two ways to deal with Forms in Angular 17

Angular Forms

> - Template-driven forms
> - Reactive forms

u can't use `[(ngModel)]="binding"` inside `form` tag in Angular 17

Reactive forms

- Reactive forms are more robust: they're more scalable, reusable, and testable. If forms are a key part of your application, or you're already using reactive patterns for building your application, use reactive forms.

```
    <!— reactive-form.component.ts—>
import { Component } from '@angular/core';
import { FormControl, ReactiveFormsModule } from '@angular/forms';

@Component({
  selector: 'app-reactive-form',
  standalone: true,
  imports: [ReactiveFormsModule],
  templateUrl: './reactive-form.component.html',
  styleUrl: './reactive-form.component.css'
})
export class ReactiveFormComponent {
   test = new FormControl('intial Value');
  constructor() {

  }

}
```

```
<!— reactive-form.component.html —>
    <label for="name">Name: </label>
    <input id="name" type="text" [formControl]="test">
    <p>Value: {{ test.value }}</p>
```

You can display the value in the following ways

- Through the `valueChanges` observable where you can listen for changes in the form's value in the template using `AsyncPipe` or in the component class using the `subscribe()` method
- With the value property, which gives you a snapshot of the current value

```
    <!— reactive-form.component.ts—>
    import {AsyncPipe} from '@angular/common';
    ```
``` html
<!— reactive-form.component.html —>
    <label for="name">Name: </label>
    <input id="name" type="text" [formControl]="test">
   <p>Value: {{ hema.valueChanges | async }}</p>
```

if u will suscribe on value changes that must be inside ngOnInit or the constructor

```
this.hema.valueChanges.subscribe(value => {
     console.log(value);
  });
// the above code will make an error
```

the below code will work

```
    <!— reactive-form.component.ts—>
    import {OnInit} from '@angular/core';
    ```
``` ts
    <!— reactive-form.component.ts—>
    export class ReactiveFormComponent implements OnInit {
    ngOnInit() {
        this.test.valueChanges.subscribe(value => {
            console.log(value);
        });
    }
}
```

Tips

Replacing a form control value

- You can replace the value of a form control using the setValue() method. This method takes the new value as an argument and updates the form control's value.

```
this.test.setValue('new value');
```

Resetting a form control value

- You can reset a form control value to its initial value using the reset() method.

```
this.test.reset();
```

Disabling a form control

- You can disable a form control using the disable() method.

```
this.test.disable();
```

Enabling a form control

- You can enable a form control using the enable() method.

```
this.test.enable();
```

When using the `setValue()` method with a `form group` or `form array` instance, the value needs to match the structure of the group or array.

Grouping form controls

- Forms typically contain several related controls.

> ⚠ **Reactive forms provide two ways of grouping multiple related controls into a single input form.**
>
> - Form groups
> - Form arrays

Note

Form groups

- Defines a form with a fixed set of controls that you can manage together. You can also nest form groups to create more complex forms.

  > ⚠ **import the FormGroup and FormControl classes from the @angular/forms package.**
  >
  > ```
  > import { FormGroup, FormControl } from '@angular/forms';
  > ```

- Create a form group instance by passing an object to the `#FormGroup` constructor. The object keys are the control names, and the values are the form controls.

```
import { Component } from '@angular/core';
import { FormControl, FormGroup, ReactiveFormsModule } from
'@angular/forms';
@Component({
selector: 'app-reactive-form',
standalone: true,
imports: [ReactiveFormsModule],
templateUrl: './reactive-form.component.html',
styleUrl: './reactive-form.component.css'
})
export class ReactiveFormComponent {
 constructor() {
 }
newForm = new FormGroup({
```

```
  name: new FormControl(''),
   email: new FormControl(''),
   password: new FormControl(''),
   confirmPassword: new FormControl('')
  });
  onSubmit() {
    console.log(this.newForm.value);
  }
  }
```

- In the template, bind the form group instance to the form element using the formGroup directive `#formControlName` .

```
<!— reactive-form.component.html —>
<form [formGroup]="newForm">
    <div class="form-group">
        <label for="name">Name</label>
        <input type="text" class="form-control" formControlName="name"
id="name">
    </div>
    <div class="form-group">
        <label for="email">Email</label>
        <input type="email" class="form-control" formControlName="email"
id="email">
    </div>
    <div class="form-group">
        <label for="password">Password</label>
        <input type="password" class="form-control" formControlName="password"
id="password">
    </div>
    <div class="form-group">
        <label for="confirmPassword">Confirm Password</label>
        <input type="password" class="form-control"
formControlName="confirmPassword" id="confirmPassword">
    </div>
    <form [formGroup]="newForm" (ngSubmit)="onSubmit()">
</form>
```

⚡ to make submit button disabled until the validation done

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

Creating nested form groups

- Form groups can accept both individual form control instances and other form group instances as `children`. This makes composing complex form models easier to maintain and logically group together.
- When building complex forms, managing the different areas of information is easier in smaller sections. Using a nested form group instance lets you break large forms groups into smaller, more manageable ones.

> **Warning**
> - To make more complex forms, use the following steps.
>   - Create a nested group.
>   - Group the nested form in the template.
>   - In the template, bind the nested form group instance to the form element using the formGroupName directive.

```
newForm = new FormGroup({
   name: new FormControl(''),
   email: new FormControl(''),
   password: new FormControl(''),
   confirmPassword: new FormControl(''),
   address: new FormGroup({
     street: new FormControl(''),
     city: new FormControl(''),
     state: new FormControl(''),
     zip: new FormControl(''),
   }),
});
```

```
<!— reactive-form.component.html —>
 <div formGroupName="address">
        <h2>Address</h2>
        <label for="street">Street: </label>
        <input id="street" type="text" formControlName="street">
        <label for="city">City: </label>
        <input id="city" type="text" formControlName="city">
        <label for="state">State: </label>
        <input id="state" type="text" formControlName="state">
        <label for="zip">Zip Code: </label>
```

```
    <input id="zip" type="text" formControlName="zip">
  </div>
```

Updating parts of the data model

There are two ways to update the model value:

- Using the setValue() method
- The `setValue()` Set a new value for an individual control. The `setValue()` method strictly adheres to the structure of the form group and replaces the entire value for the control
- Using the patchValue() method
- The `patchValue()` Replace any properties defined in the object that have changed in the form model.

The strict checks of the setValue() method help catch nesting errors in complex forms `[u should edit all values]`, while patchValue() fails silently on those errors `[u can edit just some values]`.

```
this.profileForm.patchValue({
  firstName: 'Nancy',
  address: {
    street: '123 Drew Street',
  },
});
```

Using the FormBuilder service to generate controls

- Creating form control instances manually can become repetitive when dealing with multiple forms. The FormBuilder service provides convenient methods for generating controls.
- Use the following steps to take advantage of this service.
    1. Import the FormBuilder class.
    2. Inject the FormBuilder service.

3. Generate the form contents.

```
 //- Import the FormBuilder class
 import { AsyncPipe, JsonPipe } from '@angular/common';
  import { FormBuilder } from '@angular/forms';
//- Inject the FormBuilder service
constructor(private fb: FormBuilder) { }
//- Generate the form contents
export class ReactiveFormComponent {
  constructor(private form: FormBuilder) { }
  hema =  this.form.group({
    userName: [''],
    password: [''],
  })
  onSubmit(){
    console.log(this.hema.value);
  }
}
```

```
<!— reactive-form.component.html —>
<form [formGroup]="hema" (ngSubmit)="onSubmit()">
    <div class="form-group">
        <label for="userName">User Name</label>
        <input type="text" class="form-control" formControlName="userName"
id="userName">
    </div>
    <div class="form-group">
        <label for="password">Password</label>
        <input type="password" class="form-control" formControlName="password"
id="password">
    </div>
    <button type="submit">Submit</button>
</form>
```

⚠ The FormBuilder service provides the following methods:

- `group()` : Creates a form group.
- `control()` : Creates a form control.
- `array()` : Creates a form array.

how to use Form Builder

```
//- Generate form controls
import {Component} from '@angular/core';
import {FormBuilder} from '@angular/forms';
...
@Component({
  selector: 'app-profile-editor',
  templateUrl: './profile-editor.component.html',
  styleUrls: ['./profile-editor.component.css'],
})
export class ProfileEditorComponent {
 this.formBuilder.group({
      name: '',
      email: '',
      password: '',
      confirmPassword: '',
      address: this.formBuilder.group({
        street: '',
        city: '',
        state: '',
        zip: ''
      })
    });
...
  });
...
  constructor(private formBuilder: FormBuilder) {}
...
}
```

Hint

- In the preceding example, you use the `group()` method with the same object to define the properties in the model. The value for each control name is an array containing the initial value as the first item in the array.

⚠ Warning

- You can define the control with just the initial value, but if your controls need sync or async validation, add sync and async validators as the second and third items in the array. Compare using the form builder to creating the instances manually

Validating form input

⚠ **In a reactive form, the source of truth is the component class. Instead of adding validators through attributes in the template, you add validator functions directly to the form control model in the component class. Angular then calls these functions whenever the value of the control changes.**

- Use the following steps to add form validation.
  1. Import a validator function in your form component.
  2. Add the validator to the field in the form.
  3. Add logic to handle the validation status

```
//- Import the Validators class from the @angular/forms package.
import { Validators } from '@angular/forms';
```

```
 //- Make a field required
 profileForm = this.formBuilder.group({
       userName: [''',
 [Validators.required, Validators.minLength(3), Validators.pattern('^[a-
zA-Z]+$')]
],
password: ['']
   }),
 ...
});
```

Danger

- When you add a required field to the form control, its initial status is invalid. This invalid status propagates to the parent form group element, making its status invalid.

Access the current status of the form group instance through its status property.

```
<p>Form Status: {{ profileForm.status }}</p>
```

the above code will make an error

- ***ERROR Error: NG01101: Expected async validator to return Promise or Observable. Are you using a synchronous validator where an async validator is expected?***

More on form validation with reactive forms

- Reactive forms provide synchronous and asynchronous validators. Synchronous validators run when the value of the control changes. Asynchronous validators run when the value of the control changes and return a promise or observable.

> **For performance reasons, Angular only runs async validators if all sync validators pass. Each must complete before errors are set.**

- You can choose to write your own validator functions, or you can use some of Angular's built-in validators.
- Angular provides a set of built-in validators that you can use to validate form input. These validators are available as functions in the Validators class.
- The Validators class provides the following built-in validators:
  - `#required` : Requires the control to have a non-empty value.
  - `#email` : Requires the control to have a valid email address.
  - `#minLength` : Requires the control to have a minimum length.
  - `#maxLength` : Requires the control to have a maximum length.
  - `#pattern` : Requires the control to match a regular expression.
  - `#nullValidator` : Always passes.
  - `#compose` : Combines multiple validators into a single function.
- u can use custom validators by creating a function that returns a validator function.

Example of using built-in validators (pattern) and custom validators

```
//- Import the Validators class from the @angular/forms package.
import { Validators } from '@angular/forms';
...
//- Add the pattern validator to the form control.
ngOnInit(): void {
    this.actorForm = new FormGroup({
      name: new FormControl(this.actor.name, [
        Validators.required,
        Vaidators.pattern('^[a-zA-Z]+$'),// <-- Here's how you add the pattern
validator.
        Validators.minLength(4),
        forbiddenNameValidator(/bob/i), // <-- Here's how you pass in the
custom validator.
      ]),
      role: new FormControl(this.actor.role),
      skill: new FormControl(this.actor.skill, Validators.required),
    });
  }
  get name() {
    return this.actorForm.get('name');
  }
  get skill() {
    return this.actorForm.get('skill');
  }
...
```

The pattern validator uses a regular expression to validate the form control. In this example, the pattern validator checks that the value of the control contains only letters. If the value doesn't match the pattern, the control is invalid.

```
<!-- reactive-form.component.html -->
<input type="text" id="name" class="form-control"
                formControlName="name" required>
          <div *ngIf="name.invalid && (name.dirty || name.touched)"
              class="alert alert-danger">
            <div *ngIf="name.hasError('required')">
              Name is required.
            </div>
```

```
      <div *ngIf="name.hasError('minlength')">
        Name must be at least 4 characters long.
      </div>
      <div *ngIf="name.hasError('forbiddenName')">
        Name cannot be Bob.
      </div>
    </div>
```

Creating a custom validator

- To create a custom validator, you need to create a function that returns a validator function. The validator function takes a form control as an argument and returns an object if the validation fails.
- The following example shows how to create a custom validator that checks if the value of the form control is Bob.

```
//- Create a function that returns a validator function.
export function forbiddenNameValidator(nameRe: RegExp): ValidatorFn {
return (control: AbstractControl): {[key: string]: any} | null => {
 const forbidden = nameRe.test(control.value);
 return forbidden ? {forbiddenName: {value: control.value}} : null;
};
}
...
```

Control status CSS classes

Angular automatically mirrors many control properties onto the form control element as CSS classes. Use these classes to style form control elements according to the state of the form. The following classes are currently supported.

- `.ng-valid`: The control has passed all validation checks.
- `.ng-invalid`: The control has failed one or more validation checks.
- `.ng-pending`: The control is in the process of conducting validation checks.
- `.ng-pristine`: The control has not been touched.

- `.ng-dirty` : The control has been touched.
- `.ng-touched` : The control has been touched.
- `.ng-untouched` : The control has not been touched.
  example:

```
input.ng-invalid.ng-touched {
 border: 2px solid red;
}
.ng-valid[required], .ng-valid.required  {
  border-left: 5px solid #42A948; /* green */
}
```

Cross-field validation

- Cross-field validation is a type of validation that requires comparing the values of multiple form controls. You can use the form group instance to compare the values of the controls so that if the user can choose A or B, but not both. Some field values might also depend on others; a user might be allowed to choose B only if A is also chosen..
- The examples use cross-validation to ensure that actors do not reuse the same name in their role by filling out the Actor Form. The validators do this by checking that the actor names and roles do not match.

```
<!— > form   sturcture —>
const actorForm = new FormGroup({
  'name': new FormControl(),
  'role': new FormControl(),
  'skill': new FormControl()
});
```

⚠ **Notice that the `#name` and `#role` are sibling controls. To evaluate both controls in a single custom validator, you must perform the validation in a common ancestor control: the FormGroup. You query the FormGroup for its child controls so that you can compare their values.**

To add a validator to the FormGroup, pass the new validator in as the second argument on creation.

```
const actorForm = new FormGroup({
  'name': new FormControl(),
  'role': new FormControl(),
  'skill': new FormControl()
}, { validators: unambiguousRoleValidator });

<!— Custom Validator —>
/** An actor's name can't match the actor's role */
export const unambiguousRoleValidator: ValidatorFn = (
  control: AbstractControl,
): ValidationErrors | null => {
  const name = control.get('name');
  const role = control.get('role');
  return name && role && name.value === role.value ? {unambiguousRole:
true} : null;
};
```

Notes

- The `#unambiguousRoleValidator` validator implements the `ValidatorFn interface`. It takes an Angular control object as an argument and returns either null if the form is valid, or ValidationErrors otherwise.
- The validator retrieves the child controls by calling the FormGroup's `#get` method, then compares the values of the name and role controls.
- If the values do not match, the role is unambiguous, both are valid, and the validator returns null. If they do match, the actor's role is ambiguous and the validator must mark the form as invalid by returning an error object.

To provide better user experience, the template shows an appropriate error message when the form is invalid.

```
<!— reactive-form.component.html —>
<div *ngIf="actorForm.hasError('unambiguousRole') && (actorForm.touched ||
actorForm.dirty)" class="cross-validation-error-message alert alert-
danger">
```

```
            Name cannot match role or audiences will be confused.
</div>
```

Creating asynchronous validators

- Asynchronous validators are useful when you need to validate a form control against a server. For example, you might want to check if a username is already taken. Angular provides the `AsyncValidatorFn` interface to create asynchronous validators.
- very similar to the synchronous validators, but instead of returning a `ValidationErrors` object, they return a `Promise` or `Observable` that resolves to a `ValidationErrors` object.
- not recommended to use async validators for synchronous validation tasks, as they can slow down the user experience So search for it if u search for trouble.

Note

`Form arrays`

Creating dynamic forms

- #FormArray is an alternative to #FormGroup for managing any number of unnamed controls. As with form group instances, you can dynamically insert and remove controls from form array instances, and the form array instance value and validation status is calculated from its child controls. However, you don't need to define a key for each control by name, so this is a great option if you don't know the number of child values in advance.

To define a dynamic form, take the following steps.

1. Import the FormArray class.
2. Define a FormArray control.
3. Access the FormArray control with a getter method.

4. Display the form array in a template.

```
// -Import the FormArray class
import { FormArray } from '@angular/forms';
//- Define a FormArray control
```

⚠ Warning

- Use the FormBuilder.array() method to define the array, and the FormBuilder.control() method to populate the array with an initial control.

```
// - Define a FormArray control
/* - You can initialize a form array with any number of controls, from
zero to many, by defining them in an array. Add an aliases property to the
form group instance for profileForm to define the form array.
*/
   profileForm = this.formBuilder.group({
       firstName: ['', Validators.required],
       lastName: [''],
       address: this.formBuilder.group({
         street: [''],
         city: [''],
         state: [''],
         zip: [''],
       }),
       aliases: this.formBuilder.array([this.formBuilder.control('')]),
     });
// - Access the FormArray control with a getter method
// - A getter provides access to the aliases in the form array instance
compared to repeating the profileForm.get() method to get each instance.
The form array instance represents an undefined number of controls in an
array. It's convenient to access a control through a getter, and this
approach is straightforward to repeat for additional controls.
<!--Use the getter syntax to create an aliases class property to retrieve
the alias's form array control from the parent form group.-->
get aliases() {
   return this.profileForm.get('aliases') as FormArray;
  }
// -  Because the returned control is of the type AbstractControl, you
need to provide an explicit type to access the method syntax for the form
array instance. Define a method to dynamically insert an alias control
into the alias's form array. The FormArray.push() method inserts the
control as a new item in the array.
```

```
  addAlias() {
     this.aliases.push(this.formBuilder.control(''));
   }
  // - Display the form array in a template
  // - Use the form array instance in the template to display the form array
  controls. The form array instance is an array of form controls, so you can
  use the form array instance in an ngFor directive to iterate over the
  controls.
```

```
> <!— reactive-form.component.html —>
<div formArrayName="aliases">
   <h2>Aliases</h2>
   <button type="button" (click)="addAlias()">+ Add another alias</button>
   <div *ngFor="let alias of aliases.controls; let i=index">
     <!— The repeated alias template —>
     <label for="alias-{{ i }}">Alias:</label>
     <input id="alias-{{ i }}" type="text" [formControlName]="i">
   </div>
 </div>
```

Note

- The `*ngFor` directive iterates over each form control instance provided by the aliases form array instance. Because form array elements are unnamed, you assign the index to the i variable and pass it to each control to bind it to the `formControlName` input.
- Each time a new alias instance is added, the new form array instance is provided its control based on the index. This lets you track each individual control when calculating the status and value of the root control.
- Initially, the form contains one Alias field.
- To add another field, click the Add `Alias` button. You can also validate the array of aliases reported by the form model displayed by Form Value at the bottom of the template. Instead of a form control instance for each alias, you can compose another form group instance with additional fields. The process of defining a control for each item is the same.