user navigates to the component again, the Angular does not create the new component but reuses the existing instance. In such circumstances, the ngOnInit method of the component is not called again. Hence you need a way to get the value of the parameter.
  - By subscribing to the observable params property, you will retrieve the latest value of the parameter and update the component accordingly.

Passing Query Parameters

- Query parameters allow you to pass optional parameters to a route such as pagination information.
  - For Example: /product?page=2
  - page=2 is the query parameter
- **Passing Query Parameters**
  - The query parameters are passed to the route using the queryParams directive. This directive must be used along with the routerLink directive as shown below.

```
<a [routerLink]="['product']" [queryParams]="{ page:2 }">Page 2</a>
```

Passing Query Parameters

- **Reading Query Parameters**
  - Reading the Query parameters is similar to reading the Router Parameter. There are two ways by which you can retrieve the query parameters.
  - **Using queryParams observable**

```
this.sub = this.route.queryParams
    .subscribe(params => {
        this.pageNum = +params['pageNum']||0;
    });
```

  - **Using snapshot property**

```
this.pageNum=this.route.snapshot.queryParams["pageNum"];
```

- **Difference Between Query Params and Route Params:**
  - The key difference between query parameters and route parameters is that route parameters are essential to determining route, whereas query parameters are optional

Passing static data to route

- Parent components will usually pass data to their children, but Angular also offers a mechanism to pass arbitrary data to components at the time of route configuration.
- For example, besides dynamic data like a product ID, you may need to pass a flag indicating whether the application is running in a production environment. This can be done by using the data property of your route configuration.

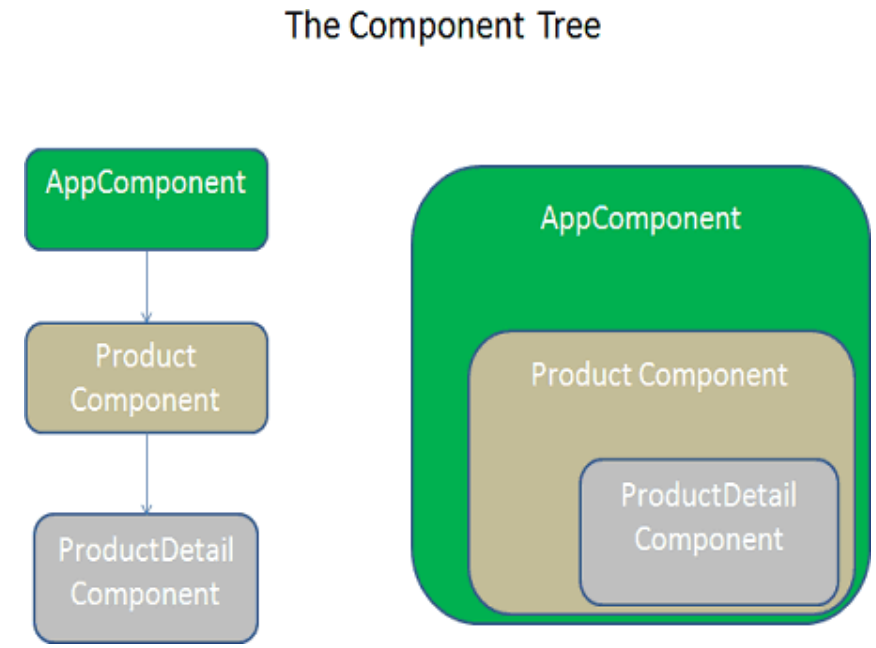
```
{path: 'product/:id', component: ProductDetailComponentParam ,
  data:[{isProd: true}]}
```

```
constructor(route: ActivatedRoute) {
    this.productID = route.snapshot.params['id'];

    this.isProdEnvironment = route.snapshot.data[0]['isProd'];
    console.log("this.isProdEnvironment = " + this.isProdEnvironment);
  }
```

Child routes

- Parent components will usually pass data to their children, but Angular also offers a mechanism to pass arbitrary data to components at the time of route configuration.



Child routes Cont..

- To make ProductDetailComponent as the child of the ProductComponent, we need to add the children key to the product route, which is an array of all child routes as shown below

```
{ path: 'product', component: ProductComponent,
  children: [
     { path: 'detail/:id', component: ProductDetailComponent }
  ],
```

- The child route definition is similar to the parent route definition. It has path and component that gets invoked when the user navigates to the child route.

Guarding routes

- To control whether the user can navigate to or away from a given route, use route guards.
- For example, we may want some routes to only be accessible once the user has logged in or accepted Terms & Conditions. We can use route guards to check these conditions and control access to routes.
- Route guards can also control whether a user can leave a certain route.
- For example, say the user has typed information into a form on the page, but has not submitted the form. If they were to leave the page, they would lose the information.

Guarding routes Cont..

- **Uses of Guards**
  - To Confirm navigational operation
  - Asking whether to save before moving away from a view
  - Allow access to certain parts of the application to specific users
  - Validating the route parameters before navigating to the route
  - Fetching some data before you display the component.
- **Route Guards**
  - The Angular Router supports Five different guards, which you can use to protect the route
    - CanActivate
    - CanDeactivate
    - Resolve
    - CanLoad
    - CanActivateChild

Guarding routes Cont..

- **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.
- **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.
- **Resolve**
  - This guard delays the activation of the route until some tasks are completed. You can use the guard to pre-fetch the data from the backend API, before activating the route
- **CanLoad**
  - This guard is used to guard the routes that load feature modules dynamically
- **CanActivateChild**
  - This guard determines whether a child route can be activated.

How to Build Angular Route Guards

- **Build the Guard as Service**
  - You need to import the corresponding guard from the Angular Router Library using the Import statement.
  - For Example to use CanActivate Guard import the CanActivate using the import statement

```
import { CanActivate } from '@angular/router';
```

  - Next, create the Guard class which implement the selected guard Interface as shown below.

```
@Injectable()
export class ProductGuardService implements CanActivate
{
}
```

Build Angular Route Guards Cont..

- **Implement the Guard Method in the Service**
  - The next step is to create the Guard Method. The name of the Guard method is same as the Guard it implements. For Example to implement the CanActivate guard, create a method CanActivate.

```
canActivate(): boolean
{
    // Check weather the route can be activated;
    return true;
    // or false if you want to cancel the navigation;
}
```

  - The guard method must return either a True or a False value.
  - if it returns true, the navigation process continues. if it returns false, the navigation process stops and the user stays out.

Build Angular Route Guards Cont..

- **Register the Guard Service in the Root Module**
  - As mentioned earlier, guards are nothing but services. Hence they need to be registered with the Providers array of the Module as shown below

```
providers: [ProductService,ProductGuardService]
```

- **Update the Routes to use the guards**
  - Finally, we need to add the guards to the routes array as shown below

```
{ path: 'product', component: ProductComponent, canActivate : [ProductGuardService
] }
```

  - You can add more than one guard as shown below

```
{ path: 'product', component: ProductComponent,
        canActivate : any[],
        canActivateChild: any[],
        canDeactivate: any[],
        canLoad: any[],
        resolve: any[]
}
```

Order of execution of route guards

- A route can have multiple guards and you can have guards at every level of a routing hierarchy.
- **CanDeactivate()** and **CanActivateChild()** guards are always checked first. The checking starts from the deepest child route to the top.
- **CanActivate()** is checked next and checking starts from the top to the deepest child route.
- **CanLoad()** is invoked next, If the feature module is to be loaded asynchronously
- **Resolve()** is invoked last.
- If any one of the guards returns false, then the entire navigation is canceled.

Developing a SPA with multiple router outlets

- A component has one primary **route** and zero or more **auxiliary** routes. Auxiliary routes allow you to use and navigate multiple routes. To define an auxiliary route you need a named router outlet where the component of the auxiliary route will be rendered.

- You can have multiple outlets in the same template:

```
<router-outlet></router-outlet>
<router-outlet name="chat"></router-outlet>
```

multiple router outlets cont..

- Let’s think we want to add to an SPA a chat area so the user can communicate with a customer service representative while keeping the current route active as well. Basically we want to add an independent chat route allowing the user to use both routes at the same time and switch from one route to another

```
const routes: Routes = [
    {path: '',  redirectTo: 'home', pathMatch: 'full'},
    {path: 'home', component: HomeComponent},
    {path: 'chat', component: ChatComponent, outlet:"aux"}
];
```

Splitting an app into modules

- Angular modules allow you to split an application into more than one module, where each module implements certain functionality
- As your app grows, you can organize code relevant for a specific feature. This helps apply clear boundaries for features.
- With feature modules, you can keep code related to a specific functionality or feature separate from other code.
- AppModule, BrowserModule, and RouterModule. AppModule is the root module of an application, but BrowserModule and RouterModule are feature modules. Note the main difference between them: the root module is bootstrapped, whereas feature modules are imported.

Splitting an app into modules

- In a feature module, the @NgModule decorator has to import CommonModule instead of BrowserModule
- This is pretty easy refactoring, compared to the process of removing functionality from a monolithic single-module app.

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';

@NgModule({
  imports: [
    CommonModule
  ],
  declarations: []
})
export class CustomerDashboardModule { }
```

Lazy-loading modules

- In large applications, you want to minimize the amount of code that needs to be downloaded to render the landing page of your application. The less code your app initially downloads, the faster the user will see it.
- This is especially important for mobile apps when they're used in a poor connection area. If your application has modules that are rarely used, you can make them downloadable on demand, or lazy-loaded.
- Lazy loaded routes need to be outside of the root app module, so you'll want to have your lazy loaded features into feature modules.

Lazy-loading modules cont..

- To implement Lazy loading into your application follow these steps
  - We use the property loadChildren instead of component.
  - We pass a string instead of a symbol to avoid loading the module eagerly.
  - We define not only the path to the feature module but the name of the class as well.

```
const routes: Routes = [
  { path: '', redirectTo: 'eager', pathMatch: 'full' },
  { path: 'eager', component: EagerComponent },
  { path: 'lazy', loadChildren: 'lazy/lazy.module#LazyModule' }
];
```

  - When we load our application for the first time, the AppModule along the AppComponent will be loaded in the browser and we should see the navigation system and the text "Eager Component". Until this point, the LazyModule has not being downloaded, only when we click the link "Lazy" the needed code will be downloaded and we will see the message "Lazy Component" in the browser.

Working with forms

- The **Angular** forms are used to collect the data from the user.
- The data entry forms can be a very simple to very complex. It can contain large no of input fields, Spanning multiple tabs. Forms may also contain complex validation logic interdependent on multiple fields

- Angular takes two approaches to building the forms.
  - **Template-Driven Forms** places most of the form handling logic within that form's template
  - **Reactive Forms** places form handling logic within a component's class properties and provides interaction through observables

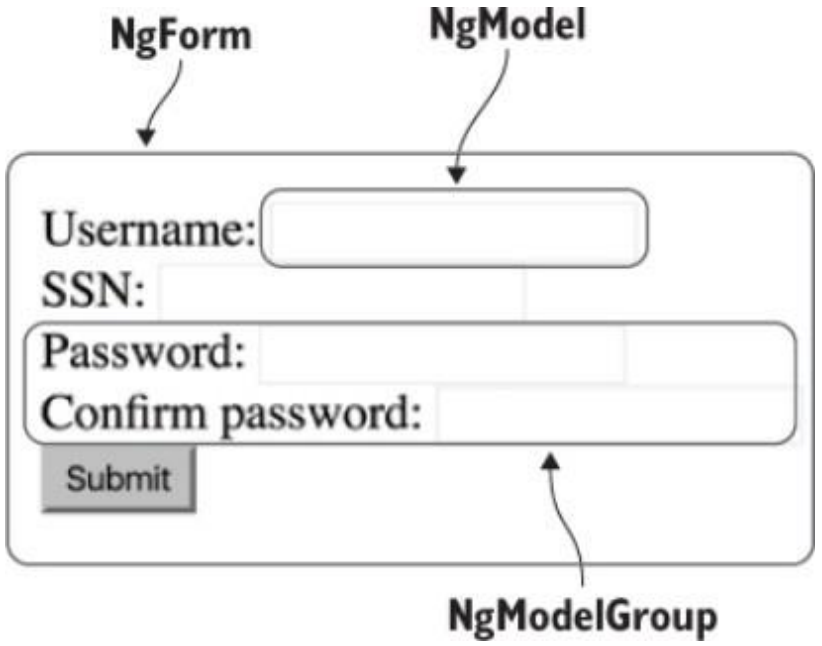
Template driven forms approach

- In **Template driven approach** is the easiest way to build the Angular forms. The logic of the form is placed in the template
- Template-driven forms in Angular allows us to create sophisticated looking forms easily without writing any javascript code
- The FormsModule contains all the form directives and constructs for working with forms.
- Add the import statement to import the forms module from the angular forms package.

```
import { FormsModule } from '@angular/forms';
```

Building Blocks of Template Driven Form

- Here we'll briefly describe the three main directives from FormsModule: **NgModel, NgModelGroup**, and **NgForm**. We'll show how they can be used in the template and highlight their most important features.



NgForm

- NgForm is the directive that represents the entire form. It's automatically attached to every <form> element.
- NgForm intercepts the standard HTML form's submit event and prevents automatic form submission.
- If you want to exclude a particular <form> from being handled by Angular, use the ngNoForm attribute.
- ngSubmit event we use to submit the form data.

```
<form #contactForm="ngForm" (ngSubmit)="onSubmit(contactForm)" >
```

NgModel

- In the context of the Forms API, NgModel represents a single field on the form.
- Like NgForm, the NgModel directive has an exportAs property, so you can create a variable in the template that will reference an instance of NgModel and its value property.

```
<div class="form-group">
    <label for="city">City</label>
    <input type="text" class="form-control" name="city" ngModel >
</div>
```

NgModelGroup

- NgModelGroup represents a part of the form and allows you to group form fields together.
- All the child fields of NgModelGroup become properties on the nested object.

```
<div ngModelGroup="address">

    <div class="form-group">
        <label for="city">City</label>
        <input type="text" class="form-control" name="city" ngModel >
    </div>

    <div class="form-group">
        <label for="street">Street</label>
        <input type="text" class="form-control" name="street" ngModel >
    </div>

    <div class="form-group">
        <label for="pincode">Pin Code</label>
        <input type="text" class="form-control" name="pincode" ngModel>
    </div>

</div>
```

Building the Template

- The **ngForm** directive is what makes the Angular Template driven Forms work.
- The Form data is submitted to the Component using the Angular directive named **ngSubmit.**

```
<form #contactForm="ngForm" (ngSubmit)="onSubmit(contactForm)" >

    <div class="form-group">
        <label for="firstname">First Name</label>
        <input type="text" class="form-control" name="firstname" ngModel>
    </div>

    <div class="form-group">
         <label for="lastname">Last Name</label>
         <input type="text" class="form-control" name="lastname" ngModel>
    </div>

    <div class="form-group">
          <button type="submit">Submit</button>
    </div>

    <pre>{{contactForm.value | json }} </pre>
</form>
```

Receive the data in Component class

- Now we need to receive the data in Component class from our form. To do this we need to create the **onSubmit** method in our component class
- The submit method receives the reference to the **ngForm** directive

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

@Component({
    selector: 'app-root',
    templateUrl: './app.component.html',
})
export class AppComponent
{
    onSubmit(contactForm) {
        console.log(contactForm.value);
    }
}
```

Initializing the Value

- The simplest way is to initialize value of the form element in Component class and Set up one way or two way binding in the Template.
- The correct way is to create an Angular Form is to build a model class. Let us build a model for our Form. Add it in *app.component.ts*

```
class Contact {
      firstname: string ;
      lastname: string ;
      address: {
          city:string
          street: string
          pincode: string
      }
}
```

Initializing the Value cont..

- And in our *AppComponent* Class create an instance of the Contact class and initialize it with the default values. In Real life application, you may initialize it with the data from the database.

```
contact:Contact = {
    firstname:"Sachin",
    lastname:"Tendulkar",
    address: { city:"Mumbai",street:"Perry Cross Rd", pincode:"400050"}
};
```

Initializing the Value cont..

- **One way binding**
  - Now we can use our contact model and bind it to the Angular Forms model using **one-way binding** syntax [ngModel]

```
<input type="text" class="form-control" name="firstname" [ngModel]="contact.firstn
ame">
```

- **Two way binding**
  - You can also use the **two-way binding** syntax [(ngModel)]

```
<input type="text" class="form-control" name="firstname" [(ngModel)]="contact.firs
tname">
```

Track control state and validity

- Angular Applies 3 classes to the form control based on its states

State	Class if true	Class if false
The control has been visited.	ng-touched	ng-untouched
The control's value has changed.	ng-dirty	ng-pristine
The control's value is valid.	ng-valid	ng-invalid

ngModel Properties

- Although the above classes can be used to provide visual feedback. Angular provide alternative.
- For each classes, angular provide associated property on ngModel directive.
- The property is same as class with “ng-” removed.
- Create a template reference variable to get reference to the ngModel directive.

ngModel properties

Class	Property
ng-untouched	untouched
ng-touched	touched
ng-pristine	pristine
ng-dirty	dirty
ng-valid	Valid
ng-invalid	invalid

Validating Template-driven forms

- The Angular forms architecture supports two ways to validate the Form. One is **Built-in validation** and other is **custom validation**
- **Built-in Validators**
  - The Angular **Built-in validators** use the HTML5 validation attributes like **required**, **minlength**, **maxlength** & **pattern**.
    - **required**: There must be a value
    - **minlength**: The number of characters must be more than the value of the attribute.
    - **maxlength**: The number of characters must not exceed the value of the attribute.
    - **pattern**: The value must match the pattern.

How to add Built-in validators

- **Disabling the Browser validation**
  - 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.

```
<form #contactForm="ngForm" (ngSubmit)="onSubmit(contactForm)" novalidate>
```

- **Adding Required Validator**
  - Now let us make the firstname as the required field. This can be done by adding the **required** attribute as shown in below

```
<input type="text"  class="form-control"  name="firstname" ngModel #firstname="ngM
odel" required >
```

How to add Built-in validators cont..

- **Using FormControl instance directly**
  - In the previous example, we have used **FormGroup** instance to access the **FormControl**.
  - Now we can also get the reference to the firstName **FormControl** by creating a Template variable "**firstname**" and assigning it to ngModel like **#firstname="ngModel"**

```
<input type="text" name="firstname" ngModel required #firstname="ngModel">
```

- **Disable Submit button**
  - We need to disable the submit button if our form is not valid.

```
<button type="submit" [disabled]="!contactForm.valid">Submit</button>
```

How to add Built-in validators cont..

- **Displaying the Validation/Error messages**
  - We need to Provide a short and meaningful error message to the user.
  - We can get the reference to the FormControl associated with the firstName field from the template variable *"contactForm"*
  - We are using the **ngIf** directive to hide/show the error message to the user.

```
<div *ngIf="!contactForm.controls.firstname?.valid && (contactForm.controls.firstn
ame?.dirty || myForm.controls.firstname?.touched)" class="alert alert-danger">
   <div [hidden]="!contactForm.controls.firstname.errors.required">
        First Name is required
   </div>
</div>
```

Reactive Forms

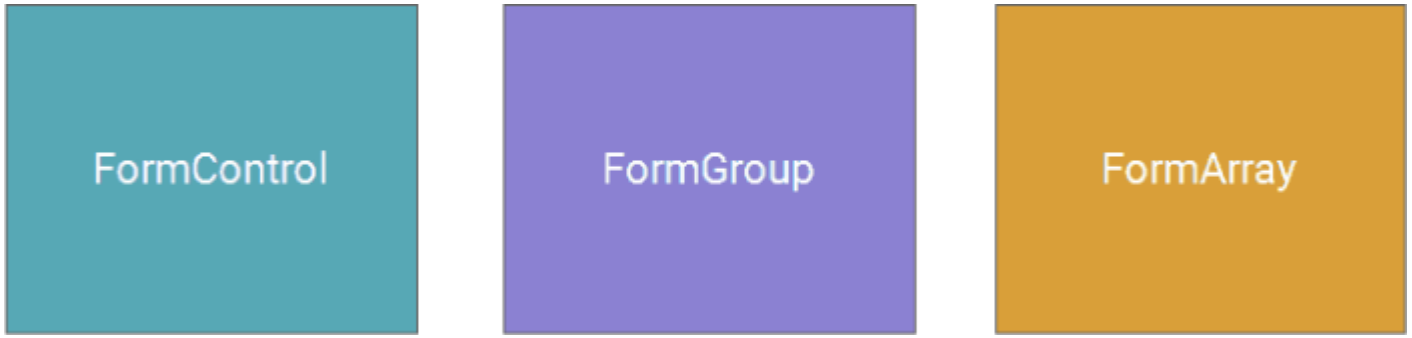
- In Reactive Forms (also known as Model Driven Forms) the form is built in the component class, unlike Template Driven Model where the model is built-in templates.
- Advantages of Reactive Forms
  - The Angular Model Driven Forms enable us to test our forms without being required to rely on end-to-end tests.
  - The model driven Forms allows you to have an ability to setup validation rules in code rather than as directive in Template.
  - You can subscribe to field value changes in your code. Since everything happens in Component class, you can easily write unit Tests.

Building Blocks of Reactive Forms

- For building reactive forms we need to import **ReactiveFormsModule** instead of **FormModule** to make use of Reactive Forms. We should also add the **ReactiveFormsModule** to the imports metadata property array.

```
import { ReactiveFormsModule } from '@angular/forms';
```

- The Reactive Forms module consists of three Building blocks



FormControl

- A FormControl represents a single input field in an Angular form.
- The FormControl is an object that encapsulates all these information related to the single input element. It Tracks the value and validation status of each of these control

```
First Name : <input type="text" name="firstname" />
```

FormControl cont..

- You can use FormControl to set the value of the Form field. You can find the status of form field like (valid/invalid, pristine/dirty, touched/untouched) etc. You can add validation rules to it.

```
    let firstname= new FormControl(); //Creating a FormControl in a Model driven f
orm
```

```
firstname.errors      // returns the list of errors
firstname.dirty       // true if the value has changed (dirty)
firstname.touched     // true if input field is touched
firstname.valid       // true if the input value has passed all the validation
```

FormGroup

- FormGroup is a group of FormControl instances.
- Often forms have more than one field. It is helpful to have a simple way to manage the Form controls together.

```
city : <input type="text" name="city" >
Street : <input type="text" name="street" >
PinCode : <input type="text" name="pincode" >
```

FormGroup cont..

- We can group these input fields under the group address as shown below.

```
let address= new FormGroup({
    street : new FormControl(""),
    city : new FormControl(""),
    pinCode : new FormControl("")
})
```

- A typical Angular Form can have more than one FormGroup. A FormGroup can also contain another FormGroup.

FormArray

- The FormArray‘s are array of FormControls.
- The FormArray’s are very much similar to FormGroups. Here FormControls are represented as an array of objects.
- The key difference is that its data gets serialized as an array. This might be especially useful when you don’t know how many controls will be present within the group, like dynamic forms.
- The FormArray are defined as shown below.

```
let address= new FormArray({
    street : new FormControl(""),
    city : new FormControl(""),
    pinCode : new FormControl("")
})
```

Building the Form Template

- We are going to use the same Template, which we used while building the **Template Driven Forms.**

```
<form [formGroup]="contactForm" (ngSubmit)="onSubmit()">
    <div class="form-group">
        <label for="firstname">First Name</label>
        <input type="text" class="form-control" name="firstname" formControlName=
"firstname">
    </div>

    <div class="form-group">
        <label for="lastname">Last Name</label>
        <input type="text" class="form-control" name="lastname" formControlName=
"lastname">
    </div>

    <div class="form-group">
         <button type="submit">Submit</button>
    </div>
</form>
```

Building the Model

- In the Template driven approach, we used ngModel and ngModelGroup directive on the HTML elements, which created the FormGroup & FormControl instances behind the scene.
- In the Model Driven approach, it is our responsibility to create the **FormGroup** and **FormControl.**
- First we need to import FormGroup, FormControl and Validator form the angular/forms. Open the app.component.ts and write the following import statement.

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

Building the Model cont..

- **FormGroup**
  - The FormGroup is created with the following syntax.

```
contactForm = new FormGroup({})
```

  - The FormGroup takes 3 arguments. a collection of child FormControl, a validator, and an asynchronous validator. The Validators are optional.
- **FormControl**
  - The first argument to FormGroup is the collection of child FormControl. They are added using the FormControl method as shown.

```
contactForm = new FormGroup({
    firstname: new FormControl(),
    lastname: new FormControl()
})
```

Building the Model cont..

- **FormControl Arguments**
  - A FormControl takes 3 arguments. a default value, a validator and an asynchronous validator. All of them are optional.
  - **Default Value**
    - You can pass default value as either as a string or as an object of key-value pair. When you pass object you can set both the value and the whether or not the control is disabled.

```
//Setting Default value as string
firstname= new FormControl('Rahul');
```

```
//Setting Default value & disabled state as object
firstname: new FormControl({value: 'Rahul', disabled: true}),
```

Building the Model cont..

Sync Validator

- The second parameter is a sync validator or a list of sync Validators. Angular has some built in Validators such as required and minLength
- You can pass with Validator function as shown below.

```
firstname: new FormControl('', Validators.required),
```

- In case you have more than one validators, then you can combine all of them using Validators. compose method or as an array of validators as shown below.

```
//using Validators.compose method
firstname: new FormControl('', Validators.compose([ Validators.required,Validators
.minLength(10)]))
```

```
//This is shorthand for the above.
firstname: new FormControl('',[ Validators.required,Validators.minLength(10)]),
```

Use FormBuilder to build Reactive Forms

- As you can see from the previous example , we need to call a new **FormControl** / **FormGroup** several times to build our model. To make this task bit pleasant the angular provides us with the high-level API **FormBuilder**
- FormBuilder is a helper class that helps us to easily add **FormControls**, **FormGroups**& **FormArrays** to build the angular forms. The **FormBuilder** is part of the **@angular/forms** module
- There are three methods available in FormBuilder
  - control – creates a new FormControl
  - group – creates a new FormGroup
  - Array – creates a FormArray

Build Reactive Forms cont..

- **Import FormBuilder**
  - To Use Formbuilder, we need to import it into our component as shown below.

```
import { FormBuilder } from '@angular/forms';
```

- **Inject FormBuilder into our component**
  - The **FormBuilder** is then injected into our component using the dependency injection. This is done in the constructor of appComponent class.

```
constructor (private fb: FormBuilder) { }
}
```

  - In the above code, the instance of the FormBuilder is created and assigned to the local variable *fb*. We can use this local variable to build our form.

Build Reactive Forms cont..

- **Creating the FormGroup**
  - The group method of the FormBuilder creates and returns the FormGroup instance.

```
this.contactForm = this.fb.group({ })
```

- **Adding the FormControls**
  - The FormGroup objects hold our FormControls. The FormControls are added to the FormGroup by passing them to the Group method as an object of key-value pairs.

```
this.customerForm = this.fb.group({
    firstname: [],
    lastname: []
});
```

Reactive Forms Validation

- In Reactive Forms we use Validator method to validate form's controls.
- A Validator is a function that checks the **FormControl** or a **FormGroup** and returns a list of errors. If the Validator returns a null means that validation has passed.
- First, we need to import Form FormGroup, FormControl and Validator from the angular/forms.

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

Reactive Forms Validation cont..

- **Adding Required Validator**
  - The Required validator can be added at the time of instantiating the firstname FormControl. The second argument of the FormControl takes the Sync Validator.

```
this.contactForm = this.fb.group({
    firstname: ['',Validators.required],
    lastname: [],
    address: this.fb.group({
        city:[],
        street: [],
        pincode: []
    })
 });
```

- **Disable Submit button**
  - We have added the **required** validator to the firstname field. But the Form still submits with the value of the firstname is empty.

```
<button type="submit" [disabled]="!contactForm.valid">Submit</button>
```

Reactive Forms Validation cont..

- **Displaying the Validation/Error messages**
  - We can get the reference to the FormControl associated with the firstName field from the Form Model *contactForm*.

```
<div *ngIf="!contactForm.controls.firstname?.valid && (contactForm.controls.firstn
ame?.dirty ||contactForm.controls.firstname?.touched)" class="alert alert-danger">
    <div [hidden]="!contactForm.controls.firstname.errors.required">
        First Name is required
    </div>
</div>
```

  - All the validators either return an object if there is any validation error or null if everything is ok. You will get reference to all these validation error objects from formControl.errors object.