

Contents

Reactive Forms.....	2
1. Add the HTML Form	2
2. Create the form model	3
3. Nesting Form Groups.....	4
4. Managing Control Values	5
5. FormBuilder Service	6
6. Simple Validation	7
7. Custom Validation	9
8. Cross-field Validation.....	11
9. Conditional Validation	13
10. Dynamic Form Controls	16
11. Submitting Form Data.....	18

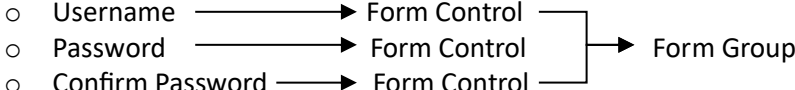
Reactive Forms

- Code and logic reside in the component class.
- No two-way binding.
- Well suited for complex scenarios like:
 - Dynamic form fields.
 - Custom Validation.
 - Dynamic Validation.

1. Add the HTML Form

```
<div class="container-fluid">
  <h2>Registration Form</h2>
  <form>
    <div class="form-group">
      <label>Username</label>
      <input type="text" class="form-control">
    </div>
    <div class="form-group">
      <label>Password</label>
      <input type="password" class="form-control">
    </div>
    <div class="form-group">
      <label>Confirm Password</label>
      <input type="password" class="form-control">
    </div>
    <button class="btn btn-primary" type="submit">Register</button>
  </form>
</div>
```

2. Create the form model

- First step `import ReactiveFormsModule in import array section in app.module.ts`
 - In Reactive forms the form is represented by a model in the component class and to able to create that model we make use of `form group and form control` classes.
 - In our form we have 3 fields
 - Username → Form Control
 - Password → Form Control
 - Confirm Password → Form Control
- 
- These fields are defined as an instance of the form control and the overall form itself that encompasses the three form fields is defines as an instance of the form group class.
 - Lets create a form model

```
registrationForm = new FormGroup({
  userName: new FormControl('Mohnish'),
  password: new FormControl(''),
  confirmPassword: new FormControl('')
});
```

- Next, we need to associate this model with the view which is our HTML form.
- So to associate the form model with the HTML form angular provide few directives like.
 - `[formGroup]` directive to bind the formgroup instance from our component class

```
<form [formGroup]="registrationForm">
```

- `formControlName` directive used to bind each of the form controls.

```
<div class="form-group">
  <label>Username</label>
  <input type="text" class="form-control" formControlName =
    "userName">
</div>
```

- So what we have done is 1-1 association between the form group, form controls and there corresponding HTML elements.
- To visualize the communication let's use interpolation on from with json file.

```
{{registrationForm.value | json}}
```

3. Nesting Form Groups

- We learned that form control class is used for individual form elements and the form group class is used to represent the entire form.
- Apart from that the form group class can also be used to group together different form controls for example city, state, postcode can be represented as one form group called address.
- Lets code,
 - First we will be adding the 3 new fields
 - City,
 - State,
 - Postal Code
 - All 3 will be enclosed in a div tag.

```
<div>
  <div class="form-group">
    <label>City</label>
    <input type="text" class="form-control">
  </div>
  <div class="form-group">
    <label>State</label>
    <input type="text" class="form-control">
  </div>
  <div class="form-group">
    <label>Postal code</label>
    <input type="text" class="form-control">
  </div>
</div>
```

- Now let's update our form model we need to add 3 new instances of the form control class but we need to group them into an address field. To achieve this group we need to use form group class.

```
registrationForm = new FormGroup({
  userName: new FormControl('Mohnish'),
  password: new FormControl(''),
  confirmPassword: new FormControl(''),
  address: new FormGroup({
    city: new FormControl(''),
    state: new FormControl(''),
    postalCode: new FormControl(''),
  })
});
```

- Now lets update the HTML to bind the right control.

```
<div formGroupName="address">
  <div class="form-group">
    <label>City</label>
    <input type="text" class="form-control" formControlName =
"city">
  </div>
  <div class="form-group">
    <label>State</label>
    <input type="text" class="form-control" formControlName =
"state">
  </div>
  <div class="form-group">
    <label>Postal code</label>
    <input type="text" class="form-control" formControlName =
"postalCode">
  </div>
</div>
```

4. Managing Control Values

- To set the value of the form which is been retrieved from database programmatically we need to use **setValue method** provided by reactiveForms.
- Let's code
 - We will be keeping a button In the HTML class "Load API data" when we click this method we will invoke click event which calls a function "loadAPIData()".

```
<button class="btn btn-secondary" type="button"
(click)="loadAPIData()">Load API Data</button>
```

- Now, we will define the "loadAPIData()" method.

```
loadAPIData(){
  this.registrationForm.setValue({
    userName: 'Barry',
    password: 'passCode',
    confirmPassword: 'passCode',
    address: {
      city: 'city',
      state: 'state',
      postalCode: '546789',
    },
  });
}
```

- When the "Load API Data" button is clicked It will call the "loadAPIData()" method which contain code to setvalue to the controls present in the HTML form.

- Now, if we want to only setValue to the username, password and confirm password and not to the address we will be removing the address key from set

```
this.registrationForm.setValue({
  userName: 'Barry',
  password: 'passCode',
  confirmPassword: 'passCode'
});
```

- But we will be having a error and the code doesn't work because setValue method requires all the controls which we are using in our form.
- So to overcome this issue instead of using setValue we can use patchValue method.

```
this.registrationForm.patchValue({
  userName: 'Barry',
  password: 'passCode',
  confirmPassword: 'passCode'
});
```

5. FormBuilder Service

- Creating multiple form controls manually can become very repetitive.
- To avoid this angular provide FormBuilder service which in turn provides methods to handle generating form controls.
- Lets code
 - As I said FormBuilder is a service we need to import it and inject in constructor and the constructor is in our class in our forms component .ts file

```
import { FormBuilder, FormControl, FormGroup } from
'@angular/forms';
```

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

- Second step is use the form builder service to generate form controls.

```
registrationForm = this.formBuilder.group({
  userName: ['Mohnish'],
  password: [''],
  confirmPassword: [''],
  address: this.formBuilder.group({
    city: [''],
    state: [''],
    postalCode: ['']
  })
})
```

6. Simple Validation

- First step, applying validation rule

```
import { FormBuilder, FormControl, FormGroup, Validators } from '@angular/forms';
```

```
registrationForm = this.formBuilder.group({
  userName: ['Mohnish', Validators.required],
  password: [''],
  confirmPassword: [''],
  address: this.formBuilder.group({
    city: [''],
    state: [''],
    postalCode: ['']
  })
});
```

- Second step is to provide visual feedback that there is a validation error.

```
<input type="text" class="form-control" formControlName = "userName"
[class.is-invalid]="registrationForm.get('userName')?.invalid">
```

- Third step display appropriate error messages.

```
<input type="text" class="form-control" formControlName = "userName"
[class.is-invalid]="registrationForm.get('userName')?.invalid">
<small class="text-danger" [class.d-none]="registrationForm.get('userName')?.valid">Username is
required</small>
```

- How to keep multiple validation rules
 - Let's apply **min-length validation** to username

```
registrationForm = this.formBuilder.group({
  userName: ['Mohnish', [Validators.required,
Validators.minLength(3)]],
  password: [''],
  confirmPassword: [''],
  address: this.formBuilder.group({
    city: [''],
    state: [''],
    postalCode: ['']
  })
});
```

- Now let's show appropriated error message

```
<input type="text" class="form-control" formControlName =
"userName" [class.is-
invalid]="registrationForm.get('userName')?.invalid">
  <!-- <small class="text-danger" [class.d-
none]="registrationForm.get('userName')?.valid">Username is
required</small> -->
  <div
*ngIf="registrationForm.get('userName')?.errors?.['required']"
class="invalid-feedback">
    Username is required
  </div>
  <div
*ngIf="registrationForm.get('userName')?.errors?.['minlength']"
" class="invalid-feedback">
    Username must be at least 3 characters long
  </div>
```

- We can minimize the above code by using getter method

- In forms ts file

```
get registrationFormControl() {return
this.registrationForm.controls}
```

- In forms html file

```
<input type="text" class="form-control" formControlName
= "userName" [class.is-
invalid]="registrationForm.get('userName')?.invalid">
  <!-- <small class="text-danger" [class.d-
none]="registrationForm.get('userName')?.valid">Username
is required</small> -->
  <div
*ngIf="registrationFormControl.userName?.errors?.['requi
red']" class="invalid-feedback">
    Username is required
  </div>
  <div
*ngIf="registrationFormControl.userName?.errors?.['minle
ngth']" class="invalid-feedback">
    Username must be at least 3 characters long
  </div>
```


7. Custom Validation

- The built-in validators won't always match the exact use case of your application.
- In this scenario we can create own custom validators.
- For example, when user enter "admin" in "username field" we have to show validation message "admin username is not allowed".
- Custom validator is nothing but a function which can be written in the ts file. But to re-use the validator functions we can keep these validators in separate file and import these files when we needed.
- Let's code
 - First create a folder share within the app folder.

- In this shared folder create file "username.validator.ts" and define validator function.

```
import { AbstractControl } from "@angular/forms";

export function forbiddenNameValidator(control:
AbstractControl): {[key:string]:any} | null{
    let forbidden = /admin/.test(control.value);
    return forbidden ? {'forbiddenName': {value:
control.value}} : null;
}
```

- Add the custom validator in form ts file.

```
registrationForm = this.formBuilder.group({
    userName: ['Mohnish', [Validators.required,
Validators.minLength(3), forbiddenNameValidator]],
    password: [''],
    confirmPassword: [''],
    address: this.formBuilder.group({
        city: [''],
        state: [''],
        postalCode: ['']
    })
})
```

import the forbiddenNameValidator as well.

- Add the new validator error message in the HTML form

```
<input type="text" class="form-control" formControlName
= "userName" [class.is-
invalid]="registrationForm.get('userName')?.invalid">
  <!-- <small class="text-danger" [class.d-
none]="registrationForm.get('userName')?.valid">Username
is required</small> -->
  <div
*ngIf="registrationFormControl.userName?.errors?.['requi
red']" class="invalid-feedback">
    Username is required
  </div>
  <div
*ngIf="registrationFormControl.userName?.errors?.['minle
ngth']" class="invalid-feedback">
    Username must be at least 3 characters long
  </div>
  <div
*ngIf="registrationFormControl.userName?.errors?.['forbi
ddenName']" class="invalid-feedback">
    {{registrationFormControl.userName.errors?.['for
biddenName'].value}} Username not allowed
  </div>
```

- It is also possible to pass parameters to custom validators.
- For Examples we are forbidding the string admin in the username but there could be another value that forbids string password.
- So, we should be able to pass in the string we want to forbid as a parameter to our custom validator.
- Let's code
 - username.validator.ts

```
export function forbiddenNameValidator(forbiddenName: RegExp):
ValidatorFn{
  return (control: AbstractControl): {[key:string]:any} |
null => {
    let forbidden = forbiddenName.test(control.value);
    return forbidden ? {'forbiddenName': {value:
control.value}} : null;
  }
}
```

- forms ts file

```
registrationForm = this.formBuilder.group({
  userName: ['Mohnish', [Validators.required,
Validators.minLength(3), forbiddenNameValidator(/password/)]],
  password: [''],
  confirmPassword: [''],
  address: this.formBuilder.group({
    city: [''],
    state: [''],
    postalCode: ['']
  })
})
```

8. Cross-field Validation

- To compare values across two different form controls to perform necessary validation.
- Example: We need to confirm confirm password must be same as password.
- Let's code
 - Create new file "password.validator.ts" within the shared folder.

```
import { AbstractControl } from "@angular/forms";

export function PasswordValidator(control: AbstractControl):
{[key:string]: boolean} | null{
  const password = control.get('password');
  const confirmPassword = control.get('confirmPassword');
  return password && confirmPassword && password.value !=
confirmPassword .value ? {'misMatch': true}:null;
}
```

- In form .ts file

```
registrationForm = this.formBuilder.group({
  userName: ['Mohnish', [Validators.required,
Validators.minLength(3), forbiddenNameValidator(/password/)]],
  password: [''],
  confirmPassword: ['',],
  address: this.formBuilder.group({
    city: [''],
    state: [''],
    postalCode: ['']
  })
}, {validators: PasswordValidator})
```

- In html file

```
<input type="password" class="form-control"
formControlName="confirmPassword" [class.is-
invalid]="registrationForm.errors?.['misMatch']">
  <div *ngIf="registrationForm.errors?.['misMatch']"
class="text-danger">
    Password do not match
  </div>
```

- But if we see the out put when ever we enter the password the confirm validation is getting invoked which not good practice. Hence we need to show confirm validation when the confirm password field has a data.
- So, to resolve this we can do in 2 ways
 - In validator function
 - In html
- I would prefer to do in validator function as the complete validation rules will be in one place and the HTML code will also be clean and simple

```
import { AbstractControl } from "@angular/forms";

export function PasswordValidator(control: AbstractControl):
{[key:string]: boolean} | null{
  const password = control.get('password');
  const confirmPassword = control.get('confirmPassword');
  if(password?.pristine || confirmPassword?.pristine){
    return null;
  }
  return password && confirmPassword && password.value !=
confirmPassword .value ? {'misMatch': true}:null;
}
```

9. Conditional Validation

- At time you may want dynamic validations i.e a particular validation should be applied only under certain conditions.
- Example: If we have an email field and a send promotional offer check box field then when the check box is checked then the email field must have required validation if not there must be no required validation.
- So, lets keep email and checkbox fields

```
<div class="form-group">
  <label>Email</label>
  <input type="email" class="form-control"
formControlName="email">
  <br>
  <div class="form-group">
    <input type="checkbox"
formControlName="subscribe">&nbsp;<label>Send me promotion
offers</label>
  </div>
```

- Now let's implement conditional validation in ts file

```
import { Component, OnInit } from '@angular/core';
import { FormBuilder, FormControl, FormGroup, Validators } from
 '@angular/forms';
import { forbiddenNameValidator } from
 './shared/username.validator';
import { PasswordValidator } from './shared/password.validator';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit{
  constructor(
    private formBuilder: FormBuilder
  ){}

  registrationForm!: FormGroup;

  ngOnInit(): void {
    this.registrationForm = this.formBuilder.group({
      userName: ['Mohnish', [Validators.required,
Validators.minLength(3), forbiddenNameValidator(/password/)]],
      password: ['', ],
      confirmPassword: ['', ],
      email: ['', ],
      subscribe: [false],
      address: this.formBuilder.group({
```

```

        city: [''],
        state: [''],
        postalCode: ['']
    })
    }, {validators: PasswordValidator})

    this.registrationForm.get('subscribe')?.valueChanges.subscribe((
value) => {
        const emailControl = this.registrationForm.get('email');
        if (value) {
            emailControl?.setValidators([Validators.required,
Validators.email]); // Add validators
        } else {
            emailControl?.clearValidators(); // Clear validators
        }
        emailControl?.updateValueAndValidity(); // Update control
        validity
    });
}

get registrationFormControl() {return
this.registrationForm.controls}

// registrationForm = new FormGroup({
//   userName: new FormControl('Mohnish'),
//   password: new FormControl(''),
//   confirmPassword: new FormControl(''),
//   address: new FormGroup({
//     city: new FormControl(''),
//     state: new FormControl(''),
//     postalCode: new FormControl(''),
//   }),
// });

loadAPIData(){
    this.registrationForm.patchValue({
        userName: 'Barry',
        password: 'passCode',
        confirmPassword: 'passCode'
    });
}
}
}

```

- We've added a checkbox named subscribe to the form group.
- We subscribe to the valueChanges of the checkbox.
- When the value of the checkbox changes, we either add the required and email validators to the email control or clear its validators based on the checkbox value.

- Finally, we call `updateValueAndValidity()` to recompute the control's value and validation status.
- Why `valueChanges`
 - The `valueChanges` observable is used in Angular reactive forms to listen for changes in the value of form controls. It's a powerful feature that allows you to react to user input in real-time and perform actions accordingly.
 - Here's why we use `valueChanges`:
 - Real-time validation
 - Dynamic behavior
 - Asynchronous operations
 - Complex form interactions
- Now let's visualize the validation message in html

```
<div class="form-group">
  <label>Email</label>
  <input type="email" class="form-control"
formControlName="email" [class.is-
invalid]="registrationFormControl['email'].invalid">
  <div *ngIf="registrationFormControl['email'].invalid"
class="text-danger">
    Email is required</div>
</div>
<br>
<div class="form-group">
  <input type="checkbox"
formControlName="subscribe">&nbsp;<label>Send me promotion
offers</label>
</div>
```

10. Dynamic Form Controls

- If user have to enter multiple data for single form control instead of entering multiple data in a single form control user can dynamically add new form control and enter data.
- For Example: Let's keep a add email button beside email label so when ever this button is clicked there will be a new input tag so that user can enter multiple emails.
- Let's code
 - First step is to import form array class from angular/forms.
 - Form array class makes it possible to maintain a dynamic list of controls.

```
import { FormBuilder, FormControl, FormGroup, Validators,
FormArray } from '@angular/forms';
```

- Second step is to define a form array in form model

```
this.registrationForm = this.formBuilder.group({
  userName: ['Mohnish', [Validators.required,
Validators.minLength(3), forbiddenNameValidator(/password/)]],
  password: [''],
  confirmPassword: [''],
  email: [''],
  subscribe: [false],
  address: this.formBuilder.group({
    city: [''],
    state: [''],
    postalCode: ['']
  }),
  alternateEmails: this.formBuilder.array([])
}, {validators: PasswordValidator})
```

- Third step create a getter to easily access the form array in the HTML

```
get alternateEmail(){
  return this.registrationForm.get('alternateEmails') as
FormArray;
}
```

- Fourt step we will be creating a method that can be called to dynamically insert form controls into the form array

```
addAlternateEmail(){
  this.alternateEmail.push(this.formBuilder.control(''));
}
```


- Fifth step is to add a button in html to call above method

```
<label>Email</label>
    <button type="button" class="btn btn-secondary btn-sm"
(click)="addAlternateEmail()">Add e-mail</button>
    <input type="email" class="form-control"
formControlName="email" [class.is-
invalid]="registrationFormControl['email'].invalid">
    <div *ngIf="registrationFormControl['email'].invalid"
class="text-danger">
        Email is required</div>
```

- Final step is to iterate over the form array and display form fields.
- Since we are using array we need *ngFor directive

```
<div class="form-group">
    <label>Email<\/label>
    <button type="button" class="btn btn-secondary btn-sm"
(click)="addAlternateEmail()">Add e-mail<\/button>
    <input type="email" class="form-control"
formControlName="email" [class.is-
invalid]="registrationFormControl['email'].invalid">
    <div *ngIf="registrationFormControl['email'].invalid"
class="text-danger">
        Email is required<\/div>

    <div formArrayName="alternateEmail" *ngFor="let email of
alternateEmail.controls; let i = index">
        <input type="text" class="form-control my-1"
[formControlName]="i">
    <\/div>
<\/div>
```

11. Submitting Form Data

- First step is to bind to ngSubmit and assign it to a method and define that method

```
<form [formGroup]="registrationForm" (ngSubmit)="onSubmit()">
```

```
onSubmit(){  
  console.log(this.registrationForm.value);  
}
```

- To send this data to server we make use of service
 - Create registration server
 - In registration.service.ts
 - Import and inject httpClient

```
import { Injectable } from '@angular/core';  
import { HttpClient } from '@angular/common/http';  
@Injectable({  
  providedIn: 'root'  
})  
export class RegistrationService {  
  
  constructor(  
    private _http:HttpClient  
  ) { }  
}
```

- Also need to include HttpClient module in the 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 { ReactiveFormsModule } from '@angular/forms';  
import { HttpClientModule } from '@angular/common/http';  
  
@NgModule({  
  declarations: [  
    AppComponent  
  ],  
  imports: [  
    BrowserModule,  
    AppRoutingModule,  
    ReactiveFormsModule,  
    HttpClientModule  
  ],  
  providers: [],  
  bootstrap: [AppComponent]
```

```
})  
export class AppModule { }
```

- Create the express service same as implemented in template driven forms.
- While adding the endpoint keep the endpoint name as register instead of enroll
- Once the server is created add url in the registration.service.ts

```
url = 'http://localhost:3000/register';
```

- Create method register which will make post request.
 - This method will accept an argument userData.
 - Within the method body make post request

```
register(userData:any){  
    return this._http.post<any>(this._url, userData);  
}
```

- Next step is to call this post request and subscribe in form components ts file

```
constructor(  
    private FormBuilder: FormBuilder,  
    private registrationService: RegistrationService  
){}  
}
```

```
onSubmit(){  
    this.registrationService.register(this.registrationForm.value).s  
ubscribe(  
    response => console.log(response),  
    error => console.log(error)  
    );  
}
```