Angular JS – DM

(github repository: github.com/deep.mm/Angular-JS)

1. Run in **terminal:** *npm install -g @angular/cli*

2. Run in **terminal**: *ng new angular-ait*

[Steps 3-7 are optional – Just for design]

3. Run in **terminal**: *ng add @angular/material*
   | -> Select theme
   | -> Hammer JS = Yes
   | -> Browser animation = Yes

4. Create a new **module**: *ng g m material*

5. Contents of **src/app/material/material.module.ts:**

```
import { NgModule } from '@angular/core';
import { MatButtonModule } from '@angular/material';

const MaterialComponents = [
  MatButtonModule
];
@NgModule({
  imports: [
    MaterialComponents
  ],
  exports: [
    MaterialComponents
  ]
})
export class MaterialModule { }
```

Now to add any modules of material library, directly add it to MaterialModules array and you are good to go

6. In **app.module.ts** in the import tab add **MaterialModules**

```
imports: [
    BrowserModule,
    AppRoutingModule,
    BrowserAnimationsModule,
    MaterialModule
  ],
```

7. Now, to use material button in html, in **app.component.html**:

```
<button mat-raised-button>Login</button>
```

8. In **app.component.ts** file:

```
export class AppComponent {
  title = 'angular-ait';
  age = 10.1;
  name = 'Deep';
  today = Date.now();
}
```

9. Using **inbuilt pipes**:

```
<h3>{{ name | uppercase }}</h3> //Converts a string to uppercase.
Similarly we have lowercase & titlecase pipes

<h3>{{ age | number:'2.3-4'}}</h3>
//2 is the number of digits before the decimal point, 3 is min number of
digits after decimal & 4 is max number of digits after decimal

<h3>{{ age | currency:'INR'}}</h3> //Enter the currency of the country

<h3>{{ age | percent}}</h3> //To display a number as percentage

<h3>{{ today | date: 'medium'}}</h3> //Format the date

<h3>{{ name | slice: 0:2 }}</h3>
//To get substring of a string 0:included, 2:excluded

<h3>{{ name | slice: 0:2 | lowercase}}</h3> //Chaining of pipes
```

10. Making **custom pipes:**

    a. Run in **terminal**: *ng g p reverseString*

    b. Let's create a pipe to reverse a string and concat another string with it

    In **reverse-string.pipe.ts**:

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

@Pipe({
  name: 'reverseString'
})
export class ReverseStringPipe implements PipeTransform {

  transform(value: string, concat: string): string {
    //let exp = parseFloat(exponent); -> To convert string to number
    return value.split('').reverse().join('') + concat;
```

```
  }
}
```

c. Use the pipe in html file:

```
<h3>{{ name | reverseString: Hello }}</h3>
// Hello is sent as concat param
```

d. Using pipes inside ts file:

```
let uppercase = new UpperCasePipe();
let str = value.split('').reverse().join('') + concat;
return uppercase.transform(str);
```

e. To make a pipe pure/impure, change the pure value to true/false:

```
@Pipe({
  name: 'reverseString',
  pure: true
})
```

11. Adding **bootstrap** to the project, in **index.html** file add this line in <head>:

```
<link rel="stylesheet"
href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/bootstrap.min.c
ss" integrity="sha384-
ggOyR0iXCbMQv3Xipma34MD+dH/1fQ784/j6cY/iJTQUOhcWr7x9JvoRxT2MZw1T"
crossorigin="anonymous">
```

12. **Forms** in Angular:

a. Let's first build a form using bootstrap classes in **app.component.html**:

```
<div class="container-fluid col-md-6">
  <h1>Angular Template Form</h1>
  <form>
    <div class="form-group">
      <label>Name</label>
      <input type="text" class="form-control">
    </div>

    <div class="form-group">
      <label>Email</label>
      <input type="email" class="form-control">
    </div>

    <div class="form-group">
      <label>Mobile</label>
      <input type="tel" class="form-control">
    </div>

    <div class="form-group">
      <label>Subjects:</label>
      <select class="custom-select">
        <option selected>I am interested in</option>
        <option *ngFor="let topic of topics">{{ topic }}</option>
      </select>
    </div>

    <div class="form-check mb-3">
      <input class="form-check-input" type="checkbox">
      <label class="form-check-label">Accept terms & conditions</label>
    </div>

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

b. In **app.module.ts**:

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

and in **Imports** array add FormsModule

c. In the **html** file where the form is make these changes:

Add #userForm directive to get all values from form

```
<form #userForm = 'ngForm'>
```

To each input tag, add name & ngModel tag

```
<input type="text" class="form-control" name="username" ngModel>
```

Use ** ** if you want to merge more than one input values into a single group.
Eg: An address can have flat, street, pincode and so on.. And thus merge all into one group called address

d. Let’s generate a new **class** to define an object: *ng g class User*

e. Contents of **user.ts** to store form data:

```
export class User {
    constructor(
        public name: string,
        public email: string,
        public phone: number,
        public topic: string,
        public subscribe: boolean
        ) {}
}
```

f. To add **horizontal** line in html: *<hr/>*

g. In all **input** fields, replace **ngModel** by,

```
[ngModel] = "userModel.name
```

and so on for all input fields

h. Now we have one-way property binding in our form, but to make this **two-way**, Make use of round parenthesis inside the square ones

```
[(ngModel)] = "userModel.name"
```

i. Now, in the **input** field, make following changes:

```
<input type="text" required #userName="ngModel" class="form-control"
name="username" [(ngModel)] = "userModel.name">
```

Thus the variable username is now binded with ngModel, and now we can use properties like username.untouched, username.valid and so on to write validations.

j. Now adding validations checks in input field,

```
<input type="text" required #userName="ngModel"
[class.is-invalid]="userName.invalid
&& userName.touched" class="form-control" name="username" [(ngModel)] =
"userModel.name">
```

Here as we can see, the class is-invalid (bootstrap class) is applied only when the name field is touched at least once and is blank.

k. To add an error message, below the fields enter this line in the ** ** of that field

```
<small class="text-danger" [class.d-none]="userName.valid ||
userName.untouched">Name is required</small>
```

l. To display specific error messages w.r.t different errors, here in phone field:

```
<div class="form-group">
      <label>Mobile</label>
      <input type="tel" #phone="ngModel" pattern="^\d{10}$" [class.is-
invalid]="phone.invalid && phone.touched" class="form-control"
name="mobile" [(ngModel)] = "userModel.phone" required>
      <div *ngIf="phone.errors && (phone.invalid || phone.touched)">
          <small class="text-danger" *ngIf="phone.errors.required">Mobile
number is a required field</small>
          <small class="text-danger" *ngIf="phone.errors.pattern">Mobile
number must be 10 digits</small>
      </div>
    </div>
```

Thus by using *ngIf we can specify conditions if that field must be visible or not

m. To check for **specific error using functions**, use this:

```
<div class="form-group">
      <label>Subjects:</label>
      <select (blur)="validateTopic(subject.value)"
(change)="validateTopic(subject.value)" #subject="ngModel" [class.is-
invalid]="topicHasError && subject.touched" class="custom-select"
[(ngModel)] = "userModel.topic" name="subjects">
        <option value="default">I am interested in</option>
        <option *ngFor="let topic of topics">{{ topic }}</option>
      </select>
      <small class="text-danger" [class.d-none]="!topicHasError ||
subject.untouched">Please choose a subject</small>
    </div>
```

ValidateTopic function is defined in **app.component.ts** as:

```
validateTopic(value) {
    if (value === 'default') {
      this.topicHasError = true;
    } else {
      this.topicHasError = false;
    }
  }
```

Thus depending on selected value the Boolean variable **typeHasError** is toggled and thus used in conditions in the [class] group

n. To check if the **whole form** is valid or not, we can use the userForm directive to do this: (mainly done with submit button)

```
<button class="btn btn-success" type="submit"
[class.disabled]="userForm.form.invalid">Submit</button>
```

o. To prevent browser validation, add this to the form line:

```
<form #userForm = 'ngForm' novalidate>
```

p. Method to run when submit is clicked and all fields are valid:

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

13. Make a new **Service** to submit form data to server: *ng g s enrolment*

14. Import HttpClientModule in **app.module.ts**:

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

15. Now in the service, **enrollment.service.ts**:

```
export class EnrollmentService {

  _url = 'anyUrl';
  constructor(private _http: HttpClient) { }

  enroll(user: User){
    return this._http.post<any>(this._url, user);
  }
}
```

16. In app.component.ts in the onSubmit() method:

```
constructor(private _enrollmentService: EnrollmentService) {}

onSubmit(){
    console.log(this.userModel);
    this._enrollmentService.enroll(this.userModel)
    .subscribe(
      data => console.log('Success!',data),
      error => console.log('Error!',error)
    );
  }
```