Reactive Forms Module

SD 555 – Web Application Development III

Maharishi International University

Department of Computer Science

Associate Professor Asaad Saad

Maharishi International University - Fairfield, Iowa

Forms in Angular

A lot of applications are very form-intensive, especially for enterprise development.

Forms can end up being really complex:

- Form inputs are meant to **modify data**, both on the page and the server
- Users cannot be trusted in what they enter, so you need to **validate** values
- The UI needs to clearly **state expectations** and errors
- **Dependent fields** can have complex logic
- We want to be able to **test** our forms, without relying on DOM selectors

Form states such as being: **`valid`**, **`invalid`**, **`pristine`**, **`dirty`**, **`untouched`**, **`touched`**, **`disabled`**, **`enabled`**, **`pending..etc`**

Choosing an approach

Angular offers two approaches to work with forms:

`Reactive forms` `(Data-driven forms)`

Provide direct, explicit access to the underlying form's object model. Compared to template-driven forms, they 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.

`Template-driven forms`

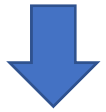
Rely on directives in the template to create and manipulate the underlying object model. They **don't scale** as well as reactive forms. If you have very **basic form requirements and logic** that can be managed solely in the template, template-driven forms could be a good fit.

Angular Form Models

`FormGroup` is an aggregation of the values of its children.

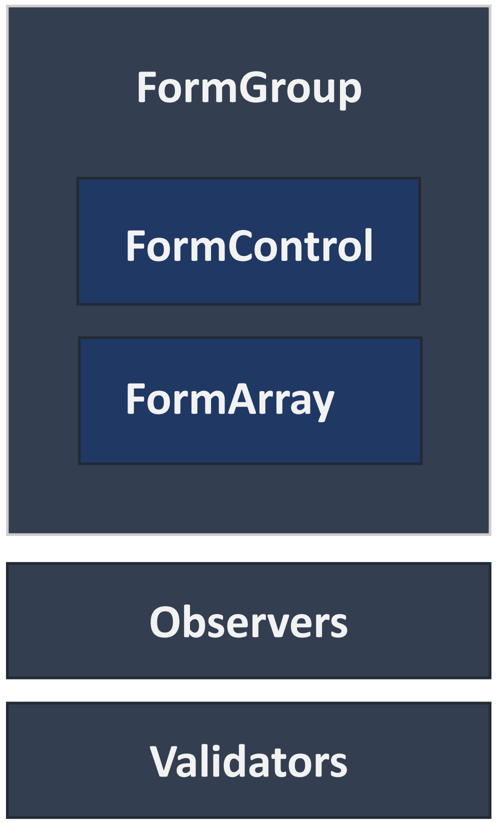
`FormControl` corresponds to a simple UI element, such as an input. It has a **`value`**, **`status`**, and a map of **`errors`**

`FormArray` A collection that has multiple `FormControl` elements.



All of the above models include two observables: **`statusChanges`** and **`valueChanges`**

The **`Validators`** interface allows us to add built-in validation rules.



Why Form Model?

The form model is a UI-independent way to represent user input consist of simple controls (`FormControl`) and their combinations (`FormGroup` and `FormArray`), where each control has a value, status, validators, errors, it emits events and it can be disabled.

Having this model has the following advantages:

- Form handling is a complex problem. Splitting it into UI-independent and UI-dependent parts makes them easier to manage. And we can test form handling without rendering UI.
- Having the form model makes reactive forms possible.

Form Parts

Form Model

Create the form models: `FormGroup`, `FormControl`, and `FormArray`.

DOM

Create your DOM form elements `<form/>, <input/>, <select/>, <textarea/>`.

Form Directives

Connect the above two parts with form directives: [`formGroup`], `formControlname.`

Type Unsafe Forms

Angular provides a type-safe checking out of the box, but only if we correctly create the form.

The following syntax is type unsafe and **NOT** recommended:

```
form!: FormGroup;
form!: FormGroup<any>;
form!: FormGroup<{
    email: FormControl<string | null>,
    password: FormControl<string | null>
}>;
```

On top of that, all form fields are considered to be **`nullable`**, this is because when we call the form **`reset()`** method, Angular will set all of the form fields to **`null`**.

FormBuilder Service

```
form = inject(FormBuilder).nonNullable.group({
    email: '',
    password: ''
})

// to pass validators:
form = formBuilerInstance.nonNullable.group({
   'form_field': ['default value’,
                  ValidatorFn | ValidatorFn[],
                  AsyncValidatorFn | AsyncValidatorFn[] ]
});

// or use the AbstractControlOptions as follows:
form = formBuilerInstance.nonNullable.group({
   'form_field': ['default value', {
                     validators?: ValidatorFn | ValidatorFn[];
                     asyncValidators?: AsyncValidatorFn | AsyncValidatorFn[];
                     updateOn?: "change" | "blur" | "submit"; }]
});
```

It helps building type-safe and reactive forms. The **`nonNullable`** does not affect the possibility of having undefined as a value, which should also be checked.

Form API

```
this.form.value // { email: '', password: '' }
this.form.valid // true of false
this.form.patchValue({ email: 'asaad@miu.edu', password: '123456' })

this.form.controls.email.value // reading the email value
this.form.controls.email.valid // true or false
this.form.controls.email.patchValue('theo@miu.edu') // setting the email value
this.form.controls.email.hasError('required') // reading validation errors
this.form.controls.email.setErrors({required: true}) // invalidate email
this.form.controls.email.setErrors(null) // set email as valid
```

Watching For Changes

Both **`FormGroup`** and **`FormControl`** have two built-in **`EventEmitter`** objects that we can use to observe changes.

```
this.form.statusChanges // observable
this.form.valueChanges // observable
this.form.events // observable, combine subscriptions to valueChanges and statusChanges

this.form.controls.email.statusChanges // observable
this.form.controls.email.valueChanges // observable
this.form.controls.email.events // observable
```

Remember to **unsubscribe** when you unmount the component.

Unified control state change events

The `events` observable emits four distinct types of events: `PristineEvent`, `StatusEvent`, `TouchedEvent`, and `ValueChangeEvent`.

```
control.events.subscribe(event => {
   if(e instanceof StatusEvent) {
        console.log(e.status)
   }
   if(e instanceof ValueChangeEvent) {
        console.log(e.value)
   }
   if(e instanceof PristineEvent) {
        console.log(e.pristine)
   }
   if(e instanceof TouchedEvent) {
        console.log(e.touched)
   }
});
```

Connecting DOM to Model

```
@Component({
  imports: [ReactiveFormsModule],
  template: `<form [formGroup]="form" (ngSubmit)="onSubmit()">
        <input type="text" formControlName="email" />
            @if(!form.controls.email.valid){ <div>Invalid email</div> }
        <input type="text" formControlName="password" />
        <button type="submit" [disabled]="!form.valid">Submit</button>
</form>`
})
```

Inspect the code and see how angular adds state change classes to the form elements.

Display Errors Gracefully

```
@Component({
   imports: [ReactiveFormsModule],
   template: `
   <form [formGroup]="form">
     <input formControlName="email" />
        @if(email.invalid && (email.dirty || email.touched)){
           <div>
              @if(email.hasError('required')){<div>Email is required</div>}
              @if(email.hasError('email')){<div>Email is not valid</div>}
           </div>
        }
   </form>
  `,
})
export class AppComponent {
  form = inject(FormBuilder).nonNullable.group({
    email: ['', [Validators.required, Validators.email]]
  })
  get email() { return this.form.controls.email; }
}
```

Custom Synchronous Validator

A validator is a function that takes a **`AbstractControl`** as an input and returns a **`StringMap<string, boolean>`** where the key is error code and the value is `true` if it fails

```
customValidator(control: AbstractControl): {[s: string]: boolean} | null {
   return control.value === 'Example'? { example: true } : null
}
```

```
@if(form_field.hasError('example')){ <div>Example is not valid</div> }
```

Custom Cross-Validation Sync Validator

```
public form: FormGroup = inject(FormBuilder).nonNullable.group({
    email: '',
    password: '',
    confirm_password: ''
  }, { validators: this.match_password })

match_password(control: AbstractControl) {
      return control.get('password')?.value === control.get('confirm_password')?.value
              ? null
              : { mismatch: true }
}
```

```
@if(form.hasError('mismatch')){ <div>Passwords do not match</div> }
```

Custom Asynchronous Validator

Asynchronous validator is a function that takes a **`FormControl`** as its input and returns a **`Promise<any>`** or an **`Observable<any>`**

```
asyncValidator(control: FormControl): Promise<any> | Observable<any> {
   // mimic HTTP request
   return of(null).pipe(delay(1500)) // valid, mimic delay after 1500ms
}
```

Notes

- Because async validators run asynchronously, make sure you bind **`this`** to the component instance.
- While the promise or the observable is being resolved, the status of the form/control will be **`PENDING`**
- You can delay updating the form validity by changing the **`updateOn`** property from **`change`** (default) to **`submit`** or **`blur`**. You may also replace the async validator by subscribing to **`valueChanges`** and add a **`debounceTime`** delay before running your custom validation logic.

Environment Variables

In the **`/src/environment`** folder you have an environment file for development and one for production.

Angular takes care of swapping the environment file for the correct one.

`ng generate environments`

```
"development": {
   "fileReplacements": [
       {
          "replace": "src/environments/environment.ts",
          "with": "src/environments/environment.development.ts"
        }
   ]
}
```

angular.json

+ OPTIONAL

Web Workers

Web workers lets you run CPU-intensive computations in a background thread, freeing the main thread to update the user interface.

`ng generate web-worker <location>`

```
addEventListener('message', ({ data }) => {
  const response = { student_id: data.id, grade: 95 }
  postMessage(response);
});
```

```
const worker = new Worker(new URL('./app.worker', import.meta.url));
worker.onmessage = ({ data }) => {
    results: { student_id: number, grade: number } = data;
};
worker.postMessage({ id: 98123 });
```

OPTIONAL

File Upload - Angular

```
@Component({
    selector: 'app-root',
    standalone: true,
    imports: [ReactiveFormsModule],
    template: `
      <form [formGroup]="form" (ngSubmit)="submit()">
        <input type="email" formControlName="email"/>
        <input formControlName="avatar" type="file" (change)="onFileSelect($event)" />
        <button [disabled]="form.invalid">Submit</button>
      </form> `
})
export class AppComponent {

    form = inject(FormBuilder).nonNullable.group({
        email: ['asaad@miu.edu', Validators.required],
        avatar: ['', Validators.required],
    })
    ... Next page
}
```

OPTIONAL

File Upload - Angular (Continued)

```
    file!: File;
    #http = inject(HttpClient);

    onFileSelect(event: Event) {
        const input = event.target as HTMLInputElement;
        if (input.files!.length > 0) this.file = input.files![0];
    }

    submit() {
        const formData = new FormData();
        formData.append('email', this.form.get('email')?.value as string);
        formData.append('avatar', this.file);

        this.#http.post<{ success: boolean }>('http://localhost:3000/', formData)
            .subscribe(response => {
                console.log(response);
                this.form.reset();
            })
    }
}
```

OPTIONAL

Deploy for Production

```
> ng build // When you run the ng build command, it creates a /dist folder.
```

- Removes unwanted white space by minifying files.
- Uglifies files by renaming functions and variable names.
- AoT (Ahead-of-Time) compilation

To serve any static resource from the /public folder, set the base for your assets:

```
> ng build --base-href /public/
```