Angular : formulaire

Achref El Mouelhi

Docteur de l'université d'Aix-Marseille
Chercheur en programmation par contrainte (IA)
Ingénieur en génie logiciel

elmouelhi.achref@gmail.com

Plan

Plan

Angular

Formulaire

- Outil graphique que nous créons avec le langage de description **HTML**
- Permettant à l'utilisateur la saisie de données
- Solution pour soumettre les données vers une deuxième page, vers une base de données...

Angular

Apport d'**Angular** ?

- Récupération des données saisies
- Validation et contrôle des valeurs saisies
- Gestion d'erreurs
- ...

Angular

Que propose **Angular** ?

- Template-driven forms : utilisant `FormsModule`, facile et conseillé pour les formulaires simples
- Reactive forms basé sur `ReactiveFormsModule` : robuste, évolutif et conçu pour les applications nécessitant des contrôles particuliers
  - Form Group
  - Form Builder

Angular

Pour commencer

- créer un composant `formulaire` dans le module `cours`
- créer un chemin `/formulaire` permettant d'afficher ce composant

Angular

Le fichier `formulaire.ts`

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-formulaire',
  imports: [],
  templateUrl: './formulaire.html',
  styleUrl: './formulaire.css'
})
export class Formulaire {

}
```

Le fichier `formulaire.html`

```
<p>
  formulaire works!
</p>
```

Angular

Le fichier `formulaire.ts`

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-formulaire',
  imports: [],
  templateUrl: './formulaire.html',
  styleUrl: './formulaire.css'
})
export class Formulaire {
  direBonjour(): void {
    console.log('Bonjour');
  }
}
```

Le fichier `formulaire.html`

```
<div>
  <button (click)="
    direBonjour()">
    cliquer
  </button>
</div>
```

Angular

Explication

- En cliquant sur le bouton `cliquer`, la méthode `direBonjour()` est exécutée.
- `Bonjour` est affiché dans la console

Angular

Questions

Comment faire si on voulait

- initialiser la zone de saisie avec la valeur d'une attribut ?
- récupérer la valeur saisie dans une zone texte et l'afficher dans une autre partie du composant ?

Angular

Questions

Comment faire si on voulait

- initialiser la zone de saisie avec la valeur d'une attribut ?
- récupérer la valeur saisie dans une zone texte et l'afficher dans une autre partie du composant ?

Solutions

- Combiner Property binding `[ ... ]` et Event binding `( ... )`, ou
- Utiliser directement Two-way binding `[( ... )]`.

Angular

Le fichier `formulaire.ts`

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-formulaire',
  imports: [],
  templateUrl: './formulaire.html',
  styleUrl: './formulaire.css'
})
export class Formulaire {
  nom = "wick";


  direBonjour(value: string){
    this.nom = value;
  }
}
```

Le fichier `formulaire.html`

```
<div>
  <input type=text [value]='nom' #entry>
</div>
<div>
  <button (click)="direBonjour(entry.value)">
    cliquer
  </button>
</div>
<div>
    Bonjour {{ nom }}
</div>
```

Angular

Le fichier `formulaire.ts`

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-formulaire',
  imports: [],
  templateUrl: './formulaire.html',
  styleUrl: './formulaire.css'
})
export class Formulaire {
  nom = "wick";


  direBonjour(value: string){
    this.nom = value;
  }
}
```

Le fichier `formulaire.html`

```
<div>
  <input type=text [value]='nom' #entry>
</div>
<div>
  <button (click)="direBonjour(entry.value)">
    cliquer
  </button>
</div>
<div>
    Bonjour {{ nom }}
</div>
```

`#entry` est une variable locale contenant les attributs de cet élément HTML. Les variables locales peuvent pointer sur n'importe quel élément **HTML**, `<p>`, `<h>`...

Angular

Remarque

Il est possible de simplifier le code précédant en utilisant le **Two way binding**.

Angular

Plusieurs formes de binding

- `{{ interpolation }}` : permet de récupérer la valeur d'un attribut déclarée dans le `.ts`
- `[ one way binding ]` : permet aussi de récupérer la valeur d'un attribut déclarée dans le `.ts`
- `( event binding )` : permet au `.ts` de récupérer des valeurs passées par le `.html`

Angular

Plusieurs formes de binding

- `{{ interpolation }}` : permet de récupérer la valeur d'un attribut déclarée dans le `.ts`
- `[ one way binding ]` : permet aussi de récupérer la valeur d'un attribut déclarée dans le `.ts`
- `( event binding )` : permet au `.ts` de récupérer des valeurs passées par le `.html`

`{{ interpolation }}` **est un raccourci de** `[ one way binding ]`

```
<p [textContent]= "result"></p>
<p> {{ result }} </p>
<!-- Les deux écritures sont équivalentes -->
```

Angular

Il est possible de combiner `one way binding` et `event binding`

- Résultat : `two way binding`
- Un changement de valeur dans `component.ts` sera aperçu dans `component.html` et un changement dans `component.html` sera reçu dans `component.ts`

Angular

Il est possible de combiner `one way binding` et `event binding`

- Résultat : `two way binding`
- Un changement de valeur dans `component.ts` sera aperçu dans `component.html` et un changement dans `component.html` sera reçu dans `component.ts`

`two way binding`

- Pour la liaison bidirectionnelle, il nous faut la propriété `ngModel`
- Pour pouvoir utiliser la propriété `ngModel`, il faut ajouter le module `FormsModule` dans `imports`

Angular

Le fichier `formulaire.ts`

```
import { Component } from '@angular/core';
import { FormsModule } from '@angular/forms';

@Component({
  selector: 'app-formulaire',
  imports: [FormsModule],
  templateUrl: './formulaire.html',
  styleUrl: './formulaire.css'
})
export class Formulaire {

  nom = "Wick";
  result = "";

  direBonjour() {
    this.result = this.nom;
  }
}
```

Le fichier `formulaire.html`

```
<div>
  <input type=text [(ngModel)]=nom>
</div>
<div>
  <button (click)="direBonjour()">
    cliquer
  </button>
</div>
<div>
  Bonjour  {{ result }}
</div>
```

Angular

Le fichier `formulaire.ts`

```
import { Component } from '@angular/core';
import { FormsModule } from '@angular/forms';

@Component({
  selector: 'app-formulaire',
  imports: [FormsModule],
  templateUrl: './formulaire.html',
  styleUrl: './formulaire.css'
})
export class Formulaire {

  nom = "Wick";
  result = "";

  direBonjour() {
    this.result = this.nom;
  }
}
```

Le fichier `formulaire.html`

```
<div>
  <input type=text [(ngModel)]=nom>
</div>
<div>
  <button (click)="direBonjour()">
    cliquer
  </button>
</div>
<div>
  Bonjour  {{ result }}
</div>
```

Nous n'avons pas besoin de cliquer pour envoyer la valeur saisie dans le champ texte, elle est mise à jour simultanément dans la classe quand elle est modifiée dans la vue.

Angular

Le fichier `formulaire.ts`

```
import { Component } from '@angular/core';
import { FormsModule } from '@angular/forms';

@Component({
  selector: 'app-formulaire',
  imports: [FormsModule],
  templateUrl: './formulaire.html',
  styleUrl: './formulaire.css'
})
export class Formulaire {

  nom = "";

}
```

Le fichier `formulaire.html`

```
<div>
  <input type=text [(ngModel)]=nom>
</div>
<div>
  Bonjour  {{ nom }}
</div>
```

Angular

Commençons par créer une interface `personne`

```
ng generate interface interfaces/personne
```

Angular

Commençons par créer une interface `personne`

```
ng generate interface interfaces/personne
```

On peut aussi utiliser le raccourci

```
ng g i interfaces/personne
```

Angular

Commençons par créer une interface `personne`

```
ng generate interface interfaces/personne
```

On peut aussi utiliser le raccourci

```
ng g i interfaces/personne
```


Mettons à jour le contenu de `personne.ts`

```
export interface Personne {
    id?: number | null | undefined;
    nom?: string | null | undefined;
    prenom?: string | null | undefined;
}
```

Angular

Considérant le formulaire `formulaire.html`

```
<form>
  <div>
    Nom : <input type=text name=nom [(ngModel)]=personne.nom>
  </div>
  <div>
    Prénom : <input type=text name=prenom [(ngModel)]=personne.prenom>
  </div>
  <div>
    <button>
      Ajouter
    </button>
  </div>
</form>
```

Angular

Considérant le formulaire `formulaire.html`

```
<form>
  <div>
    Nom : <input type=text name=nom [(ngModel)]=personne.nom>
  </div>
  <div>
    Prénom : <input type=text name=prenom [(ngModel)]=personne.prenom>
  </div>
  <div>
    <button>
      Ajouter
    </button>
  </div>
</form>
```

Pour utiliser `ngModel` dans un formulaire, il faut définir une valeur pour l'attribut `name` de chaque élément du formulaire.

Angular

Explication

- **Angular** crée un objet de formulaire interne pour suivre l'état, la validité et les valeurs des champs.
- L'attribut `name` est requis pour associer chaque champ à une clé unique dans cet objet de formulaire.
- En dehors des formulaires, l'attribut `name` n'est pas nécessaire car **Angular** n'a pas besoin de cet enregistrement formel pour gérer un input indépendant.

Angular

Le fichier `formulaire.ts`

```
import { Component } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { Personne } from '../../interfaces/personne';

@Component({
  selector: 'app-formulaire',
  imports: [FormsModule],
  templateUrl: './formulaire.html',
  styleUrl: './formulaire.css'
})
export class Formulaire {
    personne: Personne = { };
}
```

Angular

Pour soumettre le formulaire, il faut qu'il soit valide. Supposant que

- les deux zones textes sont obligatoires.
- le bouton doit être initialement désactivé, on l'active que lorsque le formulaire est valide.

Angular

Commençons par rendre les deux zones textes obligatoires

```
<form>
  <div>
    Nom : <input type=text name=nom [(ngModel)]=personne.nom required>
  </div>
  <div>
    Prénom : <input type=text name=prenom [(ngModel)]=personne.prenom
       required>
  </div>
  <div>
    <button>
      Ajouter
    </button>
  </div>
</form>
```

Angular

Désactivons le bouton

```
<form>
  <div>
    Nom : <input type=text name=nom [(ngModel)]=personne.nom required>
  </div>
  <div>
    Prénom : <input type=text name=prenom [(ngModel)]=personne.prenom
       required>
  </div>
  <div>
    <button disabled>
      Ajouter
    </button>
  </div>
</form>
```

Angular

Question : comment réactiver le bouton ?

- Écrire une fonction **JavaScript** pour tester si les deux champs ne sont pas vides.
- Écrire une méthode dans la classe qui vérifiera à chaque saisie si les deux champs ne sont pas vides pour réactiver le bouton.

Angular

Question : comment réactiver le bouton ?

- Écrire une fonction **JavaScript** pour tester si les deux champs ne sont pas vides.
- Écrire une méthode dans la classe qui vérifiera à chaque saisie si les deux champs ne sont pas vides pour réactiver le bouton.

Avec les directives**Angular**, il y a plus simple

- Utiliser la directive `ngForm` pour créer une variable locale associée au formulaire.
- Exploiter la propriété `valid` de `ngForm` pour valider le formulaire.

Angular

Pour le réactiver

```
<form #monForm=ngForm>
  <div>
    Nom : <input type=text name=nom [(ngModel)]=personne.nom required>
  </div>
  <div>
    Prénom : <input type=text name=prenom [(ngModel)]=personne.prenom
       required>
  </div>
  <div>
    <button [disabled]=!monForm.valid>
      Ajouter
    </button>
  </div>
</form>
```

Angular

Autres propriétés de `ngForm`

- `invalid` : l'inverse de `valid`, donc `true` si le formulaire est invalide.
- `dirty` : indique si le formulaire a été modifié par l'utilisateur (même si la modification a été réinitialisée par la suite). `dirty` sera `true` dès que l'utilisateur interagit avec un champ.
- `pristine` : l'inverse de `dirty`, contient `true` tant que le formulaire n'a pas été modifié.
- `touched` : indique si au moins un champ du formulaire a été "touché" (cliqué ou focalisé, puis quitté) par l'utilisateur.
- `untouched` : l'inverse de `touched`, `true` si aucun champ n'a été "touché".
- `status` : représente l'état général du formulaire sous forme de chaîne de caractères, avec des valeurs possibles comme `VALID`, `INVALID`, ou `DISABLED`.

Angular

Question : et si on voulait afficher en rouge les champs obligatoires non-renseignés ?

- Utiliser les variables locales.
- Exploiter les propriétés **CSS** fournies par **Angular**.

Angular

Utilisons les variables locales et affichons les classes CSS associées attribuées par Angular

```
<form #monForm=ngForm>
  <div>
    Nom : <input type=text name=nom [(ngModel)]=personne.nom required #
       nom>
  </div>
  {{ nom.className}}
  <div>
    Prénom : <input type=text name=prenom [(ngModel)]=personne.prenom
       required #prenom>
  </div>
   {{ prenom.className}}
  <div>
    <button [disabled]=!monForm.valid>
      ajouter
    </button>
  </div>
</form>
```

Angular

Les classes **CSS** affichées pour les deux champs

- `ng-untouched` : classe **Angular** appliquée quand le champ n'est pas touché (son inverse est `ng-touched`)
- `ng-pristine` : classe **Angular** appliquée quand le champ est vide (son inverse est `ng-dirty`)
- `ng-invalid` : classe **Angular** appliquée quand le champ n'est pas `valid` (son inverse est `ng-valid`)

Angular

Nettoyons le code précédent

```
<form #monForm=ngForm>
  <div>
    Nom : <input type=text name=nom [(ngModel)]=personne.nom
       required #nom>
  </div>
  <div>
    Prénom : <input type=text name=prenom [(ngModel)]=personne.
       prenom required #prenom>
  </div>
  <div>
    <button [disabled]=!monForm.valid>
      ajouter
    </button>
  </div>
</form>
```

Angular

Définissons des propriétés pour quelques classes CSS fournies par Angular

```
.ng-invalid:not(form){
    border-left: 5px  solid red;
}

.ng-valid:not(form){
    border-left: 5px  solid green;
}
```

Angular

On peut aussi afficher un message en cas de violation de contrainte

```
<form #monForm=ngForm>
  <div>
    Nom :  <input type=text name=nom [(ngModel)]=personne.nom required
       #nom="ngModel">
  </div>
  <div [hidden]="nom.valid">
    Le nom est obligatoire
  </div>
  <div>
    Prénom :  <input type=text name=prenom [(ngModel)]=personne.prenom
       required #prenom="ngModel">
  </div>
    <div [hidden]="prenom.valid">
      Le prénom est obligatoire
    </div>
  <div>
    <button [disabled]=!monForm.valid>
      ajouter
    </button>
  </div>
</form>
```

Angular

Pour ne pas afficher les messages d'erreur au chargement de la page

```
<form #monForm=ngForm>
  <div>
    Nom : <input type=text name=nom [(ngModel)]=personne.nom required #
       nom="ngModel">
  </div>
  <div [hidden]="nom.valid || nom.pristine">
    Le nom est obligatoire
  </div>
  <div>
    Prénom : <input type=text name=prenom [(ngModel)]=personne.prenom
       required #prenom="ngModel">
  </div>
    <div [hidden]="prenom.valid || prenom.pristine">
      Le prénom est obligatoire
    </div>
  <div>
    <button [disabled]=!monForm.valid>
      ajouter
    </button>
  </div>
</form>
```

Angular

Pour soumettre un formulaire, on utilise la directive `ngSubmit`

```
<form #monForm=ngForm (ngSubmit)=ajouterPersonne()>
  <div>
    Nom : <input type=text name=nom [(ngModel)]=personne.nom required #
       nom="ngModel">
  </div>
  <div [hidden]="nom.valid || nom.pristine">
    Le nom est obligatoire
  </div>
  <div>
    Prénom : <input type=text name=prenom [(ngModel)]=personne.prenom
       required #prenom="ngModel">
  </div>
    <div [hidden]="prenom.valid || prenom.pristine">
      Le prénom est obligatoire
    </div>
  <div>
    <button [disabled]=!monForm.valid>
      ajouter
    </button>
  </div>
</form>
```

Angular

Le fichier `formulaire.ts`

```
import { Component } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { Personne } from '../../interfaces/personne';

@Component({
  selector: 'app-formulaire',
  imports: [FormsModule],
  templateUrl: './formulaire.html',
  styleUrl: './formulaire.css'
})
export class Formulaire {

  personnes: Array<Personne> = [];
  personne: Personne = {};

  ajouterPersonne(): void {
    this.personnes.push({ ...this.personne });
    console.log(this.personnes);
  }
}
```

Angular

Pour vider les champs du formulaire après ajout

```
import { Component } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { Personne } from '../../interfaces/personne';

@Component({
  selector: 'app-formulaire',
  imports: [FormsModule],
  templateUrl: './formulaire.html',
  styleUrl: './formulaire.css'
})
export class Formulaire {

  personnes: Array<Personne> = [];
  personne: Personne = {};

  ajouterPersonne(): void {
    this.personnes.push(this.personne);
    this.personne = {};
    console.log(this.personnes);
  }
}
```

Angular

Remarques

- En ajoutant une nouvelle personnes, les champs sont vidés.
- Mais les messages d'erreurs réapparaissent

Angular

Remarques

- En ajoutant une nouvelle personnes, les champs sont vidés.
- Mais les messages d'erreurs réapparaissent

Solution

Utiliser la méthode `reset()` de `ngForm`

Angular

Envoyons le formulaire à la soumission du formulaire

```
<form #monForm=ngForm (ngSubmit)=ajouterPersonne(monForm)>
  <div>
    Nom : <input type=text name=nom [(ngModel)]=personne.nom required #
       nom="ngModel">
  </div>
  <div [hidden]="nom.valid || nom.pristine">
    Le nom est obligatoire
  </div>
  <div>
    Prénom : <input type=text name=prenom [(ngModel)]=personne.prenom
       required #prenom="ngModel">
  </div>
    <div [hidden]="prenom.valid || prenom.pristine">
      Le prénom est obligatoire
    </div>
  <div>
    <button [disabled]=!monForm.valid>
      ajouter
    </button>
  </div>
</form>
```

Angular

Utilisons la méthode `reset()` pour vider les champs du formulaire

```
import { Component } from '@angular/core';
import { FormsModule, NgForm } from '@angular/forms';
import { Personne } from '../../interfaces/personne';

@Component({
  selector: 'app-formulaire',
  imports: [FormsModule],
  templateUrl: './formulaire.html',
  styleUrl: './formulaire.css'
})
export class Formulaire {

  personnes: Array<Personne> = [];
  personne: Personne = {};

  ajouterPersonne(form: NgForm): void {
    this.personnes.push(this.personne);
    form.reset();
  }
}
```

Angular

L'objet `form: NgForm` permet aussi de récupérer les valeurs du formulaire

```
import { Component, OnInit } from '@angular/core';
import { Personne } from '../../interfaces/personne';
import { NgForm } from '@angular/forms';

@Component({
  selector: 'app-formulaire',
  templateUrl: './formulaire.html',
  styleUrls: ['./formulaire.css']
})
export class Formulaire implements OnInit {

  personnes: Array<Personne> = [];
  personne: Personne = { };

  ajouterPersonne(form: NgForm): void {
    console.log(form.value);
    this.personnes.push(this.personne);
    form.reset();
  }
}
```

Angular

Exercice 1

- Modifier le composant `formulaire` pour afficher les personnes ajoutées en-dessous du formulaire
- À chaque ajout, la nouvelle personne ajoutée apparaît comme dernier élément de la liste `personnes` (affichée en-dessous du formulaire)

Angular

Exercice 2

- Créez un composant `calculette` (n'oubliez pas de lui associer une route `calculette`)
- Dans `calculette.html`, définissez deux champs `input` de type `number` et quatre boutons : un pour chaque opération arithmétique
- Le résultat sera affiché en-dessous du formulaire.

Angular

Pour commencer

- créer un composant `form`
- créer un chemin `/form` permettant d'afficher ce composant
- ajouter `ReactiveFormsModule` dans la section `imports`

Angular

FormControl

- Une classe **Angular**
- Permettant d'associer un attribut de composant à un champ de formulaire défini dans le template associé afin de faciliter
  - le binding
  - le contrôle et la validation

Angular

Dans `form.ts`**, déclarons un attribut** `nom` **de type** `FormControl`

```
import { Component } from '@angular/core';
import { FormControl, ReactiveFormsModule } from '@angular/forms';

@Component({
  selector: 'app-form',
  imports: [ReactiveFormsModule],
  templateUrl: './form.html',
  styleUrl: './form.css'
})
export class Form {

  nom = new FormControl('');
}
```

Angular

Dans `form.ts`**, déclarons un attribut** `nom` **de type** `FormControl`

```
import { Component } from '@angular/core';
import { FormControl, ReactiveFormsModule } from '@angular/forms';

@Component({
  selector: 'app-form',
  imports: [ReactiveFormsModule],
  templateUrl: './form.html',
  styleUrl: './form.css'
})
export class Form {

  nom = new FormControl('');
}
```

Dans `form.html`, on aura un champ de formulaire associé à l'objet `nom`.

Angular

Depuis Angular 14, on a la possibilité d'utiliser les typed forms (typage générique)

```
import { Component } from '@angular/core';
import { FormControl, ReactiveFormsModule } from '@angular/forms';

@Component({
  selector: 'app-form',
  imports: [ReactiveFormsModule],
  templateUrl: './form.html',
  styleUrl: './form.css'
})
export class Form {

  nom = new FormControl<string>('');
}
```

Angular

Dans `form.html`, on ajoute un champ associé à l'attribut `nom`

```
<div>
  <label for="nom">Nom</label>
  <input type="text" id="nom" [formControl]="nom">
</div>
```

Angular

Dans `form.html`, on ajoute un champ associé à l'attribut `nom`

```
<div>
  <label for="nom">Nom</label>
  <input type="text" id="nom" [formControl]="nom">
</div>
```

On peut ajouter un bouton avec un `event binding`

```
<label>
  <label for="nom">Nom</label>
  <input type="text" id="nom" [formControl]="nom">
</label>
<button (click)='afficherNom()'>cliquer</button>
```

Angular

Dans `form.html`, on ajoute un champ associé à l'attribut `nom`

```
<div>
  <label for="nom">Nom</label>
  <input type="text" id="nom" [formControl]="nom">
</div>
```

On peut ajouter un bouton avec un `event binding`

```
<label>
  <label for="nom">Nom</label>
  <input type="text" id="nom" [formControl]="nom">
</label>
<button (click)='afficherNom()'>cliquer</button>
```

N'oublions pas de définir la méthode `afficherNom()` dans `form.ts`.

Dans `form.ts`, on ajoute la méthode `afficherNom()`

```
import { Component } from '@angular/core';
import { FormControl, ReactiveFormsModule } from '@angular/forms';

@Component({
  selector: 'app-form',
  imports: [ReactiveFormsModule],
  templateUrl: './form.html',
  styleUrl: './form.css'
})
export class Form {

  nom = new FormControl('');

  afficherNom(): void {
    console.log(this.nom.value);
  }
}
```

Dans `form.ts`, on ajoute la méthode `afficherNom()`

```
import { Component } from '@angular/core';
import { FormControl, ReactiveFormsModule } from '@angular/forms';

@Component({
  selector: 'app-form',
  imports: [ReactiveFormsModule],
  templateUrl: './form.html',
  styleUrl: './form.css'
})
export class Form {

  nom = new FormControl('');

  afficherNom(): void {
    console.log(this.nom.value);
  }
}
```

En cliquant sur le bouton, la valeur saisie dans le champ texte sera affichée dans la console.

Angular

Dans `form.ts`, on peut utiliser le constructeur de `FormControl` pour définir une valeur initiale à afficher au chargement du composant

```
import { Component } from '@angular/core';
import { FormControl, ReactiveFormsModule } from '@angular/forms';

@Component({
  selector: 'app-form',
  imports: [ReactiveFormsModule],
  templateUrl: './form.html',
  styleUrl: './form.css'
})
export class Form {

  nom = new FormControl('wick')

  afficherNom(): void {
    console.log(this.nom.value);
  }
}
```

Angular

FormGroup

- Une classe **Angular**
- Composée de `FormControl` ou de `FormGroup` imbriqués

Angular

Dans `form.ts`, commençons par définir un objet de type `FormGroup`

```
import { Component } from '@angular/core';
import { FormControl, FormGroup, ReactiveFormsModule } from '@angular/
   forms';

@Component({
  selector: 'app-form',
  imports: [ReactiveFormsModule],
  templateUrl: './form.html',
  styleUrl: './form.css'
})
export class Form {

  personneForm = new FormGroup({
    id: new FormControl(''),
    nom: new FormControl(''),
    prenom: new FormControl('')
  });
}
```

Angular

Dans `form.html`, créons maintenant le formulaire prenant les trois éléments déclarés dans le `FormGroup`

```
<form [formGroup]="personneForm">
    <div>
        <label for="id">Identifiant</label>
        <input type="text" id="id" formControlName=id>
    </div>
    <div>
        <label for="nom">Nom</label>
        <input type="text" id="nom" formControlName=nom>
    </div>
    <div>
        <label for="prenom">Prénom</label>
        <input type="text" id="prenom" formControlName=prenom>
    </div>
</form>
```

Angular

On peut ajouter un bouton avec un `event binding`

```
<form [formGroup]="personneForm">
    <div>
        <label for="id">Identifiant</label>
        <input type="text" id="id" formControlName=id>
    </div>
    <div>
        <label for="nom">Nom</label>
        <input type="text" id="nom" formControlName=nom>
    </div>
    <div>
        <label for="prenom">Prénom</label>
        <input type="text" id="prenom" formControlName=prenom>
    </div>
    <button (click)='afficherTout()'>cliquer</button>
</form>
```

Angular

On peut ajouter un bouton avec un `event binding`

```
<form [formGroup]="personneForm">
    <div>
        <label for="id">Identifiant</label>
        <input type="text" id="id" formControlName=id>
    </div>
    <div>
        <label for="nom">Nom</label>
        <input type="text" id="nom" formControlName=nom>
    </div>
    <div>
        <label for="prenom">Prénom</label>
        <input type="text" id="prenom" formControlName=prenom>
    </div>
    <button (click)='afficherTout()'>cliquer</button>
</form>
```

N'oublions pas de définir la méthode `afficherTout()` dans `form.ts`.

Dans `form.ts`, ajoutons la méthode `afficherTout()`

```
import { Component } from '@angular/core';
import { FormControl, FormGroup, ReactiveFormsModule } from '@angular/
   forms';

@Component({
  selector: 'app-form',
  imports: [ReactiveFormsModule],
  templateUrl: './form.html',
  styleUrl: './form.css'
})
export class Form {

  personneForm = new FormGroup({
    id: new FormControl(''),
    nom: new FormControl(''),
    prenom: new FormControl('')
  });

  afficherTout(): void {
    console.log(this.personneForm.value);
  }
}
```

Angular

Pour récupérer le `FormControl` associé à `nom`

```
console.log(this.personneForm.controls.nom);
```

Angular

Pour récupérer le `FormControl` **associé à** `nom`

```
console.log(this.personneForm.controls.nom);
```

Ou aussi

```
console.log(this.personneForm.get('nom'));
```

Angular

Différences entre `get()` et `controls` dans les Reactive Forms

Aspect	**`controls.nom`**	**`get('nom')`**
Accès direct à un contrôle	Oui	Oui
Accès à un contrôle imbriqué (`adresse.rue`)	Non	Oui avec `get('adresse.rue')`
Typage strict (Angular $\geq$ 14)	Oui	Non
Accès dynamique (via variable)	Non	Oui
Lisibilité dans les formulaires simples	Bonne	Moyenne

Angular

Pour vider les champs d'un formulaire

```
this.personneForm.reset();
```

Angular

Pour vider les champs d'un formulaire

```
this.personneForm.reset();
```

Pour modifier la valeur d'un `FormControl`

```
this.personneForm.controls.nom.setValue('Doe');
```

Angular

On peut initialiser les champs du formulaire avec la méthode `setValue()`

```
import { Component, OnInit } from '@angular/core';
import { FormControl, FormGroup } from '@angular/forms';

@Component({
  selector: 'app-form',
  imports: [ReactiveFormsModule],
  templateUrl: './form.html',
  styleUrl: './form.css'
})
export class Form implements OnInit {

  personneForm = new FormGroup({
    id: new FormControl(),
    nom: new FormControl(''),
    prenom: new FormControl('')
  });

  ngOnInit() {
    this.personneForm.setValue({ nom: 'doe', prenom: 'john', id: 1 });
  }

  afficherTout() {
    console.log(this.personneForm.value);
    this.personneForm.reset();
  }
}
```

Angular

`valueChanges` **permet de surveiller le changement de valeur d'un champ du formulaire**

```
import { Component, OnInit } from '@angular/core';
import { FormControl, FormGroup } from '@angular/forms';

@Component({
  selector: 'app-form',
  imports: [ReactiveFormsModule],
  templateUrl: './form.html',
  styleUrl: './form.css'
})
export class Form implements OnInit {

  personneForm = new FormGroup({
    id: new FormControl(),
    nom: new FormControl(''),
    prenom: new FormControl('')
  });

  ngOnInit() {
    this.personneForm.controls.nom.valueChanges.subscribe(change => {
      console.log(change);
    });
  }

  afficherTout() {
    console.log(this.personneForm.value);
    this.personneForm.reset();
  }
}
```

Angular

Pour la soumission de formulaire, on ajoute un event binding `(ngSubmit)` et on ajoute un bouton de type `submit`

```
<form [formGroup]="personneForm" (ngSubmit)='afficherTout()'>
    <div>
        <label for="id">Identifiant</label>
        <input type="text" id="id" formControlName=id>
    </div>
    <div>
        <label for="nom">Nom</label>
        <input type="text" id="nom" formControlName=nom>
    </div>
    <div>
        <label for="prenom">Prénom</label>
        <input type="text" id="prenom" formControlName=prenom>
    </div>
    <button>Envoyer</button>
</form>
```

Angular

Pour la validation de formulaire, on commence par désactiver le bouton tant que le formulaire n'est pas valide

```
<form [formGroup]="personneForm" (ngSubmit)='afficherTout()'>
    <div>
        <label for="id">Identifiant</label>
        <input type="text" id="id" formControlName=id>
    </div>
    <div>
        <label for="nom">Nom</label>
        <input type="text" id="nom" formControlName=nom>
    </div>
    <div>
        <label for="prenom">Prénom</label>
        <input type="text" id="prenom" formControlName=prenom>
    </div>
    <button [disabled]='!personneForm.valid'>
        Envoyer
    </button>
</form>
```

Angular

Étape suivante : définir les règles de validation

- La classe `FormControl` peut prendre deux paramètres : la valeur initiale à afficher dans le formulaire et la deuxième une règle de validation.
- Pour définir une règle de validation, on peut utiliser la classe **Angular** `Validators` contenant plusieurs règles de validation.

Dans `form.ts`, définissons quelques règles de validation

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

personneForm = new FormGroup({
  id: new FormControl('', Validators.required),
  nom: new FormControl('', [Validators.pattern(/^[A-Z][a-z]{2,10}$/),
    Validators.required]),
  prenom : new FormControl('', [ Validators.required,
    checkPrenomValidator ])
});
```

Dans `form.ts`, définissons quelques règles de validation

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

personneForm = new FormGroup({
  id: new FormControl('', Validators.required),
  nom: new FormControl('', [Validators.pattern(/^[A-Z][a-z]{2,10}$/),
    Validators.required]),
  prenom : new FormControl('', [ Validators.required,
    checkPrenomValidator ])
});
```


Explication

- Le champ `id` est obligatoire.
- Le champ `nom` est obligatoire et doit respecter une expression régulière : le premier caractère est une lettre en majuscule et le nombre de caractère est compris entre 3 et 11.
- Le champ `prenom` est aussi obligatoire et doit respecter une fonction, qu'on a définie, appelée `checkPrenomValidator`.

Angular

La fonction `checkPrenomValidator()`

```
function checkPrenomValidator(control: FormControl): ValidationErrors |
    null {

  const str: string = control.value ?? "";
  if (str[0] >= 'A' && str[0] <= 'Z') {
    return null;
  } else {
    return {
      majuscule: 'Le prénom doit commencer par une lettre en majuscule'
    };
  }

}
```

Angular

Pour afficher le message d'erreur relatif à `id`

```
<div>
    <label for="id">Identifiant</label>
    <input type="text" id="id" formControlName=id>

    <span *ngIf="personneForm.controls['id'].errors && personneForm.controls['id'].errors['
      required']">
           L'identifiant est obligatoire
    </span>
</div>
```

Angular

Pour afficher le message d'erreur relatif à `id`

```
<div>
    <label for="id">Identifiant</label>
    <input type="text" id="id" formControlName=id>

    <span *ngIf="personneForm.controls['id'].errors && personneForm.controls['id'].errors['
      required']">
           L'identifiant est obligatoire
    </span>
</div>
```


Constat

Le message s'affiche au chargement du composant.

Angular

Pour éviter que le message d'erreur s'affiche au chargement du composant, on ajoute

```
<div>
    <label for="id">Identifiant</label>
    <input type="text" id="id" formControlName=id>

    <span *ngIf="personneForm.controls['id'].invalid  &&
                 (personneForm.controls['id'].dirty ||
                  personneForm.controls['id'].touched)">
        <span *ngIf="personneForm.controls['id'].errors && personneForm
          .controls['id'].errors['required']">
            L'identifiant est obligatoire
        </span>
    </span>
</div>
```

Angular

On peut simplifier l'écriture de nos tests en remplaçant `personneForm.controls.id` **par** `id` **et en définissant une méthode qui le retourne dans** `form.ts`

```
<div>
    Identifiant :
    <input type="number" formControlName="id">

    <span *ngIf="id.invalid && (id.dirty || id.touched)">
        <span *ngIf="id.errors && id.errors['required']">
            L'identifiant est obligatoire
        </span>
    </span>
</div>
```

Angular

On peut simplifier l'écriture de nos tests en remplaçant `personneForm.controls.id` **par** `id` **et en définissant une méthode qui le retourne dans** `form.ts`

```
<div>
    Identifiant :
    <input type="number" formControlName="id">

    <span *ngIf="id.invalid && (id.dirty || id.touched)">
        <span *ngIf="id.errors && id.errors['required']">
            L'identifiant est obligatoire
        </span>
    </span>
</div>
```


Définissons le getter de `id` **dans** `form.ts`

```
get id() {
    return this.personneForm.controls['id'];
}
```

Angular

Pour le nom

```
<div>
    <label for="nom">Nom</label>
    <input type="text" id="nom" formControlName=nom>
    <span *ngIf="nom.invalid && (nom.dirty || nom.touched)">
        <span *ngIf="nom.errors">
            <span *ngIf="nom.errors['required']">
                Le nom est obligatoire
            </span>
            <span *ngIf="nom.errors['pattern']">
                Le nom doit contenir au moins 3 caractères (Première lettre en majuscule)
            </span>
        </span>
    </span>
</div>
```

Angular

Pour le prénom

```
<div>
    <label for="prenom">Prénom</label>
    <input type="text" id="prenom" formControlName=prenom>
    <span *ngIf="prenom.invalid && (prenom.dirty || prenom.touched)">
        <span *ngIf="prenom.errors">
            <span *ngIf="prenom.errors['required']; else majuscule">
                Le prénom est obligatoire
            </span>
            <ng-template #majuscule>
                <span>
                    {{ prenom.errors['majuscule'] }}
                </span>
            </ng-template>
        </span>
    </span>
</div>
```

Angular

Et un getter pour chaque `FormControl`

```
get prenom() {
  return this.personneForm.controls['prenom'];
}

get nom() {
  return this.personneForm.controls['nom'];
}
```

Angular

Remarque

- Il est possible d'imbriquer les `FormGroup`
- Par exemple : un `FormGroup` adresse défini dans le `FormGroup` personne

Angular

Dans `form.ts`, définissons les `FormGroup` imbriqués

```
import { Component, OnInit } from '@angular/core';
import { FormControl, FormGroup, Validators } from '@angular/forms';

@Component({
  selector: 'app-form',
  imports: [ReactiveFormsModule],
  templateUrl: './form.html',
  styleUrl: './form.css'
})
export class Form implements OnInit {

  personneForm = new FormGroup({
    id: new FormControl('', Validators.required),
    nom: new FormControl('', [Validators.pattern(/^[A-Z][a-z]{2,10}$/), Validators.required]),
    prenom: new FormControl('', [Validators.required, checkPrenomValidator]),
    adresse: new FormGroup({
      rue: new FormControl(''),
      ville: new FormControl(''),
      codePostal: new FormControl('')
    })
  });

  ngOnInit() {  }

  afficherTout() {
    console.log(this.personneForm.value);
    this.personneForm.reset();
  }
}
```

Angular

Dans `form.html`, ajoutons le formulaire associé au `FormGroup` imbriqué

```
<form [formGroup]="personneForm" (ngSubmit)='afficherTout()'>

  <!-- contenu précédent -->

  <div formGroupName="adresse">
    <h3>Adresse</h3>
    <div>
      <label for="rue">Rue</label>
      <input type="text" id=rue formControlName="rue">
    </div>
    <div>
      <label for="ville">Ville</label>
      <input type="text" id=ville formControlName="ville">
    </div>
    <div>
      <label for="codePostal">Code postal</label>
      <input type="text" id=codePostal formControlName="codePostal">
    </div>
  </div>
  <button type='submit'>Envoyer</button>
</form>
```

Angular

Remarque

- La méthode `setValue()` permet d'initialiser, ou modifier les valeurs de formulaire : il faut préciser une valeur pour chaque `FormControl` du `FormGroup`
- La méthode `patchValue()` permet d'initialiser, ou modifier quelques (ou tous les) `FormControl` du `FormGroup`

Angular

Exemple (dans `ngOnInit()`)

```
this.personneForm.patchValue({
    prenom: 'abruzzi',
    adresse: {
        codePostal: '13000'
    }
});
```

Angular

Exemple (dans `ngOnInit()`)

```
this.personneForm.patchValue({
    prenom: 'abruzzi',
    adresse: {
        codePostal: '13000'
    }
});
```


Les champs `Prénom` et `Code postal` sont initialisés avec les valeurs `abruzzi` et `13000`

Angular

FormBuilder

- Une classe `service` défini par **Angular**
- Donc, pour l'utiliser, il faut l'injecter dans le constructeur
- Il permet de simplifier la construction d'un formulaire en évitant les répétitions de `FormControl`

Dans `form.ts`, on injecte `FormBuilder` dans le constructeur puis on l'utilise pour construire le formulaire

```
import { Component, OnInit } from '@angular/core';
import { Validators, FormBuilder } from '@angular/forms';

@Component({
  selector: 'app-form',
  imports: [ReactiveFormsModule],
  templateUrl: './form.html',
  styleUrl: './form.css'
})
export class Form implements OnInit {

  personneForm =  this.fb.group({
    id: ['', Validators.required],
    nom: ['', [Validators.pattern(/^[A-Z][a-z]{2,10}$/), Validators.required]],
    prenom: ['', [Validators.required, checkPrenomValidator]],
    adresse: this.fb.group({
      rue: [],
      ville: [],
      codePostal: []
    }),
  });

  constructor(private fb: FormBuilder) { }
  ngOnInit(): void {  }

  afficherTout() {
    console.log(this.personneForm.value);
    this.personneForm.reset();
  }
}
```

Angular

Dans `form.html`, rien à changer

```
<form [formGroup]="personneForm" (ngSubmit)='afficherTout()'>

  <!-- contenu précédent -->

  <div formGroupName="adresse">
    <h3>Adresse</h3>
    <div>
      <label for="rue">Rue</label>
      <input type="text" id=rue formControlName="rue">
    </div>
    <div>
      <label for="ville">Ville</label>
      <input type="text" id=ville formControlName="ville">
    </div>
    <div>
      <label for="codePostal">Code postal</label>
      <input type="text" id=codePostal formControlName="codePostal">
    </div>
  </div>
  <button type='submit'>Envoyer</button>
</form>
```

Angular

On peut aussi surveiller l'évolution de notre formulaire grâce à l'attribut `status` (à placer dans le formulaire)

```
<div>
    état : {{ personneForm.status }}
</div>
```

Angular

Exercice

- Modifier le composant `builder` pour afficher les personnes ajoutées en-dessous du formulaire
- À chaque ajout, la nouvelle personne ajoutée apparaît comme dernier élément de la liste des personnes (affichée en-dessous du formulaire)

Angular

Exercice

- Ajoutez un bouton supprimer pour chaque personne affichée.
- En cliquant sur le bouton, la personne associée sera supprimée.

Angular

FormArray

- Défini dans `FormBuilder`
- Il permet de définir un tableau de taille indéterminée de `FormControl`
- Une personne peut pratiquer plusieurs sports (le nombre peut varier d'une personne à une autre) $\Rightarrow$ on peut utiliser `FormArray`

Angular

Dans `form.ts`**, définissons notre** `FormArray`

```
personneForm = this.fb.group({
    id: ['', Validators.required],
    nom: ['', [Validators.pattern(/^[A-Z][a-z]{2,10}$/), Validators.required]],
    prenom: ['', [Validators.required, checkPrenomValidator]],
    adresse: this.fb.group({
        rue: [],
        codePostal: [],
        ville: [],
    }),
    sports: this.fb.array([])
});
```

Angular

Dans `form.ts`**, définissons notre** `FormArray`

```
personneForm = this.fb.group({
    id: ['', Validators.required],
    nom: ['', [Validators.pattern(/^[A-Z][a-z]{2,10}$/), Validators.required]],
    prenom: ['', [Validators.required, checkPrenomValidator]],
    adresse: this.fb.group({
        rue: [],
        codePostal: [],
        ville: [],
    }),
    sports: this.fb.array([])
});
```


Pour afficher instantanément les sports ajoutés par l'utilisateur, on foit retourner notre `FormArray`

```
get sports(): FormArray {
    return this.personneForm.controls['sports'] as FormArray;
}
```

Angular

Dans `form.html`, ajoutons notre `FormArray` à notre formulaire précédent

```
<div formArrayName="sports">
    <h4>Sports</h4>
    <div>
        <button type="button" (click)="ajouterSport()">
            Ajouter un sport
        </button>
    </div>
    @for (sport of sports.controls; track sport) {
    <div>
        <label [for]=`sport${$index}`>Sport {{ $index + 1 }} :</label>
        <input type="text" [formControlName]="$index" [id]=`sport${$index}`>
    </div>
    }
</div>
```

Angular

Définissons maintenant la méthode `ajouterSport()`

```
ajouterSport(): void {
    this.sports.push(this.fb.control(''));
}
```

Angular

Définissons maintenant la méthode `ajouterSport()`

```
ajouterSport(): void {
    this.sports.push(this.fb.control(''));
}
```

Remarque

- On ajoute à notre tableau un nouvel élément vide pour que l'utilisateur puisse saisir un nouveau sport.
- Le sport ajouté par l'utilisateur est lié directement à notre `FormArray`

Angular

Utilisons la méthode `reset()` pour vider les champs du formulaire

```
afficherTout(): void {
    console.log(this.personneForm.value);
    this.personneForm.reset();
}
```

Angular

Utilisons la méthode `reset()` pour vider les champs du formulaire

```
afficherTout(): void {
    console.log(this.personneForm.value);
    this.personneForm.reset();
}
```


Remarque

En ajoutant une première personne avec deux sports, le formulaire sera initialisé avec deux champs sports pour le prochain ajout.

Angular

Utilisons la méthode `clear()` pour remettre à zéro le nombre de champs sports après soumission du formulaire

```
afficherTout(): void {
    console.log(this.personneForm.value);
    this.personneForm.reset();
    this.sports.clear();
}
```

Angular

Ajoutons un validateur aux champs sports

```
ajouterSport(): void {
  this.sports.push(this.fb.control('', Validators.
     required));
}
```

Angular

Affichons le message d'erreur relatif à ce validateur

```
<div formArrayName="sports">
    <h3>Sports </h3>
    <button type=button (click)="ajouterSport()">
        Ajouter sport
    </button>
    <div *ngFor="let sport of sports.controls; let i=index">
        <div>
            <label [for]=`sport${$index}`>Sport {{ $index + 1 }} :</label>
            <input type="text" [formControlName]="$index" [id]=`sport${$index}`>
            <span *ngIf="sportAt(i)?.invalid  && (sportAt(i)?.dirty || sportAt(i)?.touched)">
                <span *ngIf="sportAt(i)?.errors?.['required']">
                    Merci de saisir au moins un caractère
                </span>
            </span>
        </div>
    </div>
</div>
```

Angular

Il nous reste à définir la méthode `sportAt` dans `form.ts`

```
sportAt(i: number) {
  return this.sports.get(i.toString());
}
```

Angular

Exercice

- Ajoutez un bouton supprimer pour chaque sport.
- En cliquant sur le bouton, le sport associé sera supprimé.

Angular

Solution : `form.html`

```
<div formArrayName="sports">
    <h4>Sports</h4>
    <div>
        <button type="button" (click)="ajouterSport()">
            Ajouter un sport
        </button>
    </div>
    @for (sport of sports.controls; track sport) {
    <div>
        Sport : <input type="text" [formControlName]="$index">
        <button type="button" (click)="supprimerSport($index)">
            Supprimer
        </button>
    </div>
    }
</div>
```

Angular

Solution : `form.ts`

```
supprimerSport(i: number) {
    this.sports.removeAt(i)
}
```

Angular

Remarque

`FormArray` est une classe qui peut être aussi utilisée dans un `FormGroup`.

Angular

Exercice

- Dans un nouveau composant, `comment`, créer un formulaire qui permet à une personne de saisir son nom, son prénom ainsi qu'un tableau de commentaire de taille variable.
- Chaque commentaire est composé d'un titre, un contenu et une catégorie.
- En cliquant sur **Ajouter**, les données saisies seront affichées en bas du formulaire et le formulaire sera vidé.
- Aucun champ ne doit être vide à l'ajout, les nom et prénom doivent commencer par une lettre majuscule.

Angular

Signal forms

- API expérimentale introduite dans la version **Angular 21.0.0**.
- Pourrait encore changer avant d'atteindre une version stable, et il est recommandé de ne pas les utiliser dans des applications de production critiques.

Angular

Pour commencer

- créer un composant `signal-formulaire`
- créer un chemin `/signal-form` permettant d'afficher ce composant

Angular

Dans `signal-formulaire.ts`**, déclarons un attribut** `personneModel` **de type** `signal<Personne>`

```
import { Component, signal } from '@angular/core';
import { Personne } from '../../models/personne';

@Component({
  selector: 'app-signal-formulaire',
  imports: [],
  templateUrl: './signal-formulaire.html',
  styleUrl: './signal-formulaire.css',
})
export class SignalFormulaireComponent {
  personneModel = signal<Personne>({
    nom: '',
    prenom: '',
    age: 0
  });
}
```

Angular

Dans `signal-formulaire.ts`**, déclarons un attribut** `personneModel` **de type** `signal<Personne>`

```
import { Component, signal } from '@angular/core';
import { Personne } from '../../models/personne';

@Component({
  selector: 'app-signal-formulaire',
  imports: [],
  templateUrl: './signal-formulaire.html',
  styleUrl: './signal-formulaire.css',
})
export class SignalFormulaireComponent {
  personneModel = signal<Personne>({
    nom: '',
    prenom: '',
    age: 0
  });
}
```

Ici, on déclare une propriété `personneModel`, initialisée avec un objet de type `Personne` vide, en utilisant la fonction `signal`.

Angular

Dans `signal-formulaire.ts`**, déclarons un attribut** `personneModel` **de type** `signal<Personne>`

```
import { Component, signal } from '@angular/core';
import { Personne } from '../../models/personne';
import { form } from '@angular/forms/signals';

@Component({
  selector: 'app-signal-formulaire',
  imports: [],
  templateUrl: './signal-formulaire.html',
  styleUrl: './signal-formulaire.css',
})
export class SignalFormulaireComponent {
  personneModel = signal<Personne>({
    nom: '',
    prenom: '',
    age: 0
  });
  personneForm = form(this.personneModel);
}
```

Angular

Dans `signal-formulaire.ts`**, déclarons un attribut** `personneModel` **de type** `signal<Personne>`

```
import { Component, signal } from '@angular/core';
import { Personne } from '../../models/personne';
import { form } from '@angular/forms/signals';

@Component({
  selector: 'app-signal-formulaire',
  imports: [],
  templateUrl: './signal-formulaire.html',
  styleUrl: './signal-formulaire.css',
})
export class SignalFormulaireComponent {
  personneModel = signal<Personne>({
    nom: '',
    prenom: '',
    age: 0
  });
  personneForm = form(this.personneModel);
}
```

`form` crée une structure de formulaire basée sur les signaux qui gère la synchronisation bidirectionnelle entre le template et l'objet `personneModel`.

Angular

Dans `signal-formulaire.ts`, importons `Field` : directive standalone conçue pour être utilisée dans le template HTML pour lier un élément de formulaire (`<input>`, `<select>`, ou `<textarea>`) à un champ spécifique dans l'objet de formulaire créé par `form()`

```
import { Component, signal } from '@angular/core';
import { Personne } from '../../models/personne';
import { form, Field } from '@angular/forms/signals';

@Component({
  selector: 'app-signal-formulaire',
  imports: [Field],
  templateUrl: './signal-formulaire.html',
  styleUrl: './signal-formulaire.css',
})
export class SignalFormulaireComponent {
  personneModel = signal<Personne>({
    nom: '',
    prenom: '',
    age: 0
  });
  personneForm = form(this.personneModel);
}
```

Angular

Créons le formulaire associé et utilisons `field` pour la liaison bidirectionnelle entre les attributs de `personneForm` et les `<input>`

```
<form (ngSubmit)="afficher()">
    <div>
        <label for="nom">Nom :</label>
        <input id="nom" type="text" [field]="personneForm.nom" />
    </div>

    <div>
        <label for="prenom">Prénom :</label>
        <input id="prenom" type="text" [field]="personneForm.prenom" />
    </div>

    <div>
        <label for="age">Âge :</label>
        <input id="age" type="number" [field]="personneForm.age" />
    </div>

    <button type="submit">Afficher</button>
</form>
```

Angular

Pour récupérer et afficher les données du formulaire

```
import { Component, signal } from '@angular/core';
import { Personne } from '../../models/personne';
import { form, Field } from '@angular/forms/signals';

@Component({
  selector: 'app-signal-formulaire',
  imports: [Field],
  templateUrl: './signal-formulaire.html',
  styleUrl: './signal-formulaire.css',
})
export class SignalFormulaireComponent {
  personneModel = signal<Personne>({
    nom: '',
    prenom: '',
    age: 0
  });
  personneForm = form(this.personneModel);
  afficher() {
    const data = this.personneModel()
    console.log(data);

  }
}
```

Angular

Remarque

En cliquant sur le bouton, la page est complètement rechargée.

Angular

Remarque

En cliquant sur le bouton, la page est complètement rechargée.

Deux solutions

- Utiliser `(ngSubmit)` et importer `FormsModule`
- Sans `FormsModule`, utiliser (`submit`) et empêcher le comportement par défaut avec `preventDefault`.

Angular

Première solution

```
import { Component, signal } from '@angular/core';
import { Personne } from '../../models/personne';
import { Field, form } from '@angular/forms/signals';
import { FormsModule } from '@angular/forms';

@Component({
  selector: 'app-signal-formulaire',
  imports: [Field, FormsModule],
  templateUrl: './signal-formulaire.html',
  styleUrl: './signal-formulaire.css',
})
export class SignalFormulaireComponent {
  personneModel = signal<Personne>({
    nom: '',
    prenom: '',
    age: 0
  });
  personneForm = form(this.personneModel);
  afficher() {
    const data = this.personneModel()
    console.log(data);

  }
}
```

Angular

Rien à changer dans le template

```
<form (ngSubmit)="afficher()">
    <div>
        <label for="nom">Nom :</label>
        <input id="nom" type="text" [field]="personneForm.nom" />
    </div>

    <div>
        <label for="prenom">Prénom :</label>
        <input id="prenom" type="text" [field]="personneForm.prenom" />
    </div>

    <div>
        <label for="age">Âge :</label>
        <input id="age" type="number" [field]="personneForm.age" />
    </div>

    <button type="submit">Afficher</button>
</form>
```

Angular

Deuxième solution : en utilisant `event`

```
import { Component, signal } from '@angular/core';
import { Personne } from '../../models/personne';
import { Field, form } from '@angular/forms/signals';

@Component({
  selector: 'app-signal-formulaire',
  imports: [Field, FormsModule],
  templateUrl: './signal-formulaire.html',
  styleUrl: './signal-formulaire.css',
})
export class SignalFormulaireComponent {
  personneModel = signal<Personne>({
    nom: '',
    prenom: '',
    age: 0
  });
  personneForm = form(this.personneModel);
  afficher(event: SubmitEvent) {
    event.preventDefault();
    const data = this.personneModel()
    console.log(data);

  }
}
```

Angular

Et en remplaçant `ngSubmit` **par** `submit`

```
<form (submit)="afficher($event)">
    <div>
        <label for="nom">Nom :</label>
        <input id="nom" type="text" [field]="personneForm.nom" />
    </div>

    <div>
        <label for="prenom">Prénom :</label>
        <input id="prenom" type="text" [field]="personneForm.prenom" />
    </div>

    <div>
        <label for="age">Âge :</label>
        <input id="age" type="number" [field]="personneForm.age" />
    </div>

    <button type="submit">Afficher</button>
</form>
```

Angular

Pour désactiver le bouton tant que le formulaire est invalide

```
<form (submit)="afficher($event)">
    <div>
        <label for="nom">Nom :</label>
        <input id="nom" type="text" [field]="personneForm.nom" />
    </div>

    <div>
        <label for="prenom">Prénom :</label>
        <input id="prenom" type="text" [field]="personneForm.prenom" />
    </div>

    <div>
        <label for="age">Âge :</label>
        <input id="age" type="number" [field]="personneForm.age" />
    </div>

    <button type="submit" [disabled]="personneForm().invalid()">
        Afficher
    </button>
</form>
```

Définissons un schéma de validation pour les champs de formulaire

```
personneForm = form(this.personneModel, (schema) => {
    required(schema.nom, { message: 'Le nom est obligatoire' });
    required(schema.prenom, { message: 'Le prénom est obligatoire' });
    required(schema.age, { message: 'L\'âge est obligatoire' });
    max(schema.age, 150, { message: 'L\'âge doit être inférieur à 150' });
    min(schema.age, 0, { message: 'L\'âge doit être supérieur à 0' });
});
```

Définissons un schéma de validation pour les champs de formulaire

```
personneForm = form(this.personneModel, (schema) => {
    required(schema.nom, { message: 'Le nom est obligatoire' });
    required(schema.prenom, { message: 'Le prénom est obligatoire' });
    required(schema.age, { message: 'L\'âge est obligatoire' });
    max(schema.age, 150, { message: 'L\'âge doit être inférieur à 150' });
    min(schema.age, 0, { message: 'L\'âge doit être supérieur à 0' });
});
```

Et les imports

```
import { Field, form, max, min, required } from '@angular/forms/signals';
```

Définissons un schéma de validation pour les champs de formulaire

```
personneForm = form(this.personneModel, (schema) => {
    required(schema.nom, { message: 'Le nom est obligatoire' });
    required(schema.prenom, { message: 'Le prénom est obligatoire' });
    required(schema.age, { message: 'L\'âge est obligatoire' });
    max(schema.age, 150, { message: 'L\'âge doit être inférieur à 150' });
    min(schema.age, 0, { message: 'L\'âge doit être supérieur à 0' });
});
```

Et les imports

```
import { Field, form, max, min, required } from '@angular/forms/signals';
```

Pour afficher les messages d'erreur dans le template

```
<div>
    <label for="nom">Nom :</label>
    <input id="nom" type="text" [field]="personneForm.nom" />
    @if (personneForm.nom().touched() && personneForm.nom().invalid()) {
    <div class="erreur">
        @for (err of personneForm.nom().errors(); track err) {
        <div>{{ err.message }}</div>
        }
    </div>
    }
</div>
```

Angular

Remarques

- `personneForm : FieldTree<Personne, ...>` (la structure du formulaire)
  - Structure du formulaire
  - Définition des champs (`nom`, `prenom`, `age`)
  - Règles de validation associées
  - Interaction avec la directive `[field]`
  - Pas d'accès direct à la validité ou aux erreurs
- `personneForm() : FieldState<Personne, ...>` (l'état agrégé du formulaire)
  - État dynamique du formulaire
  - Validité : `valid()`, `invalid()`
  - État utilisateur : `touched()`, `dirty()`
  - Erreurs : `errors()`
  - Valeur complète : `value()`

Angular

Synthèse

- Utilisez **Template-driven Forms** pour les formulaires très simples et statiques, avec peu de champs et une validation simple
- Utilisez **Reactive Forms** pour la majorité des formulaires complexes et toutes les applications en production.
- Signal Forms sont à explorer pour l'apprentissage et les projets non-critiques, en gardant à l'esprit leur statut expérimental.