



Création et validation de formulaires avec l'approche ReactiveForm



United Nations
Educational, Scientific and
Cultural Organization



UNESCO Chair
"Project-based learning"
ESPRIT School of engineering, Tunisia



EUR-ACE®

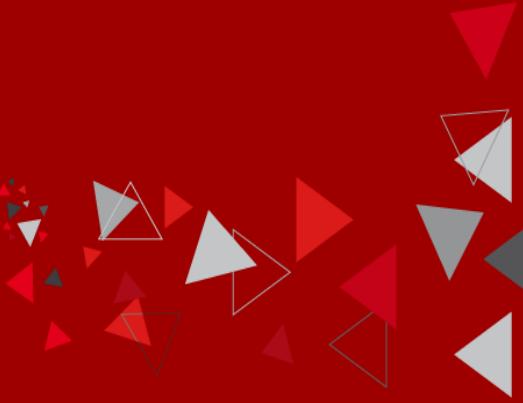
Délivrée par la
Commission
des Titres
d'Ingénieur



CONFÉRENCE DES
GRANDES
ÉCOLES



CONCEIVE DESIGN IMPLEMENT OPERATE



- ▶ **Introduction**
- ▶ **Création de formulaires**
- ▶ **Validation de formulaires**



Introduction



Introduction



- Les formulaires sont presque toujours présents sur tous les sites Web et applications.
- Ils peuvent être utilisés pour effectuer: l'authentification, la création d'un profil, l'envoie d'une demande, contact, etc.
- Un Formulaire, avant sa soumission, doit respecter un ensemble de règles afin de garantir l'intégralité des données soumises: Il doit être valide
- La validation d'un formulaire se fait moyennant des validateurs.
- Par exemple si un champ est obligatoire, alors il faut attacher le validateur « required » à ce champ.



Les approches de création de formulaire



Pour la création des formulaires, Angular propose deux approches:

1. **Template Driven Forms** : Le formulaire ainsi que les validateurs sont créés directement au niveau du template.
2. **Reactive Forms (ou Model Driven Forms)**: Le formulaire ainsi que les validateurs sont créées dans la classe du composant et ensuite liés au template grâce au DataBinding.

```
import { ReactiveFormsModule } from '@angular/forms';  Reactive Forms
import { FormsModule } from '@angular/forms';      Template Driven Forms
@NgModule({
  ...
  imports: [ // other imports ...,
    ReactiveFormsModule,
    FormsModule ],
  ....)
export class AppModule { }
```



Les approches de création de formulaire



- La différence majeure entre les deux approches est que les **Reactive forms** sont synchrones tandis que les **Template-driven forms** sont asynchrones



Création de formulaires



Reactive Form - Introduction



- En appliquant l'approche de Reactive Forms, le formulaire ainsi que ses éléments sont créés au niveau de la classe du composant et affichés par la suite au niveau du template
- La création du formulaire et ses différents éléments se fait grâce aux classes **FormControl**, **FormGroup** et **FormArray**.
- Avec cette approche, le composant est capable d'observer le changement d'état de chaque élément et d'y réagir=> les mises à jour de valeur et de validité sont toujours synchrones et sous le contrôle.
=> C'est pour qu'on parle du mode synchrone.



Reactive Form - FormControl



class **FormControl** extends AbstractControl

- La classe FormControl permet de générer un élément de contrôle (input, select, ...) au niveau de la classe du composant et lui affecter des validateurs si besoin.
- Elle permet également de suivre l'état de chaque élément de contrôle.

```
username = new FormControl('Toto',  
[Validators.required,Validators.minLength(3)]);
```

Toto: Valeur
par défaut

```
username = new FormControl({value:'Toto',disabled:true},  
[Validators.required,Validators.minLength(3)]);
```

Toto: Valeur par
défaut
Disabled: état par
défaut



Reactive Form - FormControl



- Création du formControl dans fichier html en dehors d'un formulaire

```
<input type="text" [formControl]="username">
```

- Création du formControl dans fichier html dans un formulaire

```
<form>  
<input type="text" formControlName="username">  
</form>
```



Reactive Form - FormGroup



class **FormGroup** extends AbstractControl

- Permet de créer une collection de **FormControl**.
- Cette classe suit la valeur et l'état de validité de chaque instance **FormControl** qu'elle contient.
- Si un FormControl du FormGroup est invalide alors le FormGroup en entier est invalide.

```
this.user_data = new FormGroup{  
    username: new FormControl('toto',Validators.required),  
    city: new FormControl('Ariana', Validators.required)  
};
```



Reactive Form - FormGroup



- On peut imbriquer plusieurs FormGroup.
- Le **FormGroup** racine correspond à la totalité du formulaire: <form>
- Le **FormGroup** imbriqué est un sous ensemble d'éléments dans le formulaire.

```
this.userData = new FormGroup({  
    username: new FormControl('toto',Validators.required),  
    city: new FormControl('Ariana') ,  
    autres : new FormGroup({  
        email : new FormControl (""),  
        fb: new FormControl ("")    }));  
})});
```

FormGroup racine ⇔ <form>

FormGroup imbriqué ⇔ un sous ensemble



Reactive Form - FormGroup



- L'instance créée de la classe **FormGroup racine** est liée à l'élément HTML correspondant grâce à la directive **FormGroupDirective: [formGroup]**

```
<form [formGroup]="user_data"></form>
```

- L'instance créée de classe **FormGroup imbriquée** est liée à un élément HTML avec la directive **FormGroupName : formGroupName**

```
<form [formGroup]="user_data">  
  <div formGroupName="autres">  
    .....  
  </div>  
</form>
```



Reactive Form - FormGroup



- Chaque FormControl du FormGroup (racine ou imbriqué) est lié à l'élément HTML correspondant grâce à la directive **FormControlName**.

```
<form [formGroup]="user_data">
  <input type="text" formControlName="username" />
  <input type="text" formControlName="city" />
  <div formGroupName="autres">
    Email: <input formControlName="email"> <br>
    FB: <input formControlName="fb">
  </div>
</form>
```



Reactive Form - FormGroup



La classe FormGroup contient un ensemble de méthode qui nous permette de manipuler les éléments (FormControls) d'un FormGroup comme:

- registerControl(): enregistrer un élément dans un FormGroup
- addControl(): ajouter un élément
- removeControl(): supprimer un élément
- setControl(): modifier un élément par un autre
- contains() : vérifier si un élément existe
- setValue(): modifier la valeur d'un élément
- reset(): vider les valeurs dans
- getRawValue() : Récupère toutes les valeurs quel que soit l'état (même si disabled)
- value: Récupère les valeurs des éléments actifs (non disabled)

Reactive Form – FormGroup

Récupération des valeurs



```
userForm : FormGroup = new FormGroup({...})
```

```
add(){
    this.users.push(this.userForm.getRawValue());
    this.userForm.reset();
}
```

Reset(): Le Formulaire est vidé

getRawValue(): crée un objet dont les attributs sont les noms des FormControl du FormGroup et les valeurs sont les informations saisies dans le formulaire.



Reactive Form - FormArray



```
class FormArray extends AbstractControl
```

- Une variante de **FormGroup**.
- Suit la valeur et l'état de validité d'un tableau d'éléments (FormControl, FormGroup ou FormArray)
- Utile dans le cas où le nombre de contrôles dans un formulaire est inconnu=> permet de créer des formulaires **dynamiques**.
- Un contrôle dans un FormArray peut être anonyme
- Il doit exister dans un FormGroup



Reactive Form – FormArray - Exemple



```
import { FormArray } from '@angular/forms';
```

```
userForm : FormGroup;
```

```
ngOnInit(){
  this.userForm=new FormGroup({
    ....,
    competences: new FormArray([ new FormControl("") ])
  })
}
```

```
get competences() {
  return this.loginForm.get('competences') as FormArray;
}
addCompetence() {
  this.competences.push(new FormControl(""));
}
```

```
<form [formGroup]="userForm" (submit)="addConnexion()">
  .....
  <div formArrayName="competences">
    <h2>Compétences</h2>
    <button (click)="addCompetence()" type="button">++++</button>
    <div *ngFor="let competence of competences.controls; let i=index">
      <label for="competence-{{ i }}>Competence:</label>
      <input id="competence-{{ i }}" type="text" [formControlName]= "i">
    </div>
  </div>
  <div class="div-btn">
    <button type="submit" class="btn btn-primary">s'inscrire</button>
  </div>
</form>
```



Reactive Form – Classes et directives



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

Classe	Directive	Rôle de la directive
FormControl	FormControlDirective	Synchronise une instance d'un Form control <u>autonome</u> avec un élément html
	FormControlName	Synchronise un FormControl <u>dans un FormGroup</u> avec un élément html
FormGroup	FormGroupDirective	Lie un FormGroup avec un élément du DOM
	FormGroupName	Synchronise un FormGroup imbriqué avec un élément du DOM
FormArray	FormArrayName	Synchronise un FormArray imbriqué avec un élément du DOM



Reactive Form - FormBuilder



- **FormBuilder**: est une classe (helper) qui crée pour nous des instances **FormGroup**, **FormControl** et **FormArray**. En gros, cela réduit la répétition et l'encombrement en gérant pour vous les détails de la création du contrôle de formulaire.

```
This.validations_form=this.formBuilder.group({  
    username: ['', Validators.required],  
    email: ['', Validators.compose([ Validators.required, Validators.pattern('^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-].[a-zA-Z0-9-.]+$') ]),  
    address: this.formBuilder.group({  
        street: '',  
        city: ''})});
```

FormGroup racine

FormGroup fils



setValue()



- Permet de définir une nouvelle valeur pour un contrôle individuel.
- Respecte strictement la structure du FormGroup ou FormArray et remplace la valeur entière du contrôle. **Utilisé pour mettre à jour toutes les valeurs dans le FormGroup ou le FormArray**

```
const form = this.fb.group({  
  name: ['', Validators.required],  
  age: ['', Validators.required]  
});  
  
form.setValue({ name: 'John', age: 30 });  
// correcte, tous les champs sont modifiés  
  
form.setValue({ name: 'John' });  
// erreur, le champ age manque.
```



patchValue()



- Utilisé pour mettre à jour partiellement les valeurs dans le FormGroup ou le FormArray.
- Il n'est pas nécessaire de transmettre la structure complète du formulaire.

```
const form = this.fb.group({  
  name: ['', Validators.required],  
  age: ['', Validators.required]  
});  
  
form.patchValue({ name: 'John' });  
// Met à jour uniquement le champ `name`  
form.patchValue({ name: 'Jane', age: 25 });  

```



Validation de formulaires



Reactive Form - Validation



- Les validateurs sont définis lors de la création d'un FormControl dans la classe du composant.
- Il faut importer **Validators** depuis **@angular/forms**

```
import { Validators } from '@angular/forms';
```

- Angular appelle ces validateurs pour chaque changement de la valeur du FormControl
- Il y a deux types de validateurs: les validateurs synchrones et les validateurs asynchrones



Reactive Form - Validation



1. Les validateurs **synchrones** : passés dans le deuxième argument lors de l'instanciation du FormControl. Ils retournent **immédiatement** une liste d'erreurs ou null
2. Les validateurs **asynchrones** : Il est parfois nécessaire d'implémenter des "validators" asynchrones (e.g. vérification distante via une API). Un "validator" asynchrone se comporte de la même façon qu'un "validator" synchrone mais au lieu de retourner null ou un objet d'erreur, il doit retourner un Observable.

Pour des raisons de performances, Angular n'exécute les validateurs asynchrones que si tous les validateurs synchrones réussissent.



Validateurs prédéfinis – Built-in Validators



- Validators.required
- Validators.minLength
- Validators.maxLength
- Validators.pattern
- Validators.email
- Validators.min
- Validators.max

Validateurs personnalisés: Exemple ForbiddenNameValidator



```
//forbidden-name.validator.ts
export function forbiddenNameValidator(nameRe: RegExp):
ValidatorFn {
  return (control: AbstractControl): ValidationErrors | null => {
    const forbidden = nameRe.test(control.value);
    return forbidden ? {forbiddenName: {value: control.value}} : null;
  };
}
```



Reactive Form - Validation



Les classes **FormControl** et **FormGroup** héritent de la même classe **AbstractControl**. Quelques propriétés sont dans le tableau ci dessous.

Propriété	Rôle
Valid/invalid	Si valid = true alors invalid=false => détermine si un élément est valid ou non en fonction des validateurs déclarés dessus
errors	Si null => pas d'erreur Si non null (objet) => indique les erreurs commise
Pristine /dirty	Si pristine = true alors dirty = false => l'élément sa valeur initial n'a pas changé et vis versa.
Touched/untouched	Si touched = true alors untouched = false => l'élément a été visité
Status = VALID ou INVALID	détermine si un élément est valid ou non en fonction des validateurs déclarés dessus



Reactive Form – Validation - FormControl



Nous pouvons accéder à n'importe quel FormControl à partir du FormGroup parent

- Dans le TS:

FormGroup
(propriété)

FormControl

```
get name() { return this.userForm.get('name'); }
```

```
get name() { return this.userForm.controls['name']; }
```

- OU BIEN Dans le HTML:

```
<div *ngIf="userForm.get('name').errors['required']">  
    Name is required. </div>
```



Reactive Form – Validation - FormControl



1. Définition des validateurs synchrones

```
this.userForm = new FormGroup({  
  'name': new FormControl(this.model.name, [ Validators.required,  
    Validators.minLength(4), forbiddenNameValidator(/bob/i) ])}  
  get name(){return this.userForm.get('name'); }
```

On peut créer des validateurs personnalisés tel que ici
forbiddenNameValidator()

2. Détection des erreurs et affichage des messages

```
<div *ngIf="name.hasError('required')"> Name is required. </div>  
<div *ngIf="name.hasError('minlength')"> 4 characters long. </div>  
<div *ngIf="userForm.get('name').hasError('forbiddenName')"> Name cannot  
be Bob. </div>
```

Reactive Form – Validation – FormControl dans FormGroup imbriqué



1. Définition des validateurs synchrones

```
ngOnInit(){
  this.userForm = new FormGroup({
    'name': new FormControl(this.model.name, [ Validators.required,
    Validators.minLength(4), forbiddenNameValidator(/bob/i)]})
    autres : new FormGroup({
      email : new FormControl ("",[Validators.required, Validators.minLength(4)]),
      fb: new FormControl ("", [Validators.required, Validators.pattern("^[a-zA-Z0-
      9]{8,}?"))]
    )})
  }
  get myEmail(){
    return this.userForm.controls['autres'].get('email');
```

Reactive Form – Validation – FormControl dans FormGroup imbriqué



2. Détection des erreurs et affichage des messages

```
<div *ngIf="myEmail.invalid">  
  <div *ngIf="myEmail.hasError('required')"> Name is required. </div>  
  <div *ngIf="myEmail.hasError('minlength')"> 4 characters long. </div>  
</div>
```

L'objet formControl relatif à l'email est récupéré depuis le TS sous le nom de myEmail

```
<div *ngIf="userForm.get('autres').get('fb').invalid">  
  <div *ngIf="userForm.get('autres').get('fb').hasError('required')"> Name is required. </div>  
  <div *ngIf="userForm.controls['autres'].get('fb').hasError('minlength')"> 4 characters long.  
</div>  
</div>
```

L'objet formControl **fb** est accédé à partir de l'HTML depuis la propriété userForm



Références



- [1] : <https://angular.io/api/forms/FormControl>
- [2] : <https://angular.io/api/forms/AbstractControlDirective>



► Merci de votre attention