Tutorial #12 Modules

1. Angular is a modular-based architecture
    - There are lot of modules which are built-in
    - Examples
        - BrowserModule
        - BrowserAnimationsModule

    - Angular Material Library
        - MatButtonModule
        - MatDropDownModule

2. All the code and functionality is grouped in a module

3. Whenever you see a @ symbol - it's a decorator

4. What modules consist
    - declarations
        - this is where we will add all the components of the module
    - imports
        - we can import modules inside a module

    - providers
        - services that we need will be injected here

    - Bootstrap
        - what is the first component, the module should load

    - exports
        - is to export and expose the component outside of the module

5. Every Angular application should have at least 1 module

6. By default, the Angular framework provides us with AppModule

7. The AppModule will have a component by the name
    - AppComponent

8. Whenever we are building Angular applications
    - We will always think of Modules first

    E.g
        Contacts

Users
Leads
Opportunites
Settings
Profile
Authenctication

Free User
- Contacts
- Users

Premium User
- Contacts
- Users
- Leads
- Opportunites

Enterprise Users
- Contacts
- Users
- Leads
- Opportunites
- Settings
- Profile
- Authenctication

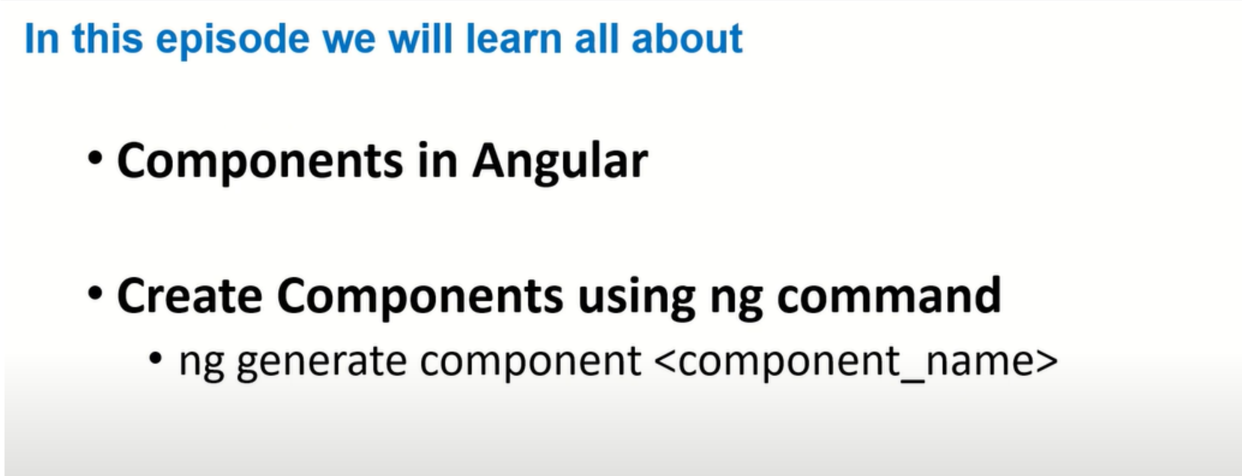
9. Feature Modules
   - You can turn on or off the modules based on conditions

10. Modules - Grouping
   - components
   - services
   - pipes
   - directives

11. Create a custom Module
ng generate module contacts

ng generate module leads
ng generate module settings
ng generate module Opportunities
ng generate module authentication

Tutorial #13 - Components



Episode #13 - Angular Components

1. Components are the most important and basic building blocks of Angular apps

2. Authentication Module
    - new-user
    - login
    - forgot-password
    - reset-password

3. Component is a smallest functionality that you will implement in your application

4. When we group multiple Components it becomes a module

5. We can have paren-child relationship of components

6. We can have components inside components

7. Tree-herirachy of components

```
Dashboard
    display-contact-list
        contact-grid
            contact-import
            contact-export
        contact-options
```

8. lets create some custom components

ng g component add-contact
ng g component edit-contact
ng g component list-contacts
ng g component delete-contact

9. Every component has 4 files auto-generated with it
   - component.html -> view or html or template file -> UI
   - component.ts -> it will be a class file which will have methods -> Logic
   - component.spec.ts -> It will have the unit test script for component
   - component.scss -> stylesheet of the component

   HOMEWORK
   -> verify all the components generated and go through the code

10. Component decorator inside the component.ts file

```
@Component({
  selector: 'app-add-contact',
  templateUrl: './add-contact.component.html',
  styleUrls: ['./add-contact.component.css']
})
```

selector -> unique identifier for the component
   -> id of the component
   -> using this selector we will use the component
templateUrl -> your HTML code
   - component.html file
styleURLS -> for linking your component stylesheet
   - component.scss

Tutorial #14 - Component Lifecycle Hooks

- **8 Lifecycle Hooks**
  - ngOnChanges()
  - ngOnInit()
  - ngDoCheck()
  - ngAfterContentInit()
  - ngAfterContentChecked()
  - ngAfterViewInit()
  - ngAfterViewChecked()
  - ngOnDestroy()

- **ngOnChanges()**
  - Used in pretty much any component that has an input.
  - Called whenever an input value changes
  - Is called the first time before ngOnInit
- **ngOnInit()**
  - Used to initialize data in a component.
  - Called after input values are set when a component is initialized.
  - Added to every component by default by the Angular CLI.
  - Called only once
- **ngDoCheck()**
  - Called during all change detection runs
  - A run through the view by Angular to update/detect changes

ARC Tutorials

- **ngAfterContentInit()**
  - Called only once after first ngDoCheck()
  - Called after the first run through of initializing content
- **ngAfterContentChecked()**
  - Called after every ngDoCheck()
  - Waits till after ngAfterContentInit() on first run through
- **ngAfterViewInit()**
  - Called after Angular initializes component and child component content.
  - Called only once after view is initialized

ARC Tutorials

- **ngAfterViewChecked()**
  - Called after all the content is initialized and checked. (Component and child components).
  - First call is after ngAfterViewInit()
  - Called after every ngAfterContentChecked() call is completed
- **ngOnDestroy()**
  - Used to clean up any necessary code when a component is removed from the DOM.
  - Fairly often used to unsubscribe from things like services.
  - Called only once just before component is removed from the DOM.

In my experience some of the most used ones are:

- ngOnChanges()
- ngOnInit()
- ngAfterViewInit()
- ngOnDestory()

Angular component lifecycle hooks

1. By default we have ngonit

2. Whichever lifecycle hooks we want to use
    1. import it in the class
    2. Extend the implements interface
    3. Implement the method

```
export class AddContactComponent implements OnInit, OnChanges, OnDestroy {

  constructor() { }

  ngOnInit(): void {
  }

  ngOnChanges(){

  }

  ngOnDestroy(){
```

3. We can have any number of lifecycle hooks implemented

4. Its too early for us to implement all of them today

    - We will revisit this topic again component communication

    - Component Communication
        - Between components
            Parent to Child
            Child to Parent

5. I am here to teach to end to end- to help you master Angular

Tutorial #15 - Component Communication

Episode #15
Communication between various Angular components

Contacts -> Module

- contact-listing -> parent component
  - contact-grid -> child component
  - contact-tools -> child component
    - download-pdf -> sub-child component
    - download-excel

Leads -> Module

- leads-listing -> parent component

Components are hierarchial
Parent-child relationship

p1
- child1
  - sub-child1
- child2

p2
- p2-child1

1. Communications between the related components
   - parent component -> child components
     - @Input
   - parent components <- child components
     - @Output

   - Leads Module
     - leads-listing
       - leads-grid
       - leads-tools
         - download-excel
         - download-pdf

2. communication between totally unrealted components

Component1 -> Services <-   Component2

services
  is a common reusable piece of functionality shared between different components

Tutorial #16 - Templates in Angular Component

Episode #16 - Templates in Angular components

1. whenever we generate a component
   - 4 files
     - template file (.html)
     - style.scss (stylesheet)
     - class (component.ts file)
     - spec (unit test file)

2. This is totally based on the choice at installing Angular app
   - if you selected scss
   - if you selected css
     style.css

3. <comp_name>.component.html
   - It works!

4. <comp_name>.component.ts file

5. Decorator it gives definitinion and meaning to the
   @ - it has prefix of @

6. by default Angular will add "app" as prefix
   - selector -> "app-leads-listing"
   - unique identifier to identify this component
   - <app-leads-listing>

     -> Can you change the default "app" prefix?
       -> YES - we can change it throughout the app
         -> "app" -> "arc-tutorials"
         -> angular.json -> change prefix

-> what will happen if i change?
  -? Nothing happenns. Only thing you change, make sure you update with latest info
-> will your app work or will it crash?
  -> If you have updated the necessary components with latest prefix

7. Templates in Components

  - Two ways of using templates in Components
    - templateUrl
      - link the html file

    - template

  - templateUrl
    -> is always 1 single html file

  - template
    -> we will pass the template itself instead of a html file
    -> we just the HTML code that we want the component to display
    -> we will use "backtick" and NOT single quote
      backtick key can be found on left top side `

8. stylesURL
  -> is an array
  -> it can take multple stylesheets as input
  -> it can be one or more stylesheets

9. Hands-on examples
  Profile -> Modules
    list-profile -> component

Tutorial #17 - Directives in Angular

1. Directive is a way to extend our HTML including both view as well as behaviour

2. Directives are used to extend the power of HTML

3. Mainly 3 types of directives

   Component Directive

   Structural Directive
      ngIf
      ngFor
      ngSwitch

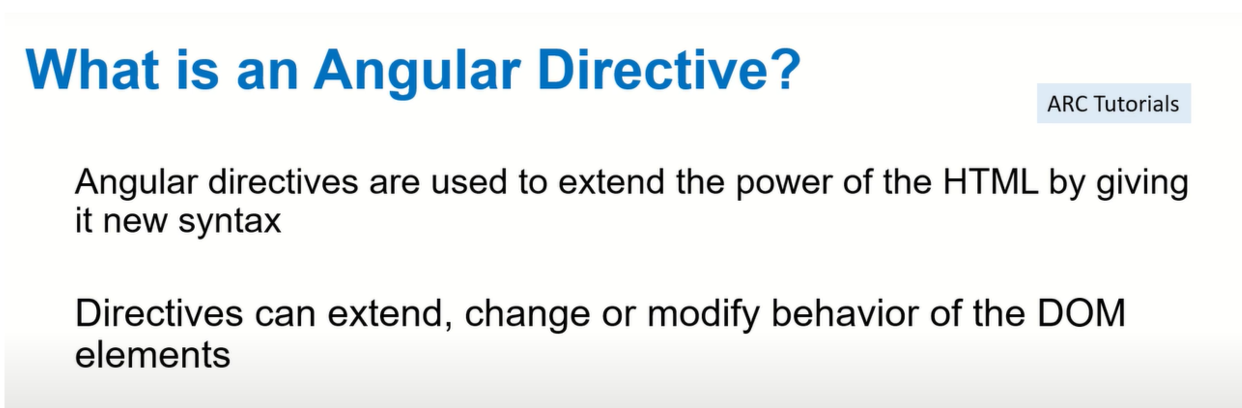
   Attribute Directive
      ngClass
      ngStyle

4. Examples
   AppComponent -> Component Directive

5. Custom Directives

   ng g directive highlight

   - This is an advanced topic
   - We will revisit this topic again once we have our foundations better



- **There are 3 types of Directives**

ARC Tutorials

- **Component Directives**
  - Every Angular application must have at least 1 component
  - Have it's own templates
  - Events attached
- **Structural Directives**
  - Updates structure of the view
  - ngFor, ngIf and ngSwitch
- **Attribute Directives**
  - ngStyle, ngClass

Custom Directive

Component Directives

```
@Component({
  selector: 'arc-tutorials-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss']
})
```

Custom Directive

- **How to generate directives**
  - **ng generate component <component-name>**
  - **ng generate directive <directive-name>**

Tutorial #18 - Structural Directives

Espisode #18 - Structrual Directives

1. Structrual Directives will help us in extending, adding or removing elements from the HTML page

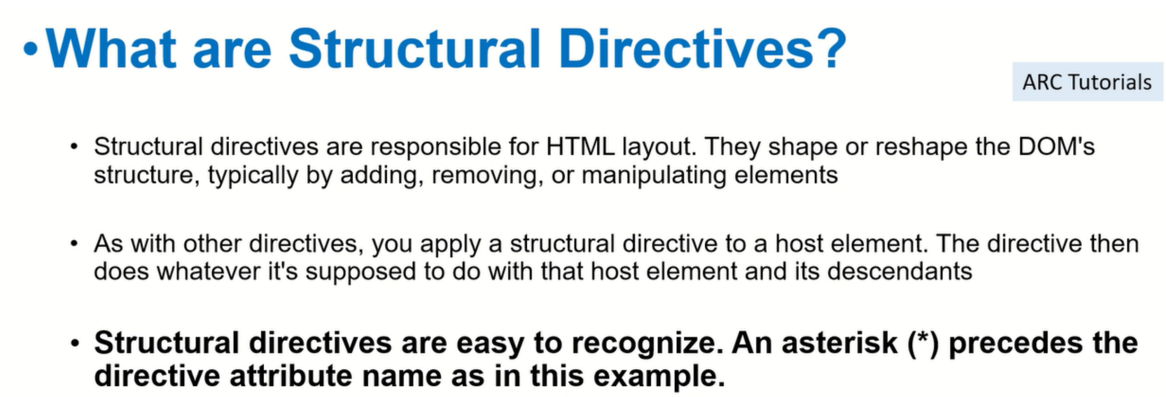
2. DOM Mainupulation

3. 3 types of Structrual Directives

   ngIf
   ngFor
   ngSwitch

4. asterisk (*) means its a built in Structrual directive

E.g ngIf
   <div *ngIf="<condition>"> Value 1</div>



• Types of Structural Directives:

- ngIf
- ngFor
- ngSwitch

Tutorial #19 - ngIf

Episode #19 - ngIf

1. It will always have * astrisk symobl
2. Built in structural directive
3. The ngIf condition will evaluate the expression and result true or false
4. We can use OR (||), And (&&) and we can also use NOT(!) operators inside ngIf

5. We can have multiple use cases

   - ngIf -> done
      -> logical opeartors
         OR (||)
         AND (&&)
         NOT (!)
   - ngIf else ->
   - ngIf then else

•ngIf

- Is a built-in structural directive that can add or remove elements
- Add * symbol in ngIf in template
- Conditionally includes a template based on an expression
- Resolves to true or else result of any given expression
- Add or remove an element dynamically
- Example code: <div *ngIf="condition"></div>

•*ngIf with Else statement

- Used along with **ngIf** statement
- **Else** statement will show the block based on false condition of the if statement
- For using **else** statement – we need to use <ng-template> and pass **template reference variable** along with it
- For e,g
  - <div ***ngIf="showValue; else showMessage"**>Show value</div>
  - <ng-template **#showMessage**>Showing else msg</ng-template>

- ## *ngIf with Then and Else statement

- Used along with **ngIf** statement
- When the condition is true – using the template variable reference, the then block will be executed
- When the condition is false – using the template variable reference, the else block will be executed
- For e,g
  - <div ***ngIf="showValue then thenBlock else showMessage"**>Show value</div>
  - <ng-template **#thenBlock**>Showing then msg</ng-template>
  - <ng-template **#elseBlock**>Showing else msg</ng-template>

```
<div *ngIf="success_msg; then updated_msg; else error_msg" style="backgroun
    user added successfully
</div>

<ng-template #updated_msg>
    user was updated
</ng-template>

<ng-template #error_msg>
    user was not added
</ng-template>
```

Tutorial #20 - ngFor

• ngFor

ARC Tutorials

- Similar to for statements in other programming languages
- Is a built-in structural directive – which modifies the DOM structure
- Loop the elements to display the array data in the template
- Syntax : **<div *ngFor="let ele of collection"> </div>**

- **Provides local variables in the array data**
  - Index – gets the current index of the current element in iteration
  - First – returns true if the current element is the first element in iteration
  - Last - returns true if the current element is the last element in iteration
  - Even - returns true if the current element is the even element in iteration
  - Odd - returns true if the current element is the odd element in iteration

```
<ul>
    <li *ngFor="let contact of contacts; index as i; first as f">
        <div *ngIf="f">this is first record</div>
        <div>
        {{ i }} {{ contact.firstName }} {{ contact.lastName }}
        </div>
    </li>
</ul>
```

Tutorial #21 - ngSwitch

• ngSwitch

ARC Tutori

- It’s a built-in directive and starts with the [] bracket symbol
- Very similar to switch statements in any other programming languages
- Allows element to be shown or hidden based on a condition expression
- Unlike if statement – switch can take multiple value parameters for condition check
- We can also define a default action for the switch
- There are mainly 3 important elements of **ngSwitch**
  - ngSwitch
  - ngSwitchCase
  - ngSwitchDefault

Subscribe and Ask your doubts in comments

• ngSwitch Example

- It’s a built-in directive and starts with the * symbol

- <div [ngSwitch]="switch_expression">
-   <div *ngSwitchCase="match_expression_1">...</div>
-   <div *ngSwitchCase="match_expression_2">...</div>
-   <div *ngSwitchCase="match_expression_3">...</div>
-   <div *ngSwitchDefault>...</div>
- </div>

```html
<div [ngSwitch]="superPower">
    <div *ngSwitchCase="'wonderWoman'">Display Wonder woman super powers</div>
    <div *ngSwitchCase="'superMan'">Display Super Man super powers</div>
    <div *ngSwitchCase="'heMan'">Display He MAN super powers</div>
    <div *ngSwitchCase="'SpiderMan'">Display Spiderman super powers</div>
    <div *ngSwitchDefault>Super power is in learning Spiderman super powers</div>
</div>


<div [ngSwitch]="tax">
    <div *ngSwitchCase="10">Display tax is 10</div>
    <div *ngSwitchCase="30">Display tax is 30</div>
</div>
```

Tutorial #22 - ngStyle

Episode #22 - ngStyle

1. ngStyle is a built in directive used to set style/css properties

2. [ngStyle]

3. Any/All css properties using ngStyle

4. more than 1 property on any DOM element

5. We can also pass dynamic values to ngStyle

6. ngStyle - hands-on examples

    6.1 basic use case of ngStyle - setting a value using ngStyle
    6.2 dynamic value from component
    6.3 ngStyle with conditional operators

7. Common mistakes
    - Not giving correct {} braces
    - Not putting correct double/single qoutes where required
    - Spelling mistakes in defining the css properties
    - DO NOT put qoutes for the variables - it will become strings

```
<h3>Learning ngStyle</h3>

<div [ngStyle]="{'background-color':'green'}">This is green DIV</div>

<div style="background-color:green"></div>
```

```
<h3>Learning ngStyle</h3>

<div [ngStyle]="{'background-color':'green', 'color': '#fff'}">This is green DIV</

<div [ngStyle]="{'background-color': styleProp}">This is Dynamic Color DIV</div>

<div [ngStyle]="{'color': txtColor === 'blue'? 'blue' : 'green'}">
    This is conditional DIV
</div>
```

[Tutorial #23 - ngClass]

Episode #23 - ngClass

1. ngClass is a directive used for setting the class name of DOM elements

2. we can provide more than 1 class names using ngClass
   -> we can re-use the code for multiple DOM elements

3. We can pass strings

4. we can pass array values

5. we can pass objects

6. Common mistakes
   - Not writing in correct qoutes
   - not putting ngClass in square brackets
   - using quotes for variables

hands-on examples:

1. <div [ngClass]="'c1'">This is ngClass example</div>

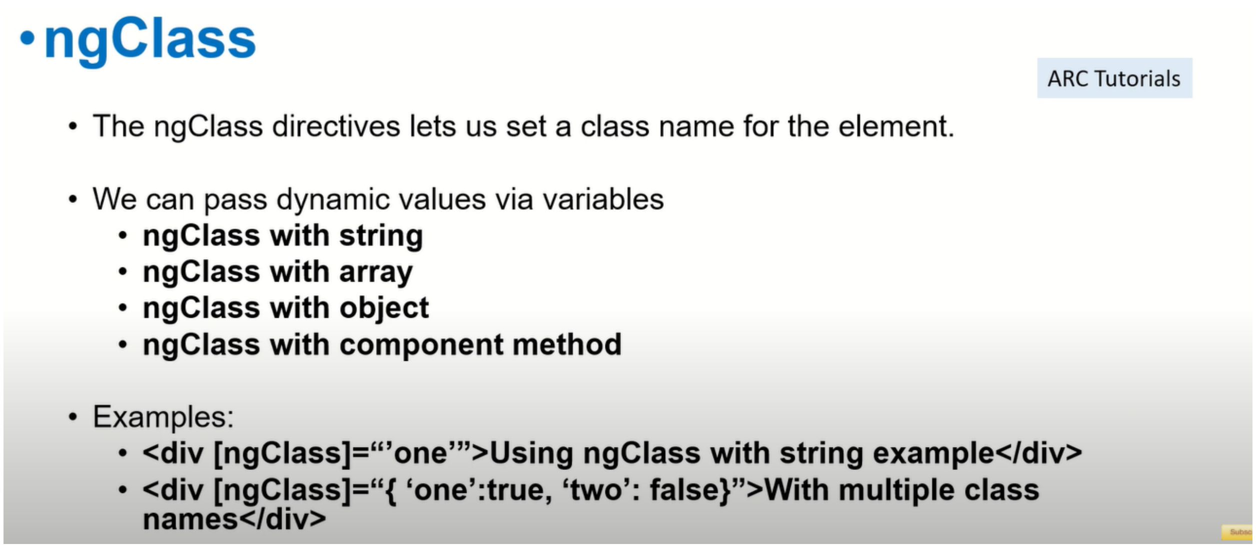
2. <div [ngClass]="'c1 c2 c3 c4'">This is ngClass Multiple classes example</div>

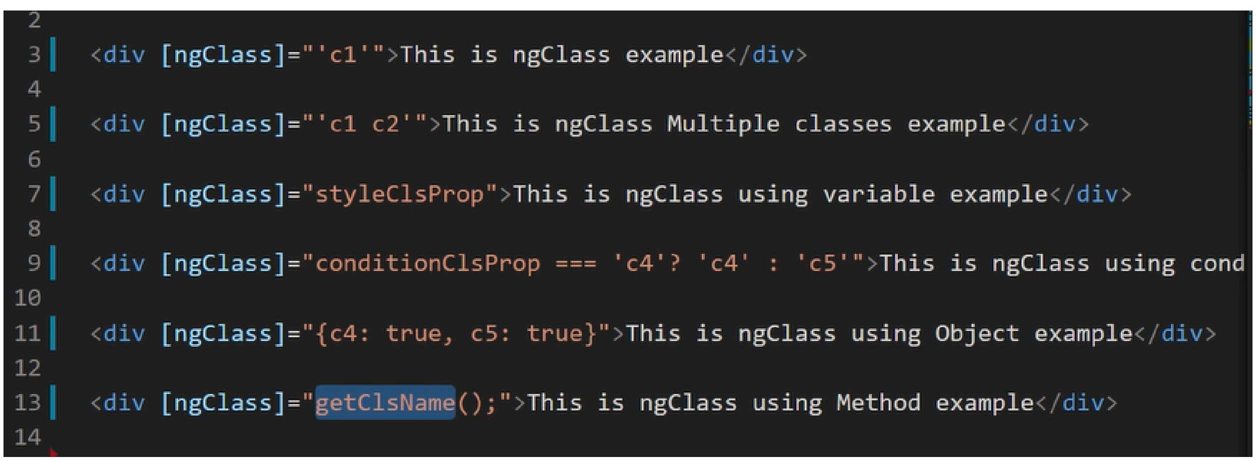
3. <div [ngClass]="styleClsProp">This is ngClass using dynamic variable example</div>

4. <div [ngClass]="conditionClsProp === 'c4'? 'c4' : 'c5'">This is ngClass using conditional check example</div>

5. <div [ngClass]="{c4: true, c5: false}">This is ngClass using Object example</div>

6. Try out the method returning class name to [ngClass]





Tutorial #24 - Data Binding

Episode #24 - Data Binding

1. Each component
   users.component.html -> view/HTML/UI
   users.component.scss -> classNames
   users.component.ts -> what data to be displayed/expressions
   users.component.spec.ts

2. Data is spread throughout these files

3. Data Binding intercating with data of the component

4. Data can be from component to template -> one way

5. Data can be from template to component -> one way

6. Two way to/from component
   from/to template

7. One way/Two way is nothing but representation of data flow
   -> one way

   -> two-way
   <-

• Data Binding

ARC Tutorials

- Means to bind the data from view (Template) to Controller (Component class) and vice versa
- Data binding as the name suggest – interacting with data
- Defines how the data flows and how the data gets updated based on business logic

• Data Binding

- **One-way Data Binding**
  - Component to View
    - Interpolation
    - Property Binding
    - Style Binding
    - Attribute Binding
  - View to Component
  - Event Binding
- **Two-way Data Binding**
  - Data flows from view to component and back to component from the view

Tutorial #25 - Interpolation

• Interpolation

ARC Tutorials

- Is a technique that allows the user to bind data from component to view(template)
- The data flow is only one-way i.e from component to view
- Can be used for integers, strings, objects, arrays and much more
- Syntax for defining Interpolation is double curly braces
- **{{ variable_name }}**

Tutorial #26 - Property Binding

• Property Binding

ARC Tutorials

- Is a technique that allows the user to bind properties of elements from component to view(template)
- The data flow is only one-way i.e from component to view
- Can be used for all properties like title, placeholder, innerHTML, src etc
- Syntax for defining Interpolation is double curly braces
- **[property]=“’expression’”**

```
<div>{{ pageHeading }}</div>

<p [style.color]="txtColorVal">Page Count Value is {{ pageCount }}</p>

<a href="#">this is a link</a>

<img [src]="imgUrl" [alt]="imgAlt" />

<div *ngIf="isUserLoggedIn">
    <div [innerHTML]="userObject.firstName"></div>
</div>
```

Tutorial #27 - Attribute

Episode #27 - Attribute Bindings

1. Attribute binbing is a unidirectional - one-way data binding

2. Syntax -> [attr.attribute_name]="expression"

3. The functionality may seem almost similar to property binding

4. one of the most important Q asked in interviews is

-> what is the difference between property binding and attr binding

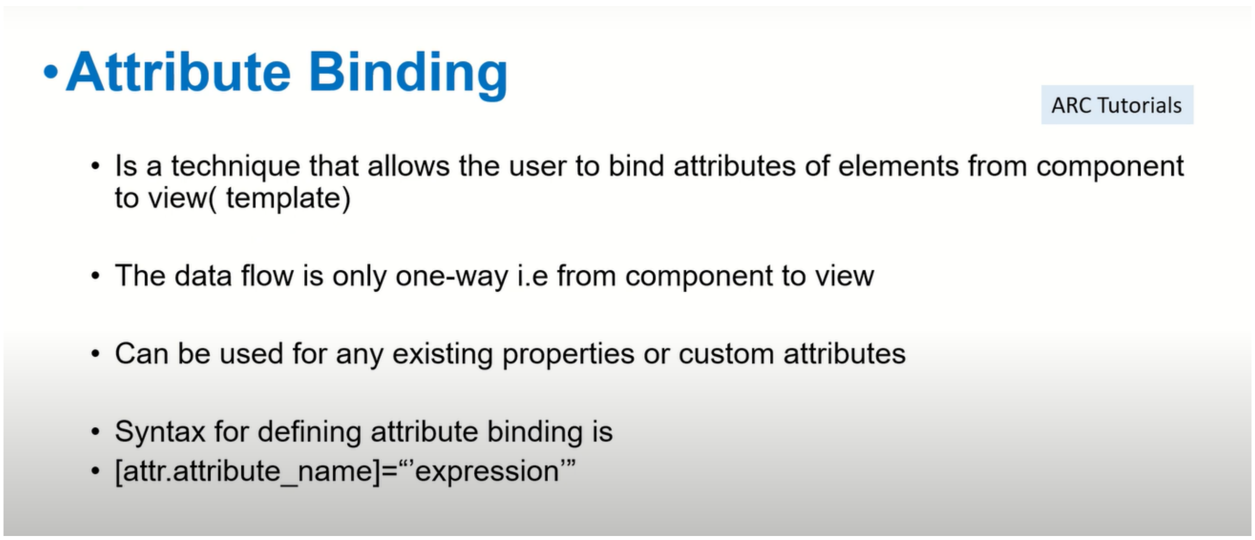
[ngClass]="expression" // property binding

[attr.className]="c1" // attribute binding

-> some attributes are not natively supported for elements

-> [colspan]="colVal" // error
-> [attr.colspan]="colVal"

-> Angular encourages to use property binding
-> attrbute binding




```
<table style="background-color: #ddd">
    <tr>
        <td>ContactId</td>
        <td>Contact Name</td>
        <td>Contact email</td>
    </tr>
    <tr>
        <td [attr.colspan]="colVal">ContactId</td>
        <td>Contact email</td>
    </tr>
</table>
```

10 Tutorial #28 - Event Binding

• Event Binding

ARC Tutorials

- Is a technique that allows the user to bind events of elements from view/template to component
- The data flow is only one-way i.e from view to component
- Can be used for all available events
- Syntax for defining attribute binding is
- <button (event_name) =“function()”> Example </button>

Tutorial #29 - Two Way Data Binding

Episode #29 - Two way Data Binding

1. Its a technique which helps us send data flow from template to component and vice versa

2. Data from Component -> To template -> To component -> To Template

3. Two way data binding is a combination
   -> Alternative of writing ngModel
   -> Property Binding and Event Binding on the same element

   E.g
   ```
   <input [value]="data1" (input)="$event.target.value" />
   ```

4. Angular provides a built-in directive called "ngModel"
   - using ngModel - it will handle both Property binding and event binding on an element

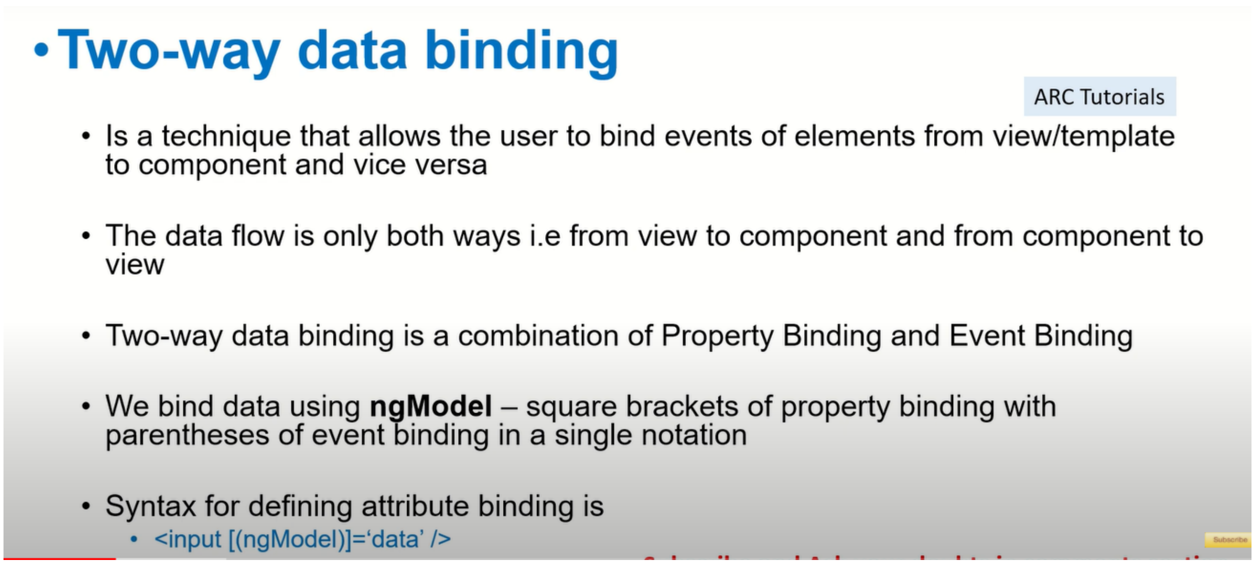
5. <input [(ngModel)]="username" />

   - Banana Syntax -> :)
   - the name of the ngModel should be declared in the component class
     - if you don't declare this variable - it will give you error
       -> Error => property does not exist on AppComponent

6. Very very important error

"Can't bind to ngModel since it isn't a known property of input"

- you have not imported FormsModule in our AppModule
-> How to fix it?
  Import FormsModule in our AppModule



Tutorial #30 - Pipes

• Pipes

- Pipes are used to transform the data
- Pipes will take data input and convert/transform into a desired format
- Pipes are written using the pipe operator (|)
- We can apply pipes to any view/template and to any data inputs

• Types of Pipes

- **Built in Pipes**
  - Lowercase
  - Uppercase
  - Currency
  - Date
- **Parametrized Pipes**
  - We can pass one or more parameters to pipes
- **Chaining Pipes**
  - We can connect multiple pipes to a data input
- **Custom Pipes**
  - We can create our own custom pipes for various data formatting

Tutorial #31 - Built-In Pipes

Episode #31 - Built-In Pipes

1. Pipe is used to transform the input data into ouput desired format

2. Built in pipes - we use them in templates

3. We can use multiple pipes in the template on elements

4. Syntax is
   <div> {{ <input_data> | <name_of_pipe> }} </div>

4. Built-In Pipes which are readily available for us to use

   lowercase
      -> saving usernames/email_address
         -> lowercase
   uppercase
      -> currency
      -> Airport Codes

   date
      -> by default Mon, dd, yyyy

   json
      -> I use this specially for debug purpose
      -> showing <code>

   currency
      -> by default uses dollar symbol $

   percent
      -> by Default is rounding to near by integer
      -> multiple by 100 and add % symbol

• Built-In Pipes

- Lowercase
- Uppercase
- Percent
- Currency
- Date
- JSON

```
-->
<h3>Built-In Pipes</h3>
<div> {{ lowerCaseExample | lowercase }} </div>

<div> {{ upperCaseExample | uppercase }} </div>

<div> {{ dateExample | date }} </div>

<div> {{ jsonExample | json }} </div>

<div> {{ currencyExample | currency }} </div>

<div> {{ percentExample | percent }} </div>
```

Tutorial #32 - Parametrized Pipes

Episode #32 - Parameterizing Pipes

1. Pipes can accept 1 or more parameters
   -> conditions/filters/specific data based on which data will be transformed

2. Some of the built in pipes which accept parameters are:

   - currency
   - date
     'MMM-dd-yyyy'
     'short'

'long'
- percent
  | percent : <minimumBeforeDecial>.<miniumDecialNumbers>-<maxDecimalNumbers>

3. Syntax for writing parameter pipes
   {{ <data_input> | <pipe_name> : <parameter> }}

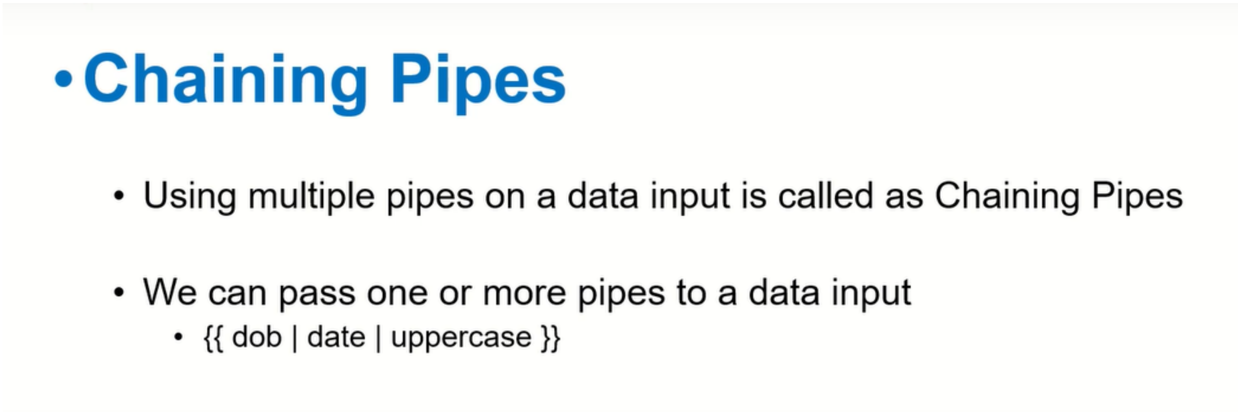
• Parametrized Pipes

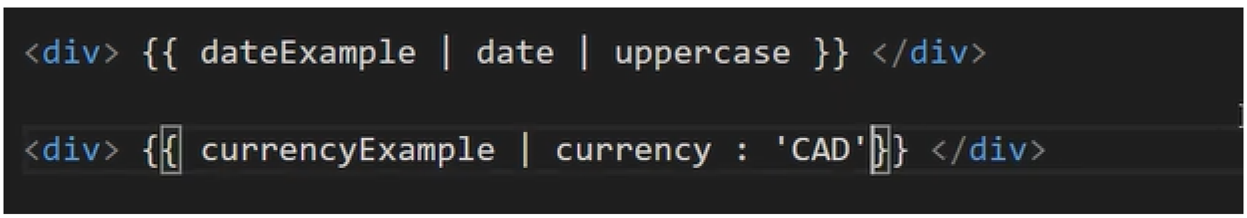
ARC Tutorials

- We can pass one or more parameters to pipes
- We pass parameters using the colon symbol (:)
  - Currency
    - Currency symbol
    - Currency Code
    - Currency Digit variations
- Date
  - 'short': equivalent to 'M/d/yy, h:mm a' (6/15/15, 9:03 AM).
  - 'medium': equivalent to 'MMM d, y, h:mm:ss a' (Jun 15, 2015, 9:03:01 AM).
  - 'long': equivalent to 'MMMM d, y, h:mm:ss a z' (June 15, 2015 at 9:03:01 AM GMT+1).
  - 'full': equivalent to 'EEEE, MMMM d, y, h:mm:ss a zzzz' (Monday, June 15, 2015 at 9:03:01 AM GMT+01:00).
  - 'shortDate': equivalent to 'M/d/yy' (6/15/15).
  - 'mediumDate': equivalent to 'MMM d, y' (Jun 15, 2015).
  - 'longDate': equivalent to 'MMMM d, y' (June 15, 2015).
  - 'fullDate': equivalent to 'EEEE, MMMM d, y' (Monday, June 15, 2015).
  - 'shortTime': equivalent to 'h:mm a' (9:03 AM).
  - 'mediumTime': equivalent to 'h:mm:ss a' (9:03:01 AM).
  - 'longTime': equivalent to 'h:mm:ss a z' (9:03:01 AM GMT+1).
  - 'fullTime': equivalent to 'h:mm:ss a zzzz' (9:03:01 AM GMT+01:00).

```
<div> {{ currencyExample | currency : 'CAD'}} </div>

<div> {{ dateExample | date : 'MMM-dd-yy'}} </div>

<div> {{ percentExample | percent : '3.1-4'}} </div>
```

Tutorial #33 - Chaining Pipes





Tutorial #34 - Angular Router

Home -> http://myapplication.com/ -> Default Route

Profile -> http://myapplication.com/profile -> Component Routing

Search -> http://myapplication.com/search?user=abc -> Query Params

Tasks -> http://myapplication.com/tasks/10/category/pending -> URL Segments

Users -> http://myapplication.com/users -> Module

view-user -> http://myapplication.com/users/view/10 -> Child Routes
edit-user -> http://myapplication.com/users/edit/10 -> Child Routes
add-user -> http://myapplication.com/users/add -> Child Routes
manage-user -> http://myapplication.com/users/manage -> Child Routes

PageNotFound -> http://myapplication.com/pageNotFound -> 404 error -> No matching routes

• Angular Router

ARC Tutorials

- Routing is a mechanism used by Angular framework to manage the "paths" and "routes" of our Angular applications
- Routing strategy helps in navigation between various views in our Angular application
- Angular framework comes with "Router" Module which has everything we need to design, develop and implement routes and navigation links
- Router is a singleton – which means there is ONLY one instance of the router in our Angular application

• Routing in Angular

ARC Tutorials

- Angular Router is the official Router module which is written and maintained by core Angular team
- The Router module is found in the package @angular/router
- We need to setup Router array – every time a request is made, the router will search in the list of array and find the most relevant match.
- Router has states- which helps us get important information about the current state and data related to routes
- All batteries included for Router

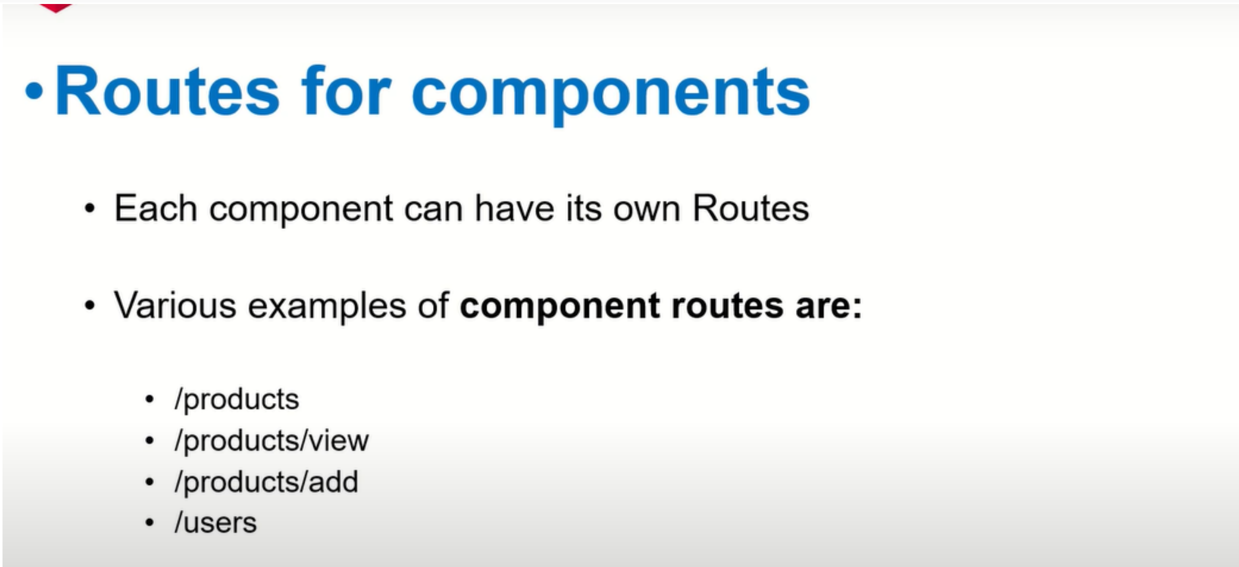
• Routing in Angular

- We can handle various types of routes in Angular app
  - Routes for components
  - Getting Query Params from routes
  - Getting the URL segments
  - Loading child routes for a module
  - Lazy Loading
  - Handling wild card routes
  - Handling default routes
  - Handling 404 route

- All batteries included for Router

```
Home -> http://myapplication.com/ -> Default Route

Profile -> http://myapplication.com/profile -> Component Routing

Search -> http://myapplication.com/search?user=abc -> Query Params

Tasks -> http://myapplication.com/tasks/10/category/pending -> URL Segments

Users -> http://myapplication.com/users -> Module

    view-user -> http://myapplication.com/users/view/10 -> Child Routes
    edit-user -> http://myapplication.com/users/edit/10  -> Child Routes
    add-user -> http://myapplication.com/users/add  -> Child Routes
    manage-user -> http://myapplication.com/users/manage  -> Child Routes

PageNotFound -> http://myapplication.com/pageNotFound -> 404 error -> No matching routes
```

Tutorial #35 - Component Routing



Tutorial #36 - Router Outlet

Episode #36 - Router Outlet

1. Router outlet is a built-in directive

2. Every Angular app should have "at least" 1 router outlet
   -> primary router outlet

3. By default - the router outlet is defined in app.component.html file

4. Router outlet will match the matching routes for the components
   -> takes its output
      -> inside inside the page

5. Multiple router outlets in application
   -> We can have more than 1 router outlet

Common Mistakes
   -> you don't router-outlet
      -> you wont see the output

   -> Best practice is to leave router-outlet empty

• Router Outlet

ARC Tutorials

- The Router-Outlet is a directive that's available from the router library where the Router inserts the component that gets matched based on the current browser's URL.
- You can add multiple outlets in your Angular application which enables you to implement advanced routing scenarios.
- By default there is always one router outlet defined – in app.component.html

Tutorial #37 - Multiple Router Outlets

Episode #37 - Multiple Router Outlets

1. We can have multiple router Outlets

2. by default there is always/"atleast" 1 router outlet in app.component.html file

3. When we don't provide any name for router-outlet it becomes primary

4. There should be only 1 primary

5. We can define multiple router outlets by giving name to them

6. That's why we call them "named" router outlets

7. we can give any name we want - give meaningful names

8. In routing module - if you don't define outlet - it means its primary

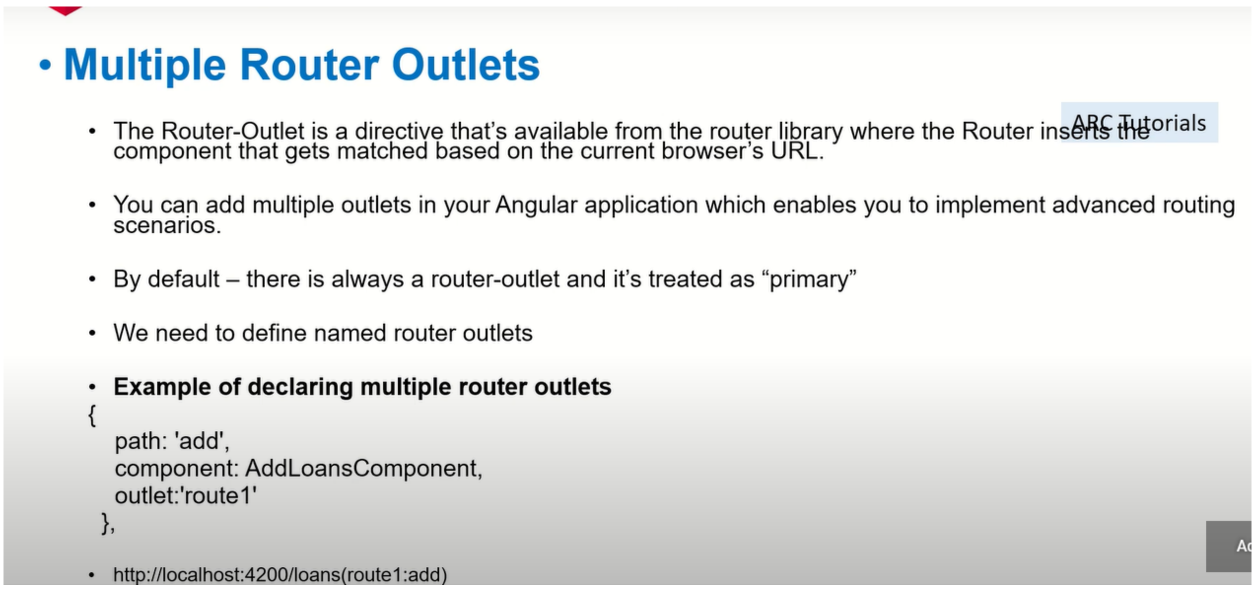
9. It will NOT show if you directly access it in the URL
----

10. Synatx should be like this
   http://localhost:4200/<primary-route>(<routerOutletName> : <secondaryPath>)

11. Why are using this?
   - Avoid this use case in applications?
   - You can inject components
      ->
12. URL is not user friendly

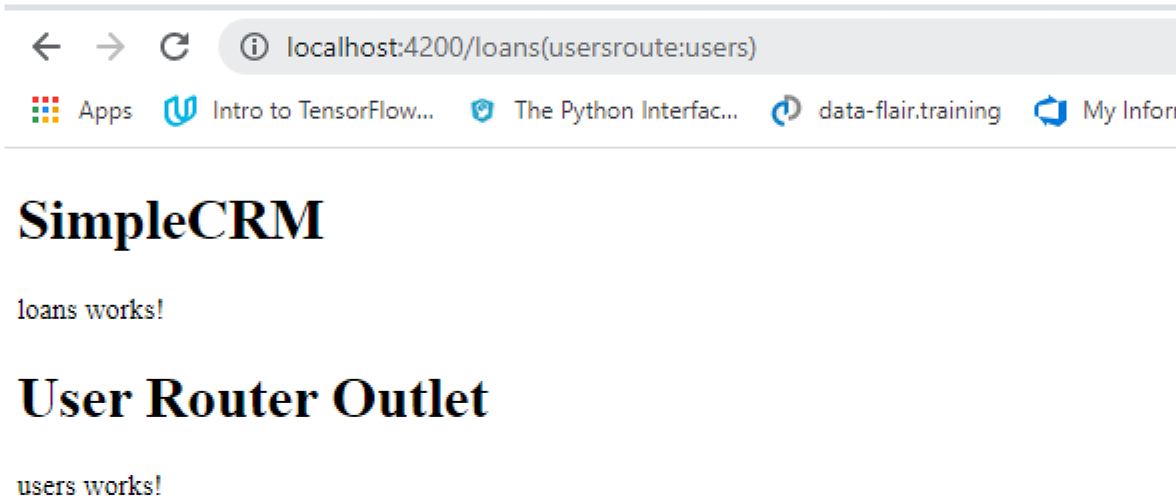
- bookmarable URL

13. I have not personally seen this used a lot
   -> It's not used very much




```
const routes: Routes = [
  {
    path: 'loans',
    component: LoansComponent
  },
  {
    path: 'users',
    component: UsersComponent,
    outlet : "usersroute",
  },

];
```

```
<h1>{{title}}</h1>
<router-outlet></router-outlet>
<router-outlet name="loans"></router-outlet>
<h1>User Router Outlet</h1>
<router-outlet name="usersroute"></router-outlet>
```



Tutorial #38 - Routing Strategy

Episode #38 - Routing Strategy

1. Routing behaviour of the applications URLs

2. Angular provies 2 routing strategies

- PathLocationStrategy
    - Default routing strategy for Angular apps
    - HTML 5 push state URL
    - Examples
        - http://myapp.com/dashboard
        - http://myapp.com/user/10
        - http://myapp.com/user/10/photos
        - http://myapp.com/search?query=abc&state=ka&city=bengaluru

- HashLocationStrategy
    - URL segments/patterns
    - URLs will have hash in the URLs
    - Examples
        - http://myapp.com/#/dashboard
        - http://myapp.com/#/user/10
        - http://myapp.com/#/user/10/photos
        - http://myapp.com/#/search?query=abc&state=ka&city=bengaluru

3. Hands-on examples for PathLocationStrategy
    - Default behaviour of Angular apps

4. Hands-on examples for HashLocationStrategy
   - We need to import HashLocationStrategy from @angular/core
   - Add it to Providers array
   - Angular will start loading our URLs using #

5. Why do we need 2 different types of routing?

   Angular is a SPA(single page app)
      - index.html

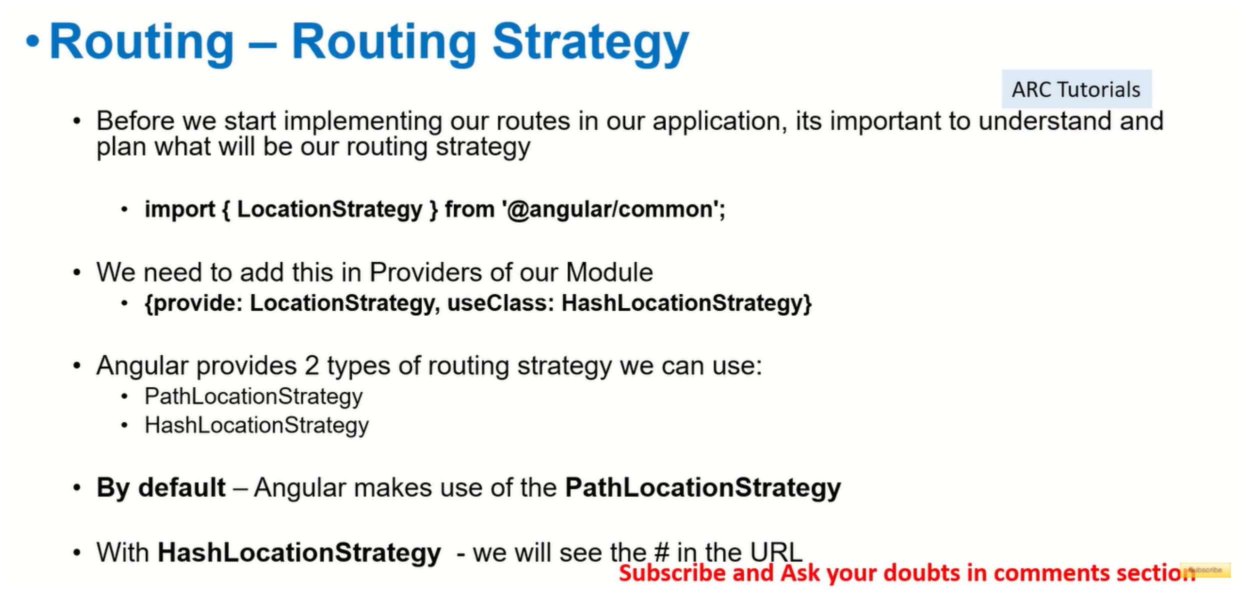
   Cloud vendors
      AWS
      GCP
      Azure
      Hosting Provider (Bluehost, Siteground, DigitalOcean)

   - /#/loans/add -> Route
   index.html/#/loans/add

6. Which one you should use when?

   Really there is no difference affect your application

   PathLocationStrategy
      -> Clean URLs
      -> Simple
      -> Bookmarbale
      -> Easy to Remember



[Tutorial #39 - Base Href]

Episode #39 - Base Href

1. Base HREF is mandatory for all Angular apps

2. Base HRef is present in your index.html file

3. The project is pointing to the "root" directory/folder of your server which is running at 4200 port

4. When your deploying your Angular apps to server

   -> root folder

   -> http://myapp.com/app1/

      http://myapp.com

5. It decides where you want to deploy your app
   - thats why its extremely important

6. <base href="/">

• Routing – Base Href

ARC Tutorials

- Every Angular application has MANDATORY base href
- Angular application is a SPA (Single Page Architecture) which means there will be only one HTML file
- The default base href is set to “/” the root folder
- The Base HREF is present in index.html file for all Angular applications

```html
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>ARCTutorial</title>
  <base href="/">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body>
  <app-root></app-root>
</body>
</html>
```

• Routing – Base Href

- Wrong configuration leads to pointing to wrong folder root path
- Setting the base href using the command line –base-href=
- Syntax: <base href=“/” >

Tutorial #40 - Router Module

Episode #40- - Routing Module / App Routing Module / Router Module

1. Its a single module and placeholder where all our routes are configured for that particular module

2. Each module can have its own routes

3. During the angular app installation
   - we get an option - Do you want to have routing in your application?
      - it will automatically create the app-routing module file for us

4. ng g module app-routing --flat --module=app

• Routing Module

ARC Tutorials

- Routing Module is a placeholder for configuring all routes and navigations in one module
- Best practice is to have all routes configured in one place
- Easy to maintain and debug
- We can generate the app routing module using the CLI
  - **ng generate module app-routing --flat –module=app**

• Router Module

- We need to import modules from the package
- import { Routes, RouterModule } from '@angular/router';
- We need to configure route path array in the file
  - const routes: Routes = [];
- Then we need to define our module
  - @NgModule({
  - imports: [RouterModule.forRoot(routes)],
  - exports: [RouterModule]
  - })
- We need to export the module
  - export class AppRoutingModule { }
- Import the module in the AppModule file

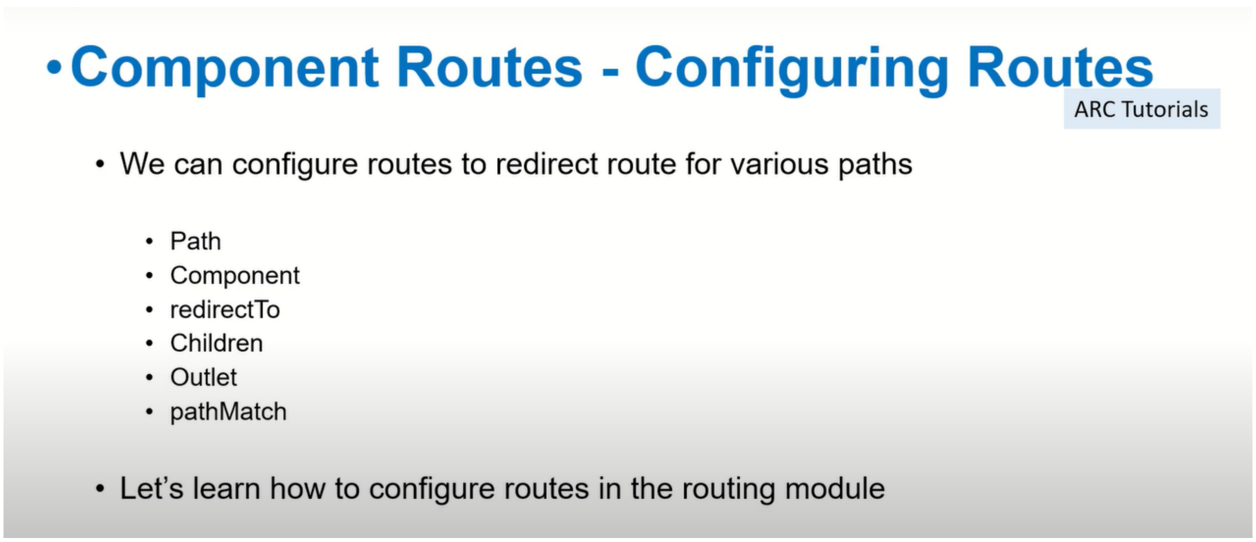
Tutorial #41 - Configure Component Routes

Episode #41 - Configuring Component Routes Options

1. There are various options that we can configure in Component Routes

2. Some of the ones that we have seem in previous/earlier tutorials ar earlier are

    2.1 path
    2.2 component
    2.3 Router Outlet Tutorial - Please check the playlist
        -> outlet
    2.4 children
    2.5 redirectTo ->
    2.6 pathMatch -> Will cover in coming episodes



```
const routes : Routes = [
  { path: 'loans',
    children: [

      {
        path: 'edit-loan', component: AddLoansComponent
      },
      {
        path: 'delete-loan', component: AddLoansComponent
      }
    ]
  },
  { path: 'customers',
    children: [
```

```
const routes : Routes = [
  {
    path: 'add-new-loan',
    redirectTo: 'add-loan'
  },
  {
    path: 'add-loan',
    component: AddLoansComponent,
  }
```

Tutorial #42 - Parametrized Routes

Episode #42 - Parameterized Routes - Dynamic Routes

1. We can send dynamic data or parameters

2. URLs will look something like this

   http://localhost.com/user/10 -> get the user with Id as 10

   http://localhost.com/search/ka/bangalore -> state and city

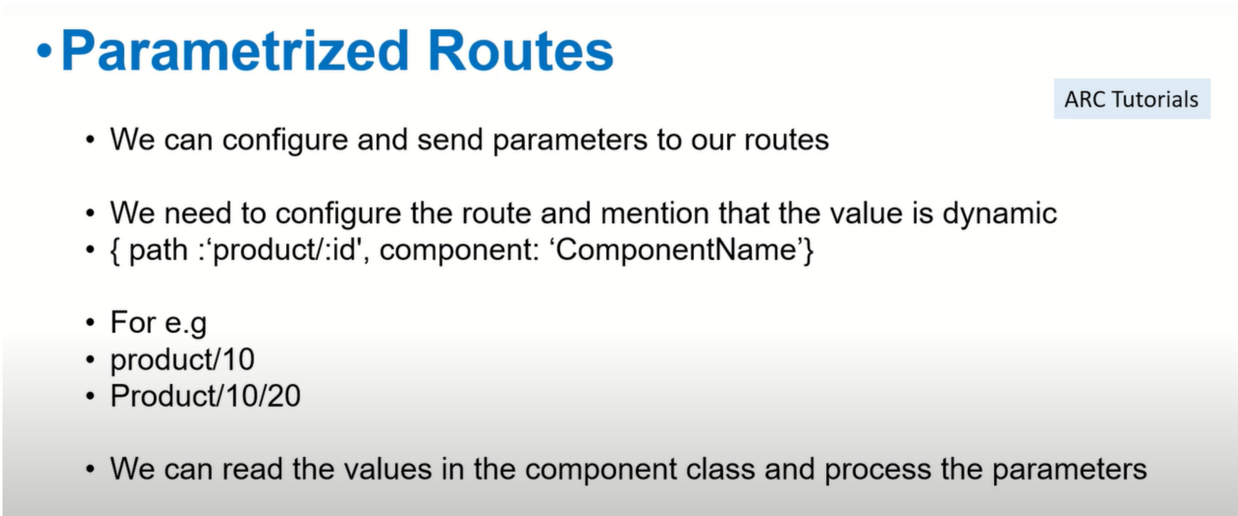
   http://localhost.com/user/10/photos/34 -> user id = 10 and photo id = 34

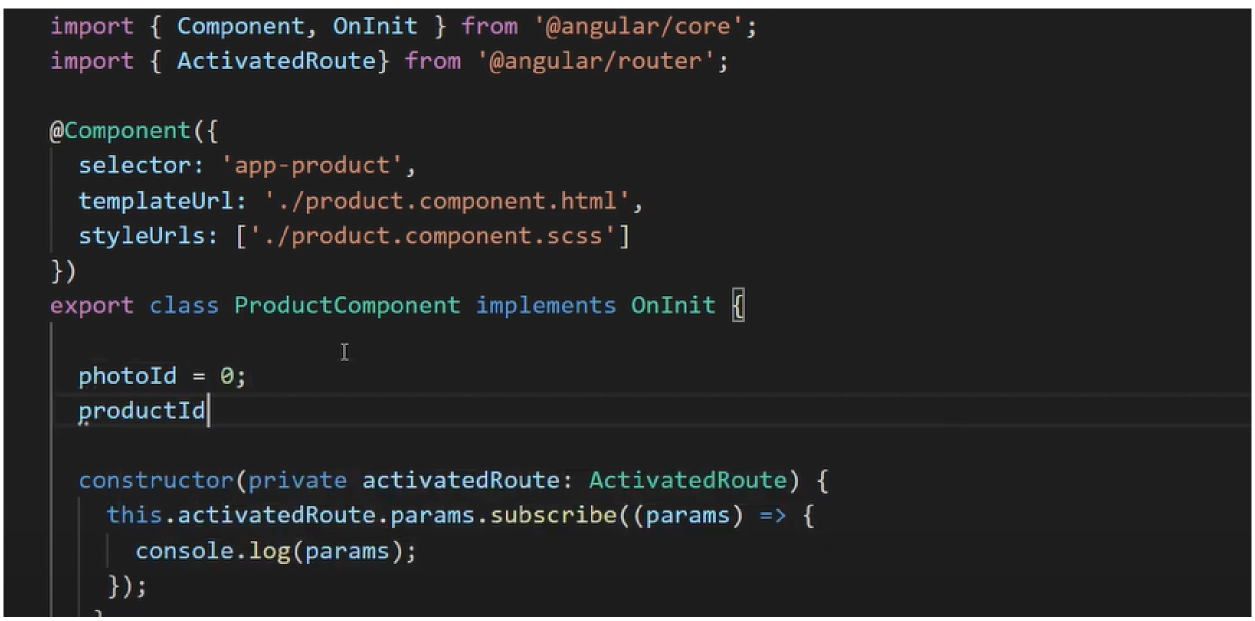
3. While writing dynamic URLs/Params - make sure you write :(colon) for dynamic data

4. Import the ActivatedRoute class

5. Create an object in constructor ->

6. We can create any number of dynamic params in our URLs





```
import { Component, OnInit } from '@angular/core';
import { ActivatedRoute} from '@angular/router';

@Component({
  selector: 'app-product',
  templateUrl: './product.component.html',
  styleUrls: ['./product.component.scss']
})
export class ProductComponent implements OnInit {

  photoId = 0;
  productId

  constructor(private activatedRoute: ActivatedRoute) {
    this.activatedRoute.params.subscribe((params) => {
      console.log(params);
    });
```

Tutorial #43 - RouterLink

Episode #43 - Router Link

1. We can have any number of router links in the template

2. Router Links can be static or can be dynamic in nature

3. Common Mistakes
   - Not putting strings in single qoute
   - Not passing dyanmic data correctly

4. Static Router Link -> <a [RouterLink]="'/user'">

5. Dynamic Router Link

6. We DO NOT have to put "/" in variables in routerLink

7. Router Link Query Params -> we will cover along with Query params in routes

• Router Link

ARC Tutorials

- When applied to an element in a template, makes that element a link that initiates navigation to a route.

- Navigation opens one or more routed components in one or more <router-outlet> locations on the page.

- For e.g
- <a [routerLink]="['/user/bob']"> Some link

```
<a [routerLink]="'/user'">User List</a>
<table>
    <th>Client Id</th>
    <th>Firstname</th>
    <th>Lastname</th>
    <tr *ngFor="let client of clientList">
        <td>{{ client.clientId }}</td>
        <td>{{ client.firstName }}</td>
        <td>{{ client.lastName }}</td>
        <td><a [routerLink]="['/user', client.clientId, 'edit']">Edit</a> | <a [rou
    </tr>
</table>
```

clients works!

User List

Client Id	Firstname	Lastname	
10	Raj	Srini	Edit | Delete
11	john	Mike	Edit | Delete
12	Moon	Amanuel	Edit | Delete
13	Cherry	Ben	Edit | Delete
14	Berry	Kumar	Edit | Delete
15	Steve	Kumar	Edit | Delete

/edit/13/users/16

Tutorial #44 - Redirect Routes

Episode #44 - Routes Redirect

1. By default the root level route is ''

2. redirectTo and specify which route it has to go

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

• Redirecting Routes

ARC Tutorials

- When we want a route to be redirected to another route – we will implement the redirectTo in our routes array

- The syntax to define the same is given below
  - { path :'', redirectTo: 'home', pathMatch: 'full'},

- The empty path indicates that it's the **default route** of the application

- The empty path also requires us to mention that **pathMatch** should be **"full"**

- Let's learn how to redirect route in the routing module

```
  {
    path: '',
    redirectTo: 'leads',
    pathMatch: 'full'
  },
  {
    path: 'leads',
    component: LeadsGridComponent
  },
];
```

Tutorial #45 - Query Params

Episode #45 - Query Params

1. We can send data from Form ->

2. We can have data from click ->

Basically -> URL -> http://localhost.com/search?key=10&state=ka&city=bangalore

3. Query Params -> visible in the URL

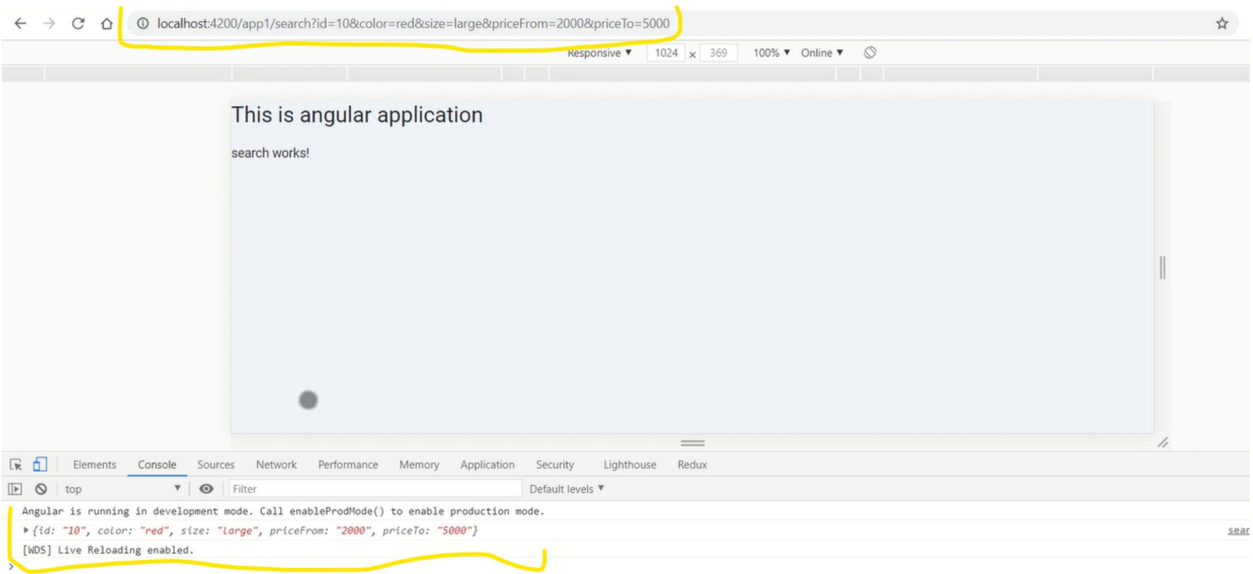
4. Most used for querying, searching or filtering data etc

facebook.com/search?page=10&pagesize=20

• Query Params

ARC Tutorials

- We can configure and send query parameters to our routes
- Search?keyword=toys&country=usa
- We can read the values in the component class and process the parameters




```
color = '';
priceTo = 0;
priceFrom = 0;
size = 'M';

constructor(private activatedRoute: ActivatedRoute) {

  this.activatedRoute.queryParams.subscribe(params => {
    this.color = params.color;
    this.color = params.color;
    this.color = params.color;
    this.color = params.color;
```

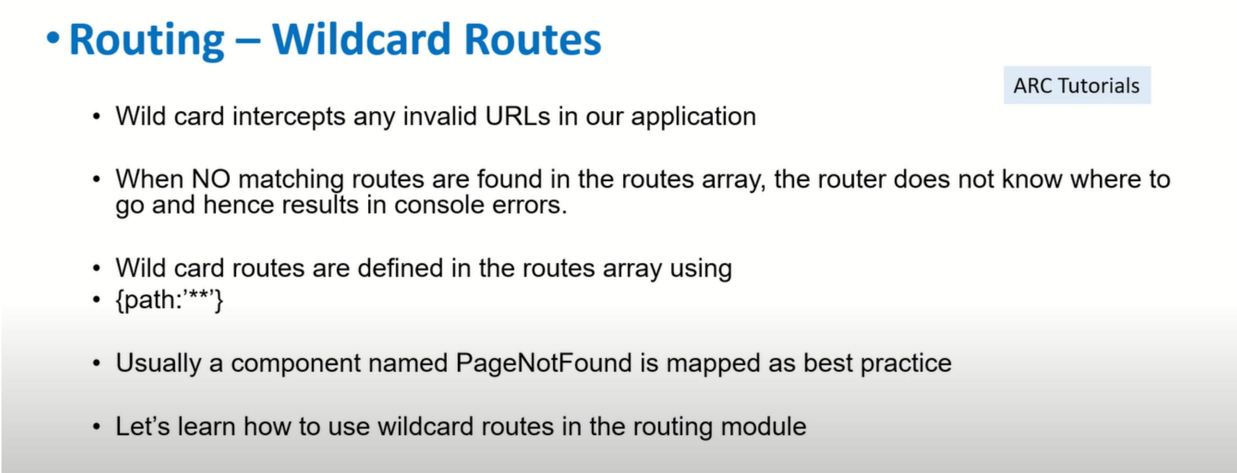
[Tutorial #46 - Wildcard Routes Link]

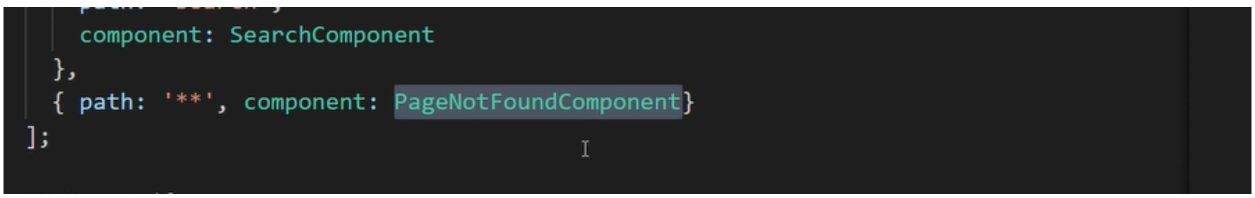
Episode #46 Wildcard Routes

1. Any unmatched route will be intercepted by Wild card route

2. This has to be the last route in your configuration

3. we define by saying the path to match "**"





whildCard should always be last

[Tutorial #47 - Lazy Loading]()

Episode #47 - Lazy Loading

1. Any angular application is made up of multiple Modules

    Loan Management System
      - Loans
      - Customers
      - Payments
      - Invoices
      - Reports
      - Authentication
      - Authroization
      - Downloads
      - Admin

2. Angular by default will load all modules at start
    - Login

3. Loading all modules initally - wheather required or not
    - makes your application slow performace wise
    - also its a bad idea to expose modules which user is NOT going to use

- user should not see/use

4. Lazy Loading comes into Picture

    -> Initially we will load only modules which are mandatory
    -> Rest we will serve as "requested"

    -> We will routes for each module

        - Payments

        /payments
            - then only we will load this module

            1. it will performance app
            2. we can verify if the user has access to this module

5. lazy Loading will help in keeping your builds smaller
    -> ng build / compile application to deploy
        -> files
            -> the size of those files will be smaller
            -> it will load fast
            -> it will respond better

6. If you are coming from previous version of Angular 8
    -> the syntax has changed
        -> please use the expetcs a function

7. The modules generated using the Angular CLI - for lazy Loading
    -> There will be NO entry in AppModule
    -> Hence, it will not be loaded initially

8. ng g module <module_name> --route <module_route> --module app.module

E.g ng g module payments --route payments --module app.module

9. The above command will generate the following
    - A routing file for the module
    - A module file
    - A component
        - html
        - css/scss
        - spec
        - class

- UPDATE the app routing module

10. /payments
    -> Module on demand and its children - if needed
    /payments/success

• Routing – Lazy Loading

ARC Tutorials

- By default, **NgModules** are eagerly loaded, which means that as soon as the app loads, so do all the **NgModules**, whether or not they are immediately necessary.
- For large apps with lots of routes, consider lazy loading—a design pattern that loads NgModules as needed.
- Lazy loading helps keep initial bundle sizes smaller, which in turn helps decrease load times.
- From Angular 8, **loadChildren** expects a function that uses the dynamic import syntax to import your lazy-loaded module only when it's needed

- With lazy loaded modules in Angular, it's easy to have features loaded only when the user navigates to their routes for the first time.
- This can be a huge help for your app's performance and reducing the initial bundle size. Plus, it's pretty straightforward to setup!
- When the application grows in size, we should always modularize the application into individual
- Load the modules on-demand (we can verify them in the console)

```
{ path: 'customer', loadChildren: () => import('./customers/customers.module').then(m =
{ path: 'payments', loadChildren: () => import('./payments/payments.module').then(m =>
{ path: '**', component: PageNotFoundComponent}
```

```
() => import('./customers/customers.module').then(m => m.CustomersModule) }
() => import('./payments/payments.module').then(m => m.PaymentsModule) },
```

Tutorial #48 - Route Guards

Episode # 48 - Route Guards

1. Route Guards helps us secure our routes and screens

2. E.g

```
User -> Route Guard                  ->   /Admin
     -> Custom Logic
        -> True                 ->   Admin
           -> can access the route

        -> False
           -> custom logic on failure condition -> home
```

3. Generate Route Guard
   ng g guard <guard_name>

4. Route Guards have something called "interfaces"
   - canActivate -> can a user access a route
   - canActivateChild -> can user access child routes of a parent route
   - canDecativate -> check if user can exit the route
   - canLoad -> Can a lazy loaded module be loaded
   - resolve -> route data retrival before route activating

5. I will cover all of these in detail in coming episodes
   - quick examples
   - use cases

6. We can implement more than 1 guards in our application

• Routing – Route Guards

ARC Tutorials

- Use route guards to prevent users from navigating to parts of an app without authorization
- Route Guards are used to secure the route paths
- In most cases, the routes and screens are protected behind a good authentication system
- The route guard resolves to true or false based on custom logic and functionality
- We can generate any number of guards based on our application requirements

• Routing – Route Guards

ARC Tutorials

- To generate the route guard we can make use of Angular CLI
  - ng generate guard <guard-name>
- Inject the guard in our module under providers
- There are various types of route guards available
  - **CanActivate** – Checks to see if a user can visit a route
  - **CanActivateChild** - Checks to see if a user can visit a routes children
  - **CanLoad** - Checks to see if a user can route to a module that lazy loaded
  - **CanDeactivate** - Checks to see if a user can exit a route
  - **Resolve** - Performs route data retrieval before route activation
- The route guard resolves to true or false based on custom logic and functionality

```
import { Injectable } from '@angular/core';
import { CanActivate, ActivatedRouteSnapshot, RouterStateSnapshot, UrlTree }
import { Observable } from 'rxjs';

@Injectable({
  providedIn: 'root'
})
export class AuthGuard implements CanActivate {
  canActivate(
    route: ActivatedRouteSnapshot,
    state: RouterStateSnapshot): Observable<boolean | UrlTree> | Promise<bool
    return true;
  }

}

```

```
{ path: 'product/:productId/photos/:photoId', component: ProductComponent},
{
  path: 'clients',
  component: ClientsComponent,
  canActivate: [AuthGuard]
},
{
```

```
export class AuthGuard implements CanActivate {
  userToken = false;

  canActivate(
    route: ActivatedRouteSnapshot,
    state: RouterStateSnapshot): Observable<boolean | UrlTree> | Promise<bool

      // Call an HTTP call to Backend API and get Auth Token for user
      this.userToken = false;
      if(this.userToken){
        return true;
      }
      else {
        return false;
      }
  }

}
```

Tutorial #49 - Route Guards - CanActivate

Episode #49 - Route Guards - CanActivate

1. ng g guard <guard_name>

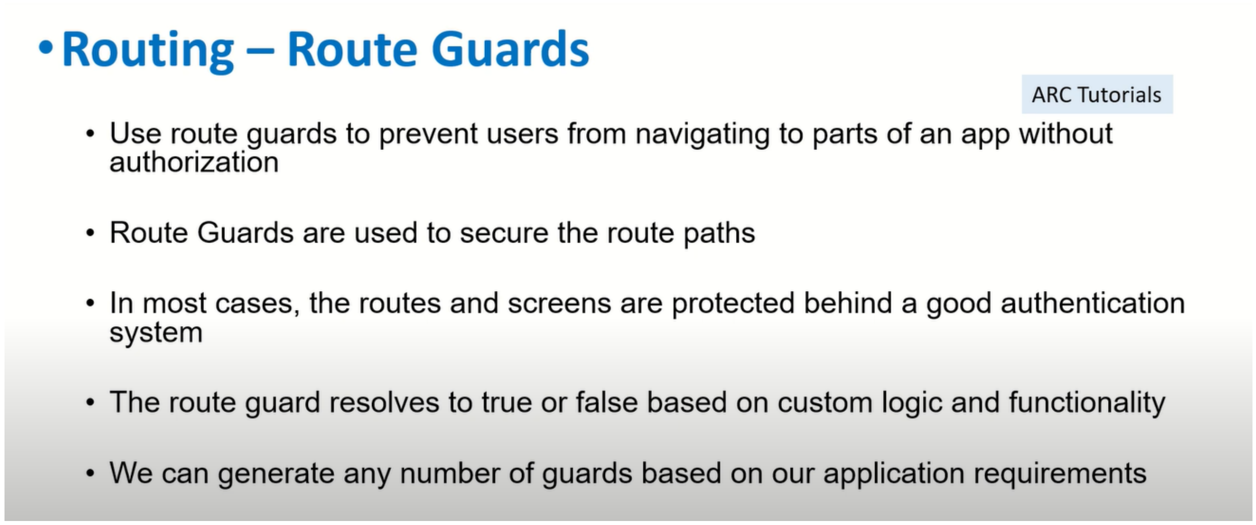
2. choose the option CanActivate

3. In the routing module -> we will use option canActivate
   -> it will resolve to true or false

   -> true means -> user can access the route
   -> false means -> user cannot access the route

4. We can use any number of route guards on canActivate
   -> Its an array
   -> all have to resolve to true

5. Use cases
   1. Check if user is loggedIn
   2. Check if user can Edit the product/order/details/profile
   3. Check if the user is an Admin



• Routing – Route Guards

ARC Tutorials

- To generate the route guard we can make use of Angular CLI
  - ng generate guard <guard-name>

- Inject the guard in our module under providers

- There are various types of route guards available
  - **CanActivate** – Checks to see if a user can visit a route
  - **CanActivateChild** - Checks to see if a user can visit a routes children
  - **CanLoad** - Checks to see if a user can route to a module that lazy loaded
  - **CanDeactivate** - Checks to see if a user can exit a route
  - **Resolve** - Performs route data retrieval before route activation

- The route guard resolves to true or false based on custom logic and functionality

• CanActivate

ARC Tutorials

- A **CanActivate** guard is useful when we want to check on something before a component gets used.

Tutorial #50 - Route Guards - CanActivateChild

• CanActivateChild

ARC Tutorials

- The **CanActivateChild** guard works similarly to the CanActivate guard, but the difference is its run before each child route is activated

TS app-routing.module.ts × | TS admin-access.guard.ts | TS super-admin.guard.ts | TS load-m

simpleCRM > src > app > TS app-routing.module.ts > routes > children > children

```
      component: LeadsGridComponent
    },
    {
      path: 'admin',
      canActivate: [SuperAdminGuard],
      children: [
        {
          path:'',
          component: AdminComponent
        },
        {
          path:'',
          canActivateChild: [AdminAccessGuard],
          children: [
            { path: 'manage', component: AdminManageComponent },
            { path: 'edit', component: AdminEditComponent },
            { path: 'delete', component: AdminDeleteComponent },
          ]
        }
      ]
    },
```

TS app-routing.module.ts | TS admin-access.guard.ts × | TS super-admin.guard.ts | TS load-m

simpleCRM > src > app > TS admin-access.guard.ts > AdminAccessGuard

```
import { Injectable } from '@angular/core';
import { CanActivateChild, ActivatedRouteSnapshot, RouterStateSnapshot, UrlTree
import { Observable } from 'rxjs';

@Injectable({
  providedIn: 'root'
})
export class AdminAccessGuard implements CanActivateChild {
  canActivateChild(
    childRoute: ActivatedRouteSnapshot,
    state: RouterStateSnapshot): Observable<boolean | UrlTree> | Promise<boolea
    return false;
  }

}
```

```
    component: LeadsGridComponent
  },
  {
    path: 'admin',
    canActivate: [SuperAdminGuard],
    children: [
      {
        path:'',
        component: AdminComponent
      },
      {
        path:'',
        canActivateChild: [AdminAccessGuard],
        children: [
          { path: 'manage', component: AdminManageComponent },
          { path: 'edit', component: AdminEditComponent },
          { path: 'delete', component: AdminDeleteComponent },
        ]
      }
    ]
  },
```

Tutorial #51 - Route Guards - CanLoad

- **Generate a Lazy Loading Module**
  - ng generate module customers --route customers --module app.module

ARC Tutorials

- **canLoad**
  - This protects the route completely. Such as lazy loading the module and also protects all the routes associated with that module

```
    canLoad: [LoadModuleGuard],
    loadChildren: () => import('./customers/customers.module').then(m => m.Cust
  },
  { path: 'payments', loadChildren: () => import('./payments/payments.module').
  { path: 'preferences',
    canLoad: [PreferencesCheckGuard],
    loadChildren: () => import('./preferences/preferences.module').then(m => m.
]
@NgModule({
  declarations: [],
```

```
  providedIn: 'root'
})
export class PreferencesCheckGuard implements CanLoad {
  canLoad(
    route: Route,
    segments: UrlSegment[]): Observable<boolean | UrlTree> | Promise<boolean |
    return true;
  }
}
```

Tutorial #52 - Route Guards - CanDeactivate

• canDeactivate

ARC Tutorials

- When we want to make sure that user can deactivate a particular route – we will use canDeactivate
- Interface that a class can implement to be a guard deciding if a route can be deactivated.
- If all guards return true, navigation continues. If any guard returns false, navigation is cancelled

• Routing – canDeactivate

```
canDeactivate(component: SearchComponent){
  console.log(component.isDirty);

  if(!component.isDirty)
  {
   return window.confirm("You have some unsaved changes?")
  }
  return true;

}
```

[Tutorial #53 - Route Guards - Resolve]

• Resolve Guard

ARC Tutorials

- Resolve route guard allows us to provide data needed for a route
- If some data is "MANDATORY" for a component – try using the logic from ngOnit to Resolve
- Using the activatedRoute.snapshot.data we can access data and process it

```
ngOnInit(): void {
  console.log(this.ActivatedRoute.snapshot.data);
}
```

Tutorial #54 - Angular Forms

Angular : Forms

ARC Tutorials

- I have planned around **20 dedicated tutorials** with in-depth use cases and complex form structures for both template and reactive forms
- We will try and use as many as input elements possible to create the dynamic forms

• Forms - Introduction

ARC Tutorials

- Forms are a very integral and essential building blocks of "almost" apps
- Common form examples we can see are
- Login
- Forgot
- Register
- Checkout form
- Contact Us
- Forms allows us to gather information and data from users
- Good way to interact with the users and almost all websites will need forms in some or other way
- We can use any CSS framework of our choice – Bootstrap or Material Design

• Angular Support for Forms

- Two Data Binding
- Change Tracking
- Validations
- Error Handling
- Unit testing

• Types of Forms in Angular

- Static / Template Driven Forms
- Dynamic / Reactive Forms

- **Static or Template Driven Forms**
  - Easy to use
  - Template driven forms are simple and straight forward
  - All the validations, form elements are all defined in the template file
  - We will need to import **FormsModule** in app module to work with Template driven forms

ARC Tutorials

- **Dynamic or Reactive or Model Driven Forms**
  - All the form elements, user interactions and validations are handled in the component class
  - We will make use of Angular's built in **formGroup** and **formControl**
  - Can control better data binding
  - Exclusive define custom regular expression patterns of error handling
  - We will need to import **ReactiveFormsModule** in our app module
  - Very flexible and allows users to define, develop complex requirements of forms
  - More logic in the component class and less in HTML mark up itself

• Which is better – Template driven Forms or Reactive Forms?

- **Template Driven Forms**
  - If your application forms are simple straight forward
  - Fixed static form fields and elements
  - No complex validations or pattern matching
- **Reactive Forms**
  - If your application forms are complex
  - Uses multiple dynamic components
  - Advanced validation requirements
  - Dependent form elements
  - Dynamic form generation based on user preferences

Tutorial #55 - Angular Template Driven Forms

• Template Driven Forms - Introduction

ARC Tutorials

- Easy to use
- Template driven forms are simple and straight forward
- All the validations, form elements are all defined in the template file
- Forms are tracked automatically
- Tracked form data traverses via various states (pristine etc)
- Uses Two-Way Data Binding techniques to bind data
- Most of the code resides in template file
- Validations are mostly the default HTML5 validations
- Minimal component code as most of the code is in template file
- Unit testing will be a challenge

• Step by Step Process for Template Driven Forms

- **Step #1 – Import and Add in the FormsModule in the app.module.ts**
  - For template driven forms – FormsModule needs to be imported
  - If we do NOT import this – we will get error when doing two way data binding
  - Add the module into the array list of imports

- **Step #2 – Create the form in app.component.html**
  - **ngForm**
    - Form name as template variable using "#" for e.g #loginForm
  - **ngModel**
    - Every form field should have a "name" attribute and ngModel attached to it

why?

```html
<form #addCustomerForm="ngForm" (submit)="addCustomer(addCustomerFo
    <input type="text" [(ngModel)]="firstname" name="firstname">

    <input type="checkbox" [(ngModel)]="terms" name="terms">

    <button type="submit">Add Customer</button>
</form>

<div> {{ firstname }}</div>
```

```typescript
  constructor() { }

  ngOnInit(): void {
  }

  addCustomer(formValue: NgForm){
    console.log(formValue.value);
    // validations
    // Data Processing
    // Then call API to save this data
  }
}
```

Tutorial #56 - Angular Forms Validation

- **Add different Form Input Types**
  - Input type="text"
  - Input type="radio"
  - Input type="checkbox"
  - Input type="email"
  - Textarea
  - Select Drop Down

- **Adding Validations in Template Driven Forms**
  - Disable the form
    - Disable the form if the form is not valid
  - HTML5 validations
    - Required
    - minLength
    - maxLength
    - checked

Tutorial #57 - Angular Reset Forms

- **Reset Template Driven Forms**
  - Reset the form using
    - Reset method
    - Form.reset();

```
button class="btn btn-link" (click)="resetForm(addCustomerForm)
m>
```

```
  resetForm(formValue : NgForm){
    formValue.reset();
  }

}
```

```
  resetForm(formValue : NgForm){
    // formValue.reset();
    formValue.resetForm();

  }
```

[Tutorial #58 - Angular Set Form Value]()

• Set Form Value in Template Driven Forms

- Set the form using
  - Form.setValue()

```
loadValues(formValue: NgForm){
  let userDetails = {
    firstname: 'ARC',
    terms: false,
    customerType: 'Premium',
    description: 'This is SET Value in Forms'
  }

  formValue.setValue(userDetails);
}
```

Tutorial #59 - Reactive Forms Complete

• Reactive Forms - Introduction

ARC Tutorials

- Reactive Forms are a way to create Forms in Angular application
- What's different is how we implement, design and handle the form and the data
- All the form elements, user interactions and validations are handled in the component class
- We will make use of Angular's built in **formGroup** and **formControl**
- Using Reactive Forms we can control better data binding
- Exclusive define custom regular expression patterns of error handling
- We will need to import **ReactiveFormsModule** in our app module
- Very flexible and allows users to define, develop complex requirements of forms
- More logic in the component class and less in HTML mark up itself

- Angular maintains the state information of forms at all times
  - ng-touched
  - ng-untouched
  - ng-dirty
  - ng-pristine
  - ng-valid
  - ng-invalid

• Reactive Forms - Misconception

- Reactive Forms are tough
- Reactive forms are very complex
- Reactive forms are difficult to learn and implement
- Reactive forms are only for "complex" applications
- Adding custom validations are tricky in Reactive Forms

Tutorial #60 - Reactive Forms - FormGroup, FormControl

Just Remember 3 important things in Reactive Forms

ARC Tutorials

- FormControl
- FormGroup
- FormBuilder

ARC Tutorials

Reactive Forms in 5 simple Steps

We will learn and do hands-on exampels of the 5 simple steps

- **How to use Reactive Forms**

ARC Tutorials

- **Step #1 – Import and Add in the ReactiveFormsModule in the app.module.ts**
  - For template driven forms – **ReactiveFormsModule** needs to be imported
  - If we do NOT import this – we will get error
  - Add the module into the array list of imports
- **Step #2 – Create the form in app.component.html**
  - **FormGroup**
    - We need to use the directive FormGroup for the entire form and give it a name
  - **formControlName**
    - Every form field should have a “formControlName” attribute

```
Reactive Forms
    Form
        Elements
            Input -> FormControl
            Textarea -> FormControl
            buttons
            checkbox
            radio
            Select
            Image

    FormGroup
        Each element is a FormControl
        when one or more formcontrol are grouped together - we call it formgroup

    Form
        FormControlName
```

```
export class LoanTypesComponent implements OnInit {
  addLoanTypesForm : FormGroup;

  constructor() { }

  ngOnInit(): void {
    this.addLoanTypesForm = new FormGroup({
      'loanName' : new FormControl(),
      'loanDescription': new FormControl()
    })
  }

}
```

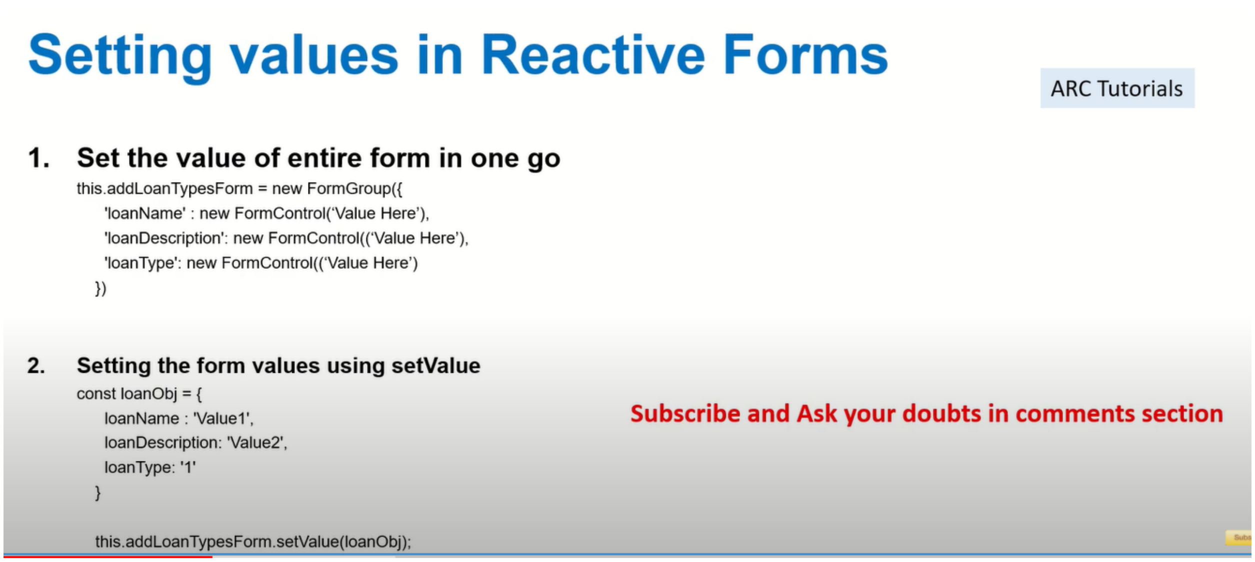
```
<form [formGroup]="addLoanTypesForm">
    <input type="text" formControlName="loanName">
    <textarea formControlName="loanDescription"></textarea>
    <button (click)="addLoanType()">Add Loan Type</button>
</form>
```

Tutorial #61 - Reactive Forms - FormBuilder

```
FormBuilder
    - Its an abstraction layer which makes it easy to design and build our form
    - FormBuilder to work when complex form structure
        Add / Remove
        Add / Remove
        Array of form controls

    - FormBuilder has 3 important
        FormGroup
        FormArray
        FormControl

    - FormBuilder is the preferred one

    - FormBuilder -> break down your form into smaller pieces of code
        -> which is reusable / simple to maintain
```

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

ngOnInit(): void {

  /*
  this.addLoanTypesForm = new FormGroup({
    'loanName' : new FormControl(),
    'loanType': new FormControl()
  })
  */

  this.addLoanTypesForm = this.fb.group({
    'loanName' : new FormControl(),
    'loanType': new FormControl()
  })
```

Tutorial #62 - Reactive Forms - Set Form Values



```
this.addLoanTypesForm = this.fb.group({
  'loanName' : new FormControl('Sooryas Loan'),
  'loanType': new FormControl('Personal Loan'),
  'loanDescription': new FormControl('This is for laptop'),
})

}
```

Alternative way

```
this.addLoanTypesForm = this.fb.group({
  'loanName' : new FormControl(),
  'loanType': new FormControl(),
  'loanDescription': new FormControl(),
})

const newLoanObj = {
  'loanName' : 'My Loan Application',
  'loanType': 'Personal Loan',
  'loanDescription': 'This is loan Application'
}

// Another way of settings values of form
this.addLoanTypesForm.setValue(newLoanObj);
```

```
// Another way of settings values of form
this.addLoanTypesForm.setValue(newLoanObj);

// The only difference is -> you do not have to pass all "keys/fields"
this.addLoanTypesForm.patchValue(newLoanObj);

/*
  there is no right or wrong way
  Encrouage you to use setValue method

  But
    -> The setValue requires data/values for all the "fields/keys"
    -> if you dont pass value -> you will see errors

  Use PatchValue
    -> You dont have to pass all key/values
    -> only selected keys/fields data can be set
*/
```

Tutorial #63 - Reactive Forms - Read Form Values

Read Reactive Form Values

1. Get Value of Entire Form
   this.addLoanTypesForm.value

2. Get Values of specific Form Control
   this.addLoanTypesForm.get('loanName').value;

3. Get values on changes
   this.addLoanTypesForm.valueChanges

```
addLoanType(){
  /*
    - Get Entire Form in one go
      this.addLoanTypesForm.value

    - Get a specific form control
    this.addLoanTypesForm.get('loanType').value

    - valueChanges
        -> subscribe
  */
  // console.log(this.addLoanTypesForm.value);
  // console.log(this.addLoanTypesForm.get('loanType').value);

  this.addLoanTypesForm.valueChanges.subscribe(data => {
    console.log(data);
  });
```

Tutorial #64 - Reactive Forms Validations

Built-In Reactive Form Validations

1. **Validations in FormControl**
   this.addLoanTypesForm = new FormGroup({
   'loanName' : new FormControl('some', [
   Validators.minLength(10),
   Validators.required
   ])

2. **Multiple Validations using Validators.compose**
   'loanName' : new FormControl('some', [
   Validators.minLength(10),
   Validators.required,
   Validators.pattern('^[a-zA-Z]+$')
   ]),

3. **Get the state of the form -> valid or not**
   <button (click)="addLoanType()" [disabled]="!addLoanTypesForm.valid">Add Loan Type</button>

4. **Disable the form button**
   this.addLoanTypesForm.valid

```
this.addLoanTypesForm = this.fb.group({
  'loanName' : new FormControl('', [
    Validators.required,
    Validators.minLength(10),
    Validators.maxLength(20)
    ]
  ),
  'loanType': new FormControl('', Validators.required),
  'loanDescription': new FormControl('', Validators.compose([
    Validators.required,
    Validators.minLength(10),
    Validators.maxLength(20)
  ]))
})
```

Tutorial #65 - Reactive Forms State and Validations

Reactive Form – Form States

1. .ng-valid
2. .ng-invalid
3. .ng-pending
4. .ng-pristine
5. .ng-dirty
6. .ng-untouched
7. .ng-touched

```
if(this.addLoanTypesForm.invalid){

}

console.log(this.addLoanTypesForm.valid); // true
console.log(this.addLoanTypesForm.invalid); // false
console.log(this.addLoanTypesForm.pending); // false
console.log(this.addLoanTypesForm.pristine);    // false
console.log(this.addLoanTypesForm.dirty); // true
console.log(this.addLoanTypesForm.touched); // true
console.log(this.addLoanTypesForm.untouched); // false
```

Tutorial #66 - Reactive Forms Reset Example

Reactive Form – Reset()

1. **Reset Form – reset()**
2. We can Reset the entire form using reset() method

   Syntax:
   this.registerForm.reset()

```
resetForm(){
  this.addLoanTypesForm.reset();
}
```

Tutorial #67 - Reactive Forms Value Changes

Reactive Form – Track Value Changes

1. **Reset Form – valueChanges()**
2. valueChanges is yet another important property of Forms or FormControl
3. valueChanges returns an Observable
4. We need to Subscribe to the Observable to read the value
5. valueChanges is a property in AbstractControl
6. valueChanges will emit an event every time there is any change in the values of the control changes

   Syntax:
   this.registerForm.valueChanges()
   this.formName.get('email').valueChanges.subscribe(data => {
   Console.log(data);
   })

Subscribe and Ask your doubts in commer

```
/*
this.addLoanTypesForm.get('loanName').valueChanges.subscribe(data => {
  console.log(data);
})
*/

this.addLoanTypesForm.get('loanName').valueChanges.pipe(
  map(data => {
    data
  })
)
```

Tutorial #68 - Reactive Forms State Changes

Reactive Form – statusChanges()

ARC Tutorials

1. We can subscribe to status changes happening in the form at any time
2. **statusChanges** is yet another important property of FormControl, FormGroup, and FormArray
3. **statusChanges** returns an Observable. We need to Subscribe to the Observable to read the value
4. **statusChanges** is a property in AbstractControl
5. statusChanges will emit an event every time there is any change in the validation status of the control changes

Syntax:

```
this.registerForm.statusChanges(data => {

});

this.formName.get('fname').statusChanges.subscribe(data => {
Console.log(data);
})
```

```
this.addLoanTypesForm.statusChanges.subscribe(data=> {
  console.log("Form Status");
 console.log(data);
 })

 /*
this.addLoanTypesForm.get('loanName').statusChanges.subscribe(data=> {
 console.log(data);
 })
 */
```

How to debug angular application in chrome

How to debug Angular apps

ARC Tutorials

1. Adding “console” statements in code to debug – simplest way

ARC Tutorials

2. Checking the sources in developers console log and adding breakpoints
3. Add the json pipe in template to check if the values are correct!
4. Always unsubscribe the subscription – always, else it will lead to memory leaks!!!
5. If making HTTP Calls -> Inspect the Network Tab for data being sent/received
6. Always “lint” to make sure – bad code does not peek in

Tutorial #69 - Reactive Forms - FormArray

Reactive Form – Form Array

1. So far, we have learned and used **FormControl**, **FormGroup** and **FormBuilder**
2. Helps in building basic form with form elements
3. one of the most important feature working in Reactive Forms.
4. For complex forms, we will need Form Arrays
5. Almost all modern applications will need us to work with Form Arrays
6. DOM interactions in Angular Reactive Forms are implemented using the Form Arrays
7. Adding and Removing elements is very easily handled using FormArrays

Reactive Form – Form Array

ARC Tutorials

8. **FormArray** aggregates the values of the “child” **FormControl** into an Array
9. The status of the **FormArray** is calculated by reducing the statuses of it’s children.
10. If any one of the controls is invalid, the entire array becomes invalid

```
FormArray

    Form
        Add Address ->
        Multiple Addresses
            Multiple Addresses
                Address 1
                Address 2
                State
                City
                Zipcode

        Add Multiple Files
            Gmail
                multiple files to an email
                    New File
```

```
    Resume Builder
        Multiple colleges
            College Name
            Degree
            Percentage

        Multiple Experiences
            Company Name
            Designation

    Invoice Builder
        Products
            Product Name
            price
            tax
            finalAmount
```

```
Form
    The forms pretty static
        We defined the formControl in our component/FormBuilder
            Template

    Dynamically adding FormControls to our form

    FormArray
        - its an Array
            - FormControl
            - FormGroup
            - FormArray
            - We can have a combination of above abstarcts

                FormArray
                    - FormControl
                    - FormArray
```

```
                FormArray
                    - FormContols
                    - FormGroup
                    - FormGroup
                    - FormControls
```

```
this.addLoanTypesForm = this.fb.group({
  'loanName' : new FormControl('', [
    Validators.required,
    Validators.minLength(10),
    Validators.maxLength(20)
    ]
  ),
  'loanType': new FormControl('', Validators.required),
  'loanDescription': new FormControl('', Validators.compose([
    Validators.required,
    Validators.minLength(10),
    Validators.maxLength(20)
  ])),
  'users': new FormArray([
    new FormControl('ARC'),
    new FormControl('Tutorials')
  ])
})
```

```
<!-- most developers make this mistake -->

<tr>
    <div class="form-group" formArrayName="users">
        <label>Users</label>
        <div *ngFor="let control of addLoanTypesForm.controls.users['contro
            <input type="text" [formControlName]="i">
        </div>
    </div>

</tr>
```

Another way

```
'users': new FormArray([
    this.fb.group({
        'name': new FormControl(''),
        'age': new FormControl(''),
        'dept': new FormControl('')
    })
])
})
```

```
<tr>
    <div class="form-group" formArrayName="users">
        <label>Users</label>
        <div *ngFor="let control of addLoanTypesForm.controls.users['controls'];i=index

        </div>
    </div>
</tr>
```

```
" formArrayName="users">
l>
ontrol of addLoanTypesForm.controls.users['controls'];i=index;" [formGroupName]="i">
```

[Tutorial #70 - Reactive Forms Add Remove FormControl]()

Add Rows Dynamically into Form

ARC Tutorials

- There will be use cases when we will not know how many rows or fields to expect.
- That’s when we will need to dynamically “Add Rows” to the form on the fly
- Adding new Rows to form simply refers to appending/pushing the form Groups or Form Contols to the Form Array at run time
- Form Array
    - Form Group
    - Form Control

Remove Rows Dynamically into Form

ARC Tutorials

- There will be use cases when we will have to remove the dynamically added rows from the forms
- That's when we will need to dynamically "Remove Rows" from the form on the fly
- We will need to capture the index of the row which we want to remove from the array
- Using the indexAt method – we can easily point to the item we want to remove


```
addUser(){
  let userArr = this.addLoanTypesForm.get('users') as FormArray;
  let newUser = this.fb.group({
    'name': '',
    'age':'',
    'dept':''
  });

  userArr.push(newUser);
}
```

```
removeUser(i){
  let arr = this.addLoanTypesForm.get('users') as FormArray;
  arr.removeAt(i);
}

assignDept(i){
  let arr = this.addLoanTypesForm.get('users') as FormArray;
  let values = arr.value;
  arr.value[i].dept = 'CSE';
  this.users.setValue(arr);
```

Tutorial #71 - Component Communication

After this tutorial – you will be absolute master in Angular components

Angular – Components Explained

- Learn how to create components
  - Learn how to use different template
  - Learn to use multiple style URLs

ARC Tutorials

- Learn how to use Components from other modules
- Learn Communication between Components - using @Input and @Output
- Angular component communication using “Service”
- Angular lifecycle hooks


```
2. How do you use Angular component
    -> Can we pass custom selector name? -> Yes of course
    -> Use the select tag as directive

3. How do you use Angular components from different module?
    - Users Module
        -> component -> list-users
    - Import the custom/newly created module into App Module
    - Export the respective components that we want to use outside of the module

4. Angular components communication
    -> @Input and @Output directives
    -> Using services
```

```
4.1 Input and Output
    -> Parent/Child -> Parent -> sending data to child
        -> list-users -> Input -> filter-users
        -> list-users <- Output <- filter-users

    -> We need to pass data from parent using [inputName]
    -> Same needs to be defined in the child component as @Input()

    -> Angular components communicate Input and Output

5. Angular component communicate via Services
```

```
5. Angular component communicate via Services
    -> Create a service
        -> ng g service <service-name>

    -> Implement methods which can be shared between components for sharin
```

```
6. Angular component lifecycle
    -> Component series of states
        -> Lifecycle hooks
            -> ngOnChanges -> Listen to chanegs in data/form/state/value c
            -> ngOnInit -> Absolute -> On initalization of the component
                -> http call and write subscribe in ngOnInit

            -> ngDoCheck
                -> ngAfterContentInIt -> If you want to wait till initaliz
                -> ngAfterContentChecked
                -> ngAfterViewInit ->
                -> ngAfterViewChecked
            -> ngOnDestroy -> When you want to exit the component
                -> Unscribe the subscription
            ->
```

Tutorial #72 - Services

Dependency Injection

- Dependency injection (DI), is an important application design pattern.
- Dependency injection is the ability to add the functionality of components at runtime
- The DI framework lets you supply data to a component from an injectable service class, defined in its own file
- Angular has its own DI framework, which is typically used in the design of Angular applications to increase their efficiency and modularity.
- Dependencies are services or objects that a class needs to perform its function.
- DI is a coding pattern in which a class asks for dependencies from external sources rather than creating them itself.

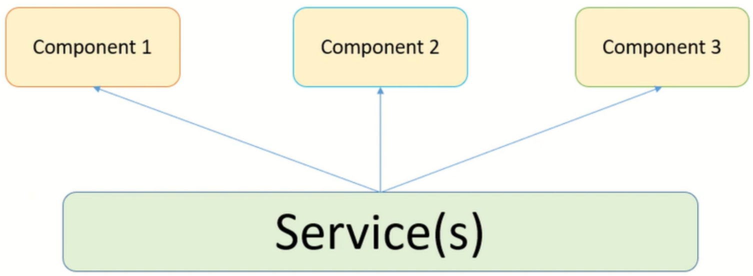
Angular Services

Introduction

1. Services allows us to create reusable common shared functionality between various modules and components
2. Services are singleton
3. Services are injected into application using Dependency Injection mechanism
4. We need to create and inject services in components where we want to use them
5. Services are an abstraction layer or process layer which consists of our application business logic
6. Services are commonly used for making HTTP requests to our endpoints APIs to request and receive the response
7. A service can have a value, methods or a combination of both!

Services serve multiple components

ARC Tutorials



Generate Service

ARC Tutorials

1. ng generate service <service_name>

2. import { Injectable } from '@angular/core';

```
4. Whats inside a service file?
    -> @Injectable
        -> Dependency Injection Mechanism
    -> ProvidedIn
        -> Its available whereever we inject it

5. Where can we inject the services?
    -> Inject in any component
        -> where we want to use it

6. Do we need to import this services in any Module file?
    - NO
    - only imported in component where its used
```

@Injectable

ARC Tutorials

1. Dependency injection (DI), is an important application design pattern.

2. The DI framework lets you supply data to a component from an injectable service class, defined in its own file

3. DI is a coding pattern in which a class asks for dependencies from external sources rather than creating them itself.

4. A class which has the injectable decorator becomes a service class

5. The injectable decorator allows the functionality of this class to be injected and used in any Angular JS module

Step by Step Guide

1. **Generate the Service**
   ng generate service leads

2. **Import the Service in the component class**
   Import { leads } from '../leads.service';

3. **Initialize in the constructor method i.e Dependency Injection**
   constructor(private formBuilder: FormBuilder, private leadsService: LeadsService)

4. **Call the methods and properties in the class (component class)**
   this.leadsService.methodName()

   OR

1. **Call the methods directly in the template (html file)**
   Make sure your service instance is PUBLIC – else error in compiling

   leadsService.methodName

Tutorial #73 - HttpClient

Introduction

ARC Tutorials

1. **HttpClient** is used for performing HTTP requests and responses.

2. The **HttpClient** service is available in the **@angular/common/http** package

3. The new **HttpClient** service is included in the Http Client module which is used to initiate HTTP request and responses in Angular apps

4. The **HttpClientModule** needs to be imported into the module. Usually in the app module.

5. **HttpClient** also gives other useful functionality like params, interceptors, headers etc

Http Methods

ARC Tutorials

- get()
- post()
- put()
- delete()
- head()
- jsonp()
- options()
- patch()

Benefits of HttpClient

1. HttpClient includes Observable APIs
2. HttpClient can have the HTTP Headers in options
3. HttpClient includes the highly testability features
4. HttpClient includes typed request
5. HttpClient includes response objects
6. HttpClient includes request and response interception
7. HttpClient include error handling

Standard CRUD Operations

1. **Create** – usually a POST method call
2. **Read** – usually a GET method call
3. **Update** – Usually a PUT method call
4. **Delete** – is a Delete method Call.
   Technically – we should NEVER delete anything. Only soft deletes which means setting a flag or so.

Tutorial #74 - HTTP GET