breaking thought barriers

Angular Training

Session -20

Outlines

Angular Form

Angular Form Validation

Angular Forms

Why we need Forms?

- Forms are the main building blocks of any type of application. When we use forms for login, registration, submission.
- Forms are really very very important to collect the data from the users. Often, each website contains forms to collect the user data.
- You can use forms to login, submit a help request, place an order, book a flight, schedule a meeting, and perform other countless data entry tasks.

What are Angular Forms?

The Angular Framework, provides two different ways to collect and validate the data from a user.

They are :

1. Template-Driven Forms

2. Model-Driven Forms (Reactive Forms)

Template Driven Forms in Angular:

- Template Driven Forms are simple forms which can be used to develop forms.

- These are called Template Driven as everything that we are going to use in an application is defined into the template that we are defining along with the component.

Features of Template Driven Forms:

- ✓ Easy to use.
- ✓ Suitable for simple scenarios and fail for complex scenarios.
- ✓ Similar to Angular 1.0 (Angular JS)
- ✓ Two way data binding using NgModule syntax.
- ✓ Minimal Component code
- ✓ Automatic track of the form and its data.
- ✓ Unit testing is another challenge

Model-Driven Forms (Reactive Forms) in Angular

- In a model driven approach, the model which is created in the .ts file is responsible for handling all the user interactions and validations.
- For this, first, we need to create the model using Angular's inbuilt classes like formGroup and formControl and then we need to bind that model to the HTML form.
- This approach uses the Reactive forms for developing the forms which favor the explicit management of data between the UI (User Interface) and the Model.
- With this approach, we create the tree of Angular Form Controls and bind them in the Native Form Controls.
- As we create the form controls directly in the component, it makes it a bit easier to push the data between the data models and the UI elements.
- In order to use Reactive Forms, you need to import **ReactiveFormsModule** into the applications root module i.e. app.module.ts file.

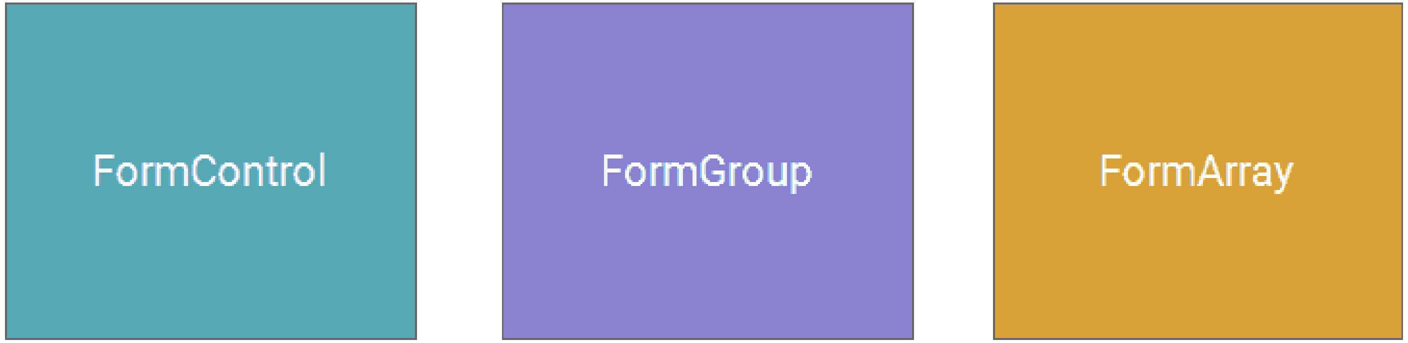
Features of Reactive Forms:

- ✓ More flexible, but need a lot of practice
- ✓ Handles any complex scenarios.
- ✓ No data binding is done (Immutable data model preferred by most developers).
- ✓ More component code and less HTML Markup.
- ✓ Easier unit testing.
- ✓ Reactive transformations can be made possible such as
- ✓ Handling a event based on a denounce time.
- ✓ Handling events when the components are distinct until changed.
- ✓ Adding elements dynamically.

Note: They both are two different approaches, so we can use whichever suits our needs the most. we can use both in the same application.

Building Blocks of Angular Form

The Angular Forms module consists of three Building blocks, irrespective of whether you are using Template-driven or Reactive forms approach.



FormControl

- A FormControl represents a single input field in an Angular form.
- The FormControl is an object that encapsulates all the information related to the single input element.
- It Tracks the value and validation status of each of these control
- The FormControl is just a class.
- A FormControl is created for each form field.
- We can refer them in our component class and inspect its properties and methods
- We can use FormControl to set the value of the Form field, find the status of form field like (valid/invalid, pristine/dirty, touched/untouched) etc & add validation rules to it.

- We can check the validation status of the First Name element as shown below

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

let firstname= new FormControl (); //In Reactive Form

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

FormGroup

- FormGroup is a collection of FormControls .
- Each FormControl is a property in a FormGroup. with the control name as the key.
- Often forms have more than one field. It is helpful to have a simple way to manage the Form controls together.
- Example :

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

- A FormGroup tracks the status of each child FormControl and aggregates the values into one object. with each control name as the key

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

- You can read the value of an address using the value method, which returns the JSON object

```
            address.value
        The Return value
              address {
                 street :"",
                 city:"",
                 Pincode:""
              }
```

- You can access child control as : **address.get("street")**

- Check the Validation status as follows

```
address.errors      // returns the list of errors
address.dirty       // true if the value of one of the child control has changed (dirty)
address.touched     // true if one of the child control is touched
address.valid       // true if all the child controls passed the validation
```

- A typical Angular Form can have more than one FormGroup.
- A FormGroup can also contain another FormGroup.
- The Angular form is itself a FormGroup

FormArray

- FormArray is an array of form controls.
- It is similar to FormGroup except for one difference.
    - In **FormGroup** each FormControl is a property with the control name as the key.
    - In **FormArray** is an array of form controls.
- We define the FormArray as

```
contactForm = new FormGroup( {
  name: new FormControl(''),
  cities:new FormArray([
    new FormControl('Mumbai'),
    new FormControl('Delhi')
  ])
 });
```

- We can get the reference to the cities from the contactForm.get method

```
cities(): FormArray {
   return this.contactForm.get("cities") as
FormArray
  }
```

Angular Template Driven Forms

- In Template Driven Forms we specify behaviors/validations using directives and attributes in our template and let it work behind the scenes.
- All things happen in Templates hence very little code is required in the component class.
- This is different from the reactive forms, where we define the logic and controls in the component class.

The Template-driven forms

1.The form is set up using ngForm directive

2.controls are set up using the ngModel directive

3.ngModel also provides the two-way data binding

4.The Validations are configured in the template via directives

Template-driven forms are

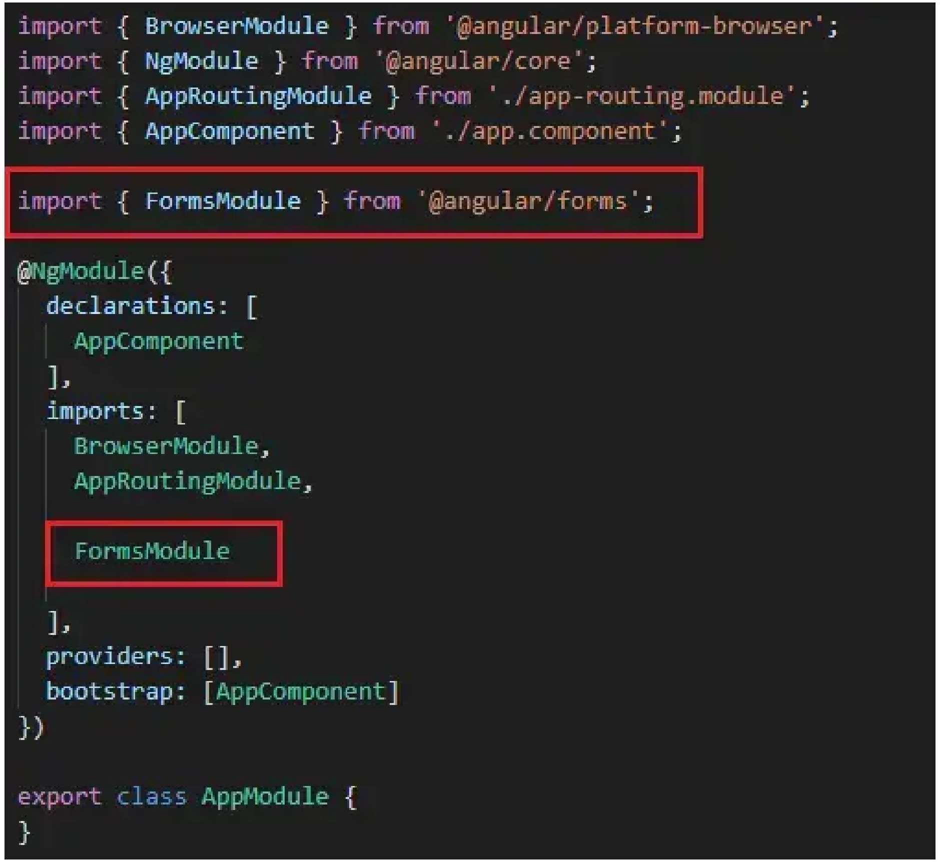
1. Contains little code in the component class
2. Easier to set up

While they are

1. Difficult to add controls dynamically
2. Unit testing is a challenge

How we can develop and use these forms in Angular Application?

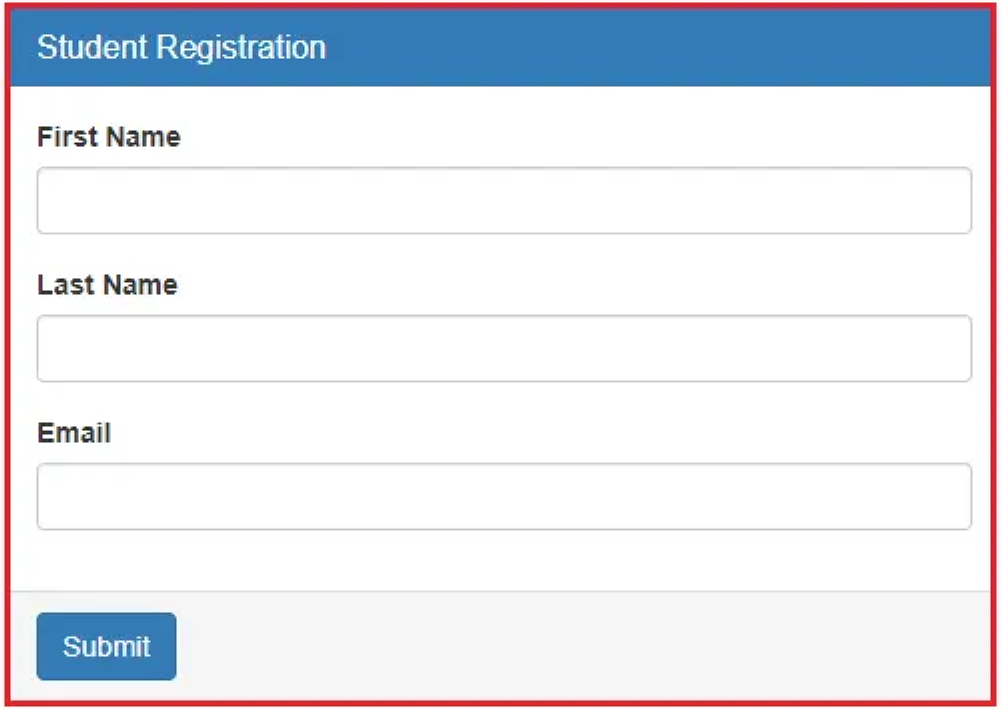
Step1: Importing Forms module



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

How we can develop and use these forms in Angular Application?

Step2: Create a Form



ngForm

- **The ngForm does the following**

1.Binds itself to the <Form> directive

2.Creates a top-level FormGroup instance

3.CreatesFormControl instance for each of child control, which has ngModel directive.

4.CreatesFormGroup instance for each of the **NgModelGroup** directive.

- We can export the ngForm instance into a local template variable using ngForm as the key (ex: #contactForm="ngForm")

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

FormControl

- The FormControl is the basic building block of the Angular Forms.
- It represents a single input field in an Angular form.
- The Angular Forms Module binds the input element to a FormControl.
- We use the FormControl instance to track the value, user interaction and validation status of an individual form element. Each individual Form element is a FormControl
- We need to bind the FormControl instance to each input control using ngModel directive as **<input type="text" name="firstname" ngModel>**
- ngModel will use the name attribute to create the FormControl instance for each of the Form field it is attached.

Submit Form

- We use the ngSubmit event, to submit the form data to the component class.
- We use the event binding (parentheses) to bind ngSubmit to OnSubmit method in the component class.
- When the user clicks on the submit button, the ngSubmit event will fire.

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

- We are passing the local template variable contactForm in onSubmit method.
- contactForm holds the reference to the ngForm directive.
- We can use this in our component class to extract the data from the form fields.

Receive Form Data

- We need to receive the data in component class from our form.
- To do this we need to create a method in our component class.
- The method receives the reference to the ngForm directive.
- Example :

```
onSubmit(contactForm) {
  console.log( contactForm.value );
 }
```

- We have access to the ngForm instance via the local template variable #contactForm.

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

```
<pre>Value : {{contactForm.value | json }} </pre>
<pre>Valid : {{contactForm.valid}} </pre>
<pre>Touched : {{contactForm.touched  }} </pre>
<pre>Submitted : {{contactForm.submitted  }} </pre>
```

- We can also get access to the FormControl instance by assigning the ngModel to a local variable as

<input type="text" name="firstname" #fname="ngModel" ngModel>

```
<pre>Value    : {{fname.value}} </pre>
<pre>valid    : {{fname.valid}} </pre>
<pre>invalid  : {{fname.invalid}} </pre>
<pre>touched  : {{fname.touched}} </pre
```

Nested FormGroup

- The FormGroup is a collection of FormControl. It can also contain other FormGroup's.
- The ngForm directive creates the top Level FormGroup behind the scene, when we use the <Form> directive.
- We can add new FormGroup using the **ngModelGroup** directive.

```
<div ngModelGroup="address">
    <p>
     <label for="city">City</label>
     <input type="text" name="city" ngModel>
   </p>

   <p>
     <label for="street">Street</label>
     <input type="text" name="street" ngModel>
   </p>
   <p>
     <label for="pincode">Pin Code</label>
     <input type="text" name="pincode" ngModel>
   </p>
 </div>
```

Setting the Initial Value in template-driven forms

There are two ways you can set the value of the form elements

- Two-way data binding
- Use the template reference variable

Model class

```
export class contact {
  firstname:string;
  lastname:string;
  email:string;
  gender:string;
  isMarried:boolean;
  country:string;
  address: {
    city:string;
    street:string;
    pincode:string;
  }
}
```

1. Two-way data binding

- The two-way data binding is the recommended way to set the value in the template-driven forms.
- The advantageous here is that any changes made in the form are automatically propagated to the component class and changes made in component class are immediately shown in the form.

```
<label for="firstname">First Name </label>
<input type="text" id="firstname" name="firstname" [(ngModel)]="contact.firstname">
```

Set the default/initial value

```
ngOnInit() {

    this.contact = {
      firstname: "Sachin",
      lastname: "Tendulkar",
      email: "sachin@gmail.com",
      gender: "male",
      isMarried: true,
      country: "2",
      address: { city: "Mumbai", street: "Perry Cross Rd", pincode: "400050" }
    };

  }
```

Set the value individually or dynamically

```
changeCountry() {
  this.contact.country = "1";
}
```

Reset form

```
<button type="button" (click)="reset(contactForm)">Reset</button>

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

2. Template reference variable

- We have a #contactForm reference variable, which is an instance of ngForm

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

- We can get the reference to the #contactForm in the app.component.ts, using the viewchild

```
@ViewChild('contactForm',null) contactForm: NgForm;
```

- Once we have the reference, we can use the **setValue** method of the ngForm to set the initial value

```
setTimeout(() => {
    this.contactForm.setValue(this.contact);
  });
```

Set the value individually or dynamically

- We can also set the value individually using the setValue method of the individual FormControl.
- We shall get the reference to the individual FormControl from the controls collection of the ngForm.
- Once you get the reference use the setValue on the FormControl instance to change the value.

```
changeCountry() {
     this.contactForm.controls["country"].setValue("1");
}
```

Set Default Value

```
this.contactForm.setValue(this.contact);


this.contactForm.resetForm();
```

patch value

- We can make use of the patchValue to change the only few fields anytime.
- The control property of the ngForm returns the reference to the top level FormGroup.

```
patchValue() {
    let obj = {
      firstname: "Rahul",
      lastname: "Dravid",
      email: "rahul@gmail.com",
    };

    this.contactForm.control.patchValue(obj);

  }
```

Set value of nested FormGroup

We can update nested FormGroup by getting a reference to the nested FormGroup from the controls collection of ngForm.

```
changeAddress() {
    let obj = {
      city: "Bangalore",
      street: "Brigade Road",
      pincode: "600100"
    };
    let address= this.contactForm.controls["address"] as FormGroup
    address.patchValue(obj);

  }
```

Template driven form validation in Angular

- Validations in Template-driven forms are provided by the Validation directives.
- The Angular Forms Module comes with several built-in validators.
- We can also create your own custom Validator.

Disabling the Browser validation

First, we need to disable browser validator interfering with the Angular validator.

To do that we need to add novalidate attribute on <form> element as shown below

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

Built-in Validators

- The Built-in validators use the HTML5 validation attributes like **required**, **minlength**, **maxlength** & **pattern**.

- Angular interprets these validation attributes and add the validator functions to FormControl instance.

Adding in Built-in Validators

1. Required Validation

The required validator returns true only if the form control has non-empty value entered.

```
<input type="text" id="firstname" name="firstname" required [(ngModel)]="contact.firstname">
```

2. Minlength Validation

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

```
<input type="text" id="firstname" name="firstname" required
minlength="10" [(ngModel)]="contact.firstname">
```

Adding in Built-in Validators

3. Maxlength Validation

This Validator requires that the number of characters must not exceed the value of the attribute.

```
<input type="text" id="lastname" name="lastname" required maxlength="15"
[(ngModel)]="contact.lastname">
```

4. Pattern Validation

This Validator requires that the control value must match the regex pattern provided in the attribute.

```
<input type="text" id="lastname" name="lastname" required maxlength="15"
    pattern="^[a-zA-Z]+$" [(ngModel)]="contact.lastname">
```

Adding in Built-in Validators

5. Email Validation

This Validator requires that the control value must be a valid email address. We apply this to the email field

```
<input type="text" id="email" name="email" required email
[(ngModel)]="contact.email">
```

Disable Submit button

- We need to disable the submit button if our form is not valid.
- The FormGroup has a valid property, which is set to true if all of its child controls are valid. We use it to set the disabled attribute of the submit button.

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

Displaying the Validation/Error messages

- We need to provide a short and meaningful error message to the user.
- Angular creates a FormControl for each and every field, which has ngModel directive applied.
- The FormControl exposes the state of form element like valid, dirty, touched, etc.
- There are two ways in which you can get the reference to the FormControl.

1. Using Template Reference Variable of Form

   contactForm.controls.firstname.valid

2. Using Template Reference Variable of the Individual Controls

```
<input type="text" id="firstname" name="firstname" required minlength="10"
#firstname="ngModel" [(ngModel)]="contact.firstname">
```

firstname.valid

Displaying the Validation/Error messages

```
<div *ngIf="!firstname?.valid && (firstname?.dirty || firstname?.touched)">
  Invalid First Name
</div>
```

Why check dirty and touched?

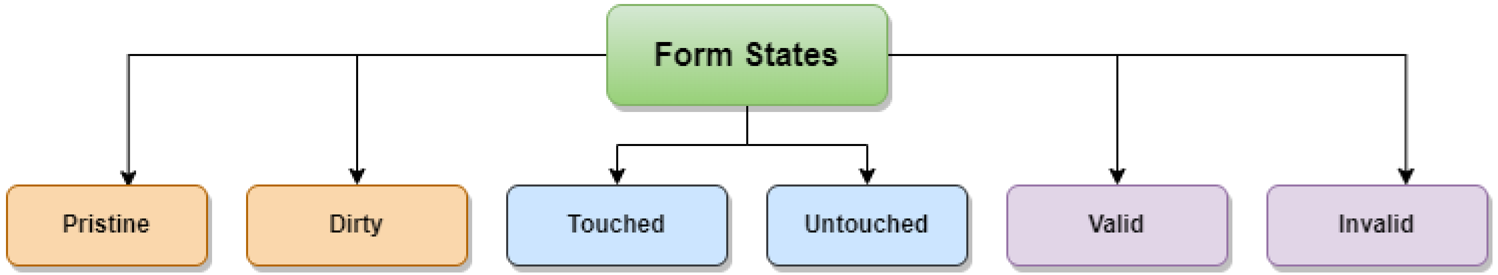
We do not want the application to display the error when the form is displayed for the first time. We want to display errors only after the user has attempted to change the value.

dirty: A control is dirty if the user has changed the value in the UI.

touched: A control is touched if the user has triggered a blur event on it.

Error message

```
<div *ngIf="!firstname?.valid && (firstname?.dirty || firstname?.touched)">
  Invalid First Name
  <div *ngIf="firstname.errors.required">
     First Name is required
  </div>
  <div *ngIf="firstname.errors.minlength">
    First Name Minimum Length is {{firstname.errors.minlength?.requiredLength}}
  </div>
</div>
```



Form States	True when?
Pristine	The user has not modified the form control
Dirty	The user has modified the form control
Touched	The user has interacted with the form control, e.g., by clicking or focusing on it.
Untouched	The form control has not been interacted with by the user.
Valid	The form control's value meets the validation rules defined in the application.
Invalid	The form control's value does not meet the validation rules defined in the application.

Thank You