Contents

Forms

- There are 2 types of forms
    - Template Driven Forms
        - Heavy coding on the component template.
    - Reactive Forms
        - Heavy coding on the component class.

Template Driven Forms (TDF)

- Two way binding with ngModel.
- Bulky HTML and minimal component code.
- Automatically track the form and form elements state and validity.
- Readability decreases with complex forms and validations.
- Suitable for simple scenarios.
- Steps to Implement TDF
    1. Add HTML Form

```
<form>
    <div class="row">
        <div class="col-sm-4 form-group">
            <label>First Name</label>
            <input type="text" class="form-control">
        </div>
        <div class="col-sm-4 form-group">
            <label>Middle Name</label>
            <input type="text" class="form-control">
        </div>
        <div class="col-sm-4 form-group">
            <label>Last Name</label>
            <input type="text" class="form-control">
        </div>
    </div>
    <div class="row">
        <div class="col-sm-12"><br></div>
    </div>
    <div class="row">
        <div class="col-sm-4">
            <label>Select Gender</label>
            <div class="form-group">
                <input type="radio" name="gender" value="male">
                <label class="form-check-
label"> Male </label>
                <input type="radio" name="gender" value="female">
                <label class="form-check-label"> Female</label>
            </div>
        </div>
        <div class="col-sm-4 form-group">
            <label>Date of Birth</label>
            <input type="date" class="form-control">
```

```
        </div>
        <div class="col-sm-4 form-group">
            <label>Mobile Number</label>
            <input type="number" class="form-control">
        </div>
    </div>
    <div class="row">
        <div class="col-sm-12"><br></div>
    </div>
    <div class="row">
        <div class="col-sm-4 form-group">
            <label>Role</label>
            <select class="form-control">
                <option *ngFor="let role of roles"
value="role.value">
                    {{role.name}}
                </option>
            </select>
        </div>
        <div class="col-sm-4">
            <label>Time Preference</label>
            <div class="form-group">
                <input type="radio" name="timePreference"
value="morning">
                <label class="form-check-label">  Morning (9AM-
6PM)  </label>
                <input type="radio" name="timePreference"
value="night">
                <label class="form-check-label">  Night (5PM-
2AM)</label>
            </div>
        </div>
    </div>
    <div class="row">
        <div class="col-sm-12"><br></div>
    </div>
    <div class="row">
        <div class="col-sm-12">
            <button class="btn btn-sm btn-success float-
end">Submit</button>
        </div>
    </div>
</form>
```

2. Binding Data with ngForm

- Import formsModule in the imports array in app.module.ts

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { FormsModule } from '@angular/forms';

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

- ngForm is a directive in Angular that is automatically applied to <form> elements when you import the FormsModule. It is part of Angular's template-driven forms approach and provides a way to track the form and its controls, as well as to hook into form submission and validation.
- Here are some key aspects of ngForm:
  - Form Binding
  - Form Submission
  - Accessing Form Controls
  - Tracking Form State
  - Disabling Submit Button
  - Accessing Form Properties in the Component

```
<form #employeeForm="ngForm">

        {{ employeeForm.value | json}}
```

- Below the form tag we are just printing the value of the form just to see the value of every form control(input).
- Next we need to keep ngModel to every input control.

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

- But if we enter in the input control we can't able to see the form value which we are printing and in the console we will be having a error because to use ngModel either we need
  - Name attribute must be in the tag.
  - 2-way Binding is required.

- For now lets set name attribute

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

- Now we can see the error will be gone and value of that particular input control is visible in the place where we are printing the form values.
- **ngModelGroup** is a directive in Angular that is used in conjunction with ngModel to group together related form controls within a template-driven form. It allows you to create a nested structure of form controls, which can be useful for organizing and managing complex forms.

```
<div class="row" [(ngModel)]="employeeData"Group="address">
     <div class="col-sm-3 form-group">
          <label>Street</label>
          <input type="text" class="form-control" name="street">
     </div>
     <div class="col-sm-3 form-group">
          <label>City</label>
          <input type="text" class="form-control" name="city">
     </div>
     <div class="col-sm-3 form-group">
          <label>State</label>
          <input type="text" class="form-control" name="state">
     </div>
     <div class="col-sm-3 form-group">
          <label>Postal Code</label>
          <input type="text" class="form-control"
name="postalCode">
     </div>
</div>
```

3. Binding Data to a Model

- First generate a modal class. By using below function we can create a class.
  ng generate class <class_name>
- In our example we are entering employee details so we will be creating a employeeClass.
- Now we will be creating a class with different properties of the class.

```
export class EmployeeModel{
    fname?:string = undefined;
    mname?:string = undefined;
    lname?:string = undefined;
    gender?:string = undefined;
    dob?:string = undefined;
    mobileNumber?:number = undefined;
    role?:string = undefined;
    timePreference?:string = undefined;
}
```

- Next, we will be creating the instance of this model and assign values to the properties of the model in our forms component ts file.

```
employeeData:EmployeeModel = new EmployeeModel();
ngOnInit(): void {
  this.employeeData = this.getEmployeeData();
}

getEmployeeData(): EmployeeModel {
  return{
    fname: "Mohnish",
    mname: "Swamy",
    lname: "Purella",
    gender: "male",
    dob: "15-05-1997",
    mobileNumber: 7075012545,
    role: "development",
    timePreference: "morning"
  }
}
```

- Now let's keep our instance in our html.

```
{{employeeData | json}}
```

- Now, Binding the user model to the form is really simple we bind the properties of the model to ngModel directive and for property binding we make use of [] brackets.

```
<input #firstName type="text" class="form-control" name='fname'
[ngModel]="employeeData.fname">
```

- Thought we are doing property binding the model instances property value will not be changes/updated so to resolve that we have to implement 2-way binding instead of property binding.

```
<input #firstName type="text" class="form-control" name='fname'
[(ngModel)]="employeeData.fname">
```

4. Tracking state and validity

State	Class if true	Class if false
Field/Control has been visited	ng-touched	ng-untouched
Field/Control has changed	ng-dirty	ng-pristine
Field/Control is valid	ng-valid	ng-invalid

- Now, lets keep a template reference variable to the bind the variable to input class name property.

```
<div class="col-sm-4 form-group">
     <label>First Name</label>
     <input #firstName type="text" class="form-control"
name='fname' [(ngModel)]="employeeData.fname">
     {{firstName.className}}
</div>
```

- In the browser we can see the classes which are applied to the form field/control. Where, we can see the 3 class present which we mentioned above.
- To check the ng-valid/ng-invalid we need to keep required keyword to form control/field.

```
<div class="col-sm-4 form-group">
     <label>First Name</label>
     <input #firstName type="text" class="form-control"
name='fname' [(ngModel)]="employeeData.fname" required>
     {{firstName.className}}
</div>
```

- Angular also provide alternative approach which is better.
- In this approach angular provide an associated property for these classes on ngModel directive.

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

- To access these ngModel properties we need to create a reference to the ngModel directive.
- So, if we see the template reference variable of the form control/field points to the input element in the DOM by assigning it a value of ngModel then the reference variable points ngModel of this particular form control/field.

```
<input #firstName = "ngModel" type="text" class="form-control"
name='fname' [(ngModel)]="employeeData.fname" required>
{{firstName.untouched}}
```

5. Validation with Visual Feedback
   - For visual feedback of validation can be done in 2 ways
     1. By creating own class with own styles.
     2. By using predefined css(bootstrap) classes.
   - What ever method we are using we must use them conditionally.

```
<input #firstName = "ngModel" [class.is-
invalid]="firstName.invalid" type="text" class="form-control"
name='fname' [(ngModel)]="employeeData.fname" required>
```

   - We can even keep more than 1 condition.

```
<input #firstName = "ngModel" [class.is-
invalid]="firstName.invalid && firstName.untouched" type="text"
class="form-control" name='fname'
[(ngModel)]="employeeData.fname" required>
```

   - We can even keep patterns for check valid or invalid.
   - So lets keep pattern for mobile number i.e if it contain 10 digits its valid if not its invalid.

```
<input type="tel" #phone = "ngModel" pattern="^\d{10}$"
[class.is-invalid]="phone.invalid && phone.untouched"
class="form-control" name='mobileNumber'
[(ngModel)]="employeeData.mobileNumber">
```

6. Displaying Error Messages

```
<input #firstName = "ngModel" [class.is-invalid]="firstName.invalid
&& firstName.untouched" type="text" class="form-control"
name='fname' [(ngModel)]="employeeData.fname" required>
<small class="text-danger" [class.d-none]="firstName.valid ||
firstName.untouched">Name is requied</small>
```

   - Here **d-none** means don't show the tag.

```
<input type="tel" #phone = "ngModel" pattern="^\d{10}$" [class.is-
invalid]="phone.invalid && phone.untouched" class="form-control"
name='mobileNumber' [(ngModel)]="employeeData.mobileNumber"
required>
<div *ngIf="phone.errors && (phone.invalid || phone.touched)">
     <small class="text-danger" *ngIf="phone.errors['required']"
>Phone number is required</small>
     <small class="text-danger" *ngIf="phone.errors['pattern']"
>Phone number must be of 10 digits</small>
</div>
```

   - Here I am showing 2 validation messages for mobile number
     1. Mobile number is required
     2. Mobile number must be of 10 digits
   - According to the error any one message will be displayed.

7. Select control validation

```
<select change)="validateRole(role.value)" #role="ngModel"
[class.is-invalid]="roleHasError && role.touched" class="form-
control" name="role" [(ngModel)]="employeeData.role">
     <option *ngFor="let role of roles" [value]="role.value">
             {{role.name}}
     </option>
</select>
<small class="text-danger" [class.d-none]="!roleHasError ||
role.untouched">Please Choose a topic</small>
```

- In ts file

```
roleHasError = true;

validateRole(value:any){
  if(value == ''){
    this.roleHasError = true;
  }else{
    this.roleHasError = false;
  }
}
```

8. Form validation
   - Since we are binding our form with ngForm.
   - Similarly like ngModel directory ngForm also have validation property which tells weather the entire form is valid or not instead of individual fields.

```
<form #employeeForm="ngForm">

        {{employeeForm.valid}}
```

   - With the above code help we can validate our entire form i.e disabling the submit button if form is invalid.

```
<button class="btn btn-sm btn-success float-end"
[disabled]="employeeForm.invalid || roleHasError">Submit</button>
```

   - Here with invalid we are keeping roleHasError because we are validating the select field with a variable so we have to keep that particular variable in the condition.
   - We must also check if the variables field has value or not.

```
ngOnInit(): void {
  this.employeeData = this.getEmployeeData();
  if(this.employeeData.role){
    this.roleHasError = false;
  }
}
```

9. Submitting form data

- First step is to keep novalidate attribute on form tag. This will prevent browsers validation when submit button is clicked.

```
<form #employeeForm="ngForm" novalidate>
```

- Next step is to bind ngSubmit which gets emitted when submit button is clicked.

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

- Lets define onSubmit handler in ts file.

```
onSubmit(){
  console.log(this.employeeData);
}
```

- To send this data to the server we need to use service. To create a servicewe use following command.
  ng g s enrollment
- Now, in
  enrolment.service.ts

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';

@Injectable({
  providedIn: 'root'
})
export class EnrollmentService {

  constructor( private _http: HttpClient) { }
}
```

app.module.ts

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

imports: [
    BrowserModule,
    AppRoutingModule,
    FormsModule,
    HttpClientModule
],
```

- Now,
  enrolment.service.ts

```
export class EnrollmentService {
  constructor( private _http: HttpClient) { }
  _url = '';

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

app.component.ts

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

onSubmit(){
  this.enrollmentService.enroll(this.employeeData)
  .subscribe(
    data => console.log('Sucess!', data),
    error => console.log('Error!', error)
  );
}
```

10. Express Server to Receive Form

- Create a new folder “Server” outside the angular folder and initialize new package json file using the command.

  npm init –yes

- Now, lets install dependencies using the command

  npm install --save express body-parser cors.

  express is our webserver.
  body-parser is the middleware to handle form data
  cors is a package to makes request across different ports

- Now dependencies are installed.
- Create a new file and name its as “server.js” within server folder. Within the file we will be begin by requiring the packages that we have just installed express, body-parser and cors.

```
const express = require('express');
const bodyParser = require('body-parser');
const cors = require('cors');
```

Next we will create const for port number that our express server will run on.

```
const PORT = 3000;
```

Now we will create instance of express and body parser to handle json data and we need to use cors package.

```
const app = express();

app.use(bodyParser.json());

app.use(cors());
```

Next we will add code to test get request.

```
app.get('/', function(req, res){
    res.send('Hello from server');
});

app.listen(PORT, function(){
    console.log("Server running on localhost: " +PORT);
});
```

To start the serve we need to use the command in the server folder path terminal

node server

Once the server starts running give a request from our browser my entering localhost:3000 then you can see the response message there.

If you can see the response message then the express server is running fine.

Now we need to add an endpoint to which our angular application will post the form data.

```
app.post('/enroll', function(req, res){
    console.log(req.body);
    res.status(200).send({"message": "Data received"})
})
```

req.body will be having the data that was submitted by the angular application then we will send a response of status 200 and we will send an object of message “Data received”.

Now, we have an endpoint we will be going back to our angular application and update the “_url” in “the emrollment.service.ts”.

```
_url = 'http://localhost:3000/enroll';
```

Now, restart the server and submit the form.

11. Error Handling

- In the enrollment service we will be catching the error from the server and throwing it. So, we need the help of rxjs.

```
import { Injectable } from '@angular/core';
import { HttpClient, HttpErrorResponse } from
'@angular/common/http';
import { EmployeeModel } from './employeeModel';
import { catchError, throwError } from 'rxjs';

@Injectable({
  providedIn: 'root'
})
export class EnrollmentService {


  constructor( private _http: HttpClient) { }
  _url = 'http://localhost:3000/enroll';

  enroll(employee:EmployeeModel){
    return this._http.post<any>(this._url, employee)
              .pipe(catchError(this.errorHandler))
  }

  errorHandler(error: HttpErrorResponse){
    return throwError(error);
  }
}
```

Change the status in the server from 200 to 401 then restart the server and submit the form.
You can see the error in the console which we are printing.