

FORMULAIRES TEMPLATE-DRIVEN

INTRODUCTION AUX TEMPLATE-DRIVEN FORMS

CONCEPT

Les **formulaire Template-driven** sont une approche basée sur des modèles pour créer des formulaires dans **Angular**.

AVANTAGES ET INCONVÉNIENTS

AVANTAGES

- **Syntaxe simple** et facile à comprendre
- Bon pour les **applications simples** et petits **formulaires**

INCONVÉNIENTS

- **Validation** et **tests** plus difficiles
- Moins de contrôle sur la **logique** et **validation**

SYNTAXE NGMODEL

UTILISATION

La directive **ngModel** permet de lier les éléments de **formulaire** à des propriétés de leur **modèle**.

Exemple :

```
<form>
  <label for="username">Nom d'utilisateur:</label>
  <input type="text" id="username" [(ngModel)]="user.username" name="username">
</form>
```

```
export class AppComponent {
  user = {
    username: ''
  };
}
```


EXEMPLES

```
<input [(ngModel)]="name" />
```

VALIDATION AVEC TEMPLATE-DRIVEN FORMS

VALIDATORS INTÉGRÉS

Angular fournit des validateurs intégrés pour la validation des **formulaires**, tels que `required`, `minlength` et `pattern`.

Validator	Description
<code>required</code>	Vérifie si le champ est non vide
<code>minlength</code>	Vérifie si la longueur minimale est respectée
<code>pattern</code>	Vérifie si la valeur correspond au motif défini

VALIDATORS PERSONNALISÉS

Pour créer des validateurs personnalisés, il faut suivre les étapes suivantes:

1. Créer une **fonction de validation**
2. Lier la **fonction** au **contrôle de formulaire**

SOUMISSION DES FORMULAIRES

GESTION DES ÉVÉNEMENTS

Les événements de soumission de formulaire doivent être gérés en utilisant la directive **(ngSubmit)**.

Exemple :

```
<form (ngSubmit)="onSubmit()">
  ...
  <button type="submit">Envoyer</button>
</form>
```

```
onSubmit() {
  // Traiter la soumission du formulaire
}
```

EXEMPLES

```
<form (ngSubmit)="onSubmit()">
  <!-- Contenu du formulaire -->
  <button type="submit">Envoyer</button>
</form>
```

```
onSubmit() {
  // Traitement des données du formulaire
}
```

FORMULAIRES REACTIVE

INTRODUCTION AUX REACTIVE FORMS

CONCEPT

Les **Reactive forms** sont basés sur la **programmation réactive**, permettant de gérer la logique de formulaire dans les **composants** de manière **déclarative**.

AVANTAGES ET INCONVÉNIENTS

Avantages :

- **Meilleur contrôle** sur la logique
- **Scalabilité**

Inconvénients :

- Plus **complexe** que les Template-driven forms
- **Apprentissage** plus difficile

SYNTAXE FORMCONTROL, FORMGROUP ET FORMARRAY

CRÉATION DE FORMES RÉACTIVES

Pour créer des formes réactives, utilisez les classes `FormControl`, `FormGroup`, et `FormArray`.

- `FormControl` : représente un élément de formulaire unique
- `FormGroup` : permet de grouper plusieurs `FormControl`
- `FormArray` : représente un tableau de `FormControl`

ASSOCIATION AVEC LES ÉLÉMENTS DU FORMULAIRE

Utilisez la syntaxe suivante pour lier un **élément de formulaire** à un `FormControl` ou `FormGroup` :

```
<form [formGroup]="nomForm">  
  <input formControlName="nomFormControl">  
</form>
```

VALIDATION AVEC REACTIVE FORMS

VALIDATORS INTÉGRÉS

Angular fournit plusieurs **validateurs intégrés** pour les **Reactive forms**, tels que :

- **required**
- **minLength**
- **maxLength**
- **pattern**
- **email**

VALIDATORS PERSONNALISÉS

Créez vos propres **validators** pour des règles de validation spécifiques :

```
function monValidator(control: FormControl): { [s: string]: boolean } {  
    // Logique de validation personnalisée  
}
```

SOUMISSION DES FORMULAIRES

GESTION DES ÉVÉNEMENTS

Gérez les **événements** de soumission des **formulaire**s avec les **Reactive forms** :

```
<form [formGroup]="nomForm" (ngSubmit)="soumettreForm()">
  <!-- Éléments du formulaire -->
  <button type="submit">Envoyer</button>
</form>
```

EXEMPLES

Exemple de soumission de formulaire avec **Reactive form** :

```
soumettreForm() {  
  if (this.nomForm.valid) {  
    console.log(this.nomForm.value);  
  } else {  
    // Afficher les erreurs de validation  
  }  
}
```

AFFICHER ET MASQUER DES ÉLÉMENTS DE FORMULAIRE

UTILISATION DE DIRECTIVES CONDITIONNELLES

NGIF

La directive `**ngIf**` permet d'afficher ou de masquer un élément de **formulaire** en fonction d'une **condition**.

```
// Exemple d'utilisation
<div *ngIf="condition">
  Contenu visible si la condition est vraie
</div>
```

EXEMPLES D'UTILISATION

NGIF - AFFICHER UN CHAMP SELON UNE CONDITION

```
<input *ngIf="afficherChamp" type="text" [(ngModel)]="champ">  
<button (click)="afficherChamp = !afficherChamp">Toggle champ</button>
```

UTILISATION DE NGSWITCH

SYNTAXE

La directive `**ngSwitch**` permet de créer des structures conditionnelles, similaires à un `**switch**` en programmation.

```
<div [ngSwitch]="expression">
  <ng-container *ngSwitchCase="'cas1'">Contenu du cas 1</ng-container>
  <ng-container *ngSwitchCase="'cas2'">Contenu du cas 2</ng-container>
  <ng-container *ngSwitchDefault>Contenu par défaut</ng-container>
</div>
```

EXEMPLES D'UTILISATION

```
<div [ngSwitch]="typeDeChamp">
  <input *ngSwitchCase="'text'" type="text" [(ngModel)]="champ">
  <input *ngSwitchCase="'number'" type="number" [(ngModel)]="champ">
  <input *ngSwitchDefault type="text" [(ngModel)]="champ">
</div>

<select [(ngModel)]="typeDeChamp">
  <option value="text">Texte</option>
  <option value="number">Nombre</option>
</select>
```

MANIPULATION DES DONNÉES DU FORMULAIRE

UTILISATION DES FILTRES (PIPE)

Les **Pipes** permettent de **transformer** des données avant de les afficher dans le DOM. Ils peuvent être utilisés pour **filtrer**, **formater** ou **modifier** les données.

Exemple de Pipes	Description
Lowercase	Convertit en minuscules
Uppercase	Convertit en majuscules
Currency	Formate en devise
Date	Formate une date

SYNTAXE

```
{{ valeur | pipeName:paramètre }}
```

EXEMPLES D'UTILISATION

- Mise en **majuscule** : `{{ texte | uppercase }}`
- Formater une **date** : `{{ date | date:'dd/MM/yyyy' }}`
- Trier un **tableau** : `{{ tableau | orderBy:'classement' }}`

SERVICES ANGULAR

Les services sont des **classes** qui encapsulent une **fonctionnalité spécifique**. Ils peuvent être utilisés pour partager des données, communiquer avec une **API** ou effectuer des opérations.

Avantages	Exemples d'utilisation
Réutilisabilité	Partage de données entre composants
Isolation	Interactions avec une API
Modularité	Gestion des erreurs

CRÉATION DE SERVICE

Pour créer un **service Angular**, utilisez la commande :

```
ng g s nom-du-service
```

COMMUNICATION AVEC API

Injectez **HttpClient** dans votre service et créez une fonction pour interagir avec une **API**. Par exemple :

```
import { HttpClient } from '@angular/common/http';

constructor(private http: HttpClient) { }

getData() {
  return this.http.get('https://api.example.com/data');
}
```

EXEMPLES D'UTILISATION

Dans un **composant**, injectez le service et utilisez ses méthodes pour afficher les données du formulaire.

```
import { DataService } from '../data.service';

constructor(private dataService: DataService) { }

ngOnInit() {
  this.dataService.getData().subscribe(data => {
    // Traitez et affichez les données ici
  });
}
```

BONNES PRATIQUES DES FORMULAIRES

GROUPING DES CONTRÔLES DE FORMULAIRE

Il est important de **grouper** et d'**organiser** les contrôles de formulaire pour améliorer la **lisibilité** et la **maintenance**.

UTILISATION DE `FormGroup` ET `FormArray`

- `FormGroup` : permet de **regrouper** plusieurs instances de `FormControl`
- `FormArray` : permet de gérer un **tableau dynamique** de contrôles

EXEMPLES D'UTILISATION

```
// FormGroup
let group = new FormGroup({
  firstName: new FormControl(),
  lastName: new FormControl()
});

// FormArray
let array = new FormArray([
  new FormControl(),
  new FormControl(),
  new FormControl()
]);
```


ORGANISER LES PARTIES DU FORMULAIRE

Utiliser des **éléments spécifiques** pour organiser et compartimenter les formulaires.

- Les **FormGroup** sont utilisés pour regrouper ensemble des champs de formulaire connexes.
- Les **FormControl** permettent de gérer les états des champs de formulaire individuels.
- Les **FormArray** sont utilisés pour gérer un ensemble de contrôles de formulaire de manière dynamique.

UTILISATION DE `ngContainer`

`ngContainer` est un élément qui ne génère pas de rendu **HTML** supplémentaire:

```
<form>
  <ng-container *ngIf="condition">
    <input ...>
    <select ...>
  </ng-container>
</form>
```

EXEMPLES D'UTILISATION

```
<ng-container *ngIf="isEditMode">  
  <button>Save</button>  
</ng-container>  
<ng-container *ngIf="!isEditMode">  
  <button>Edit</button>  
</ng-container>
```

TEST DES FORMULAIRES

Tester les formulaires est important pour s'assurer que l'**application répond aux exigences** et fonctionne correctement.

TESTING DES TEMPLATE-DRIVEN FORMS

1. Tester les composants individuels des formulaires (**validation**, **apparence**, etc.).
2. Tester l'intégration des composants avec leurs **modèles**.

TESTING DES REACTIVE FORMS

1. Tester les **FormBuilder**, **FormControl**, et **FormGroup**.
2. Tester les services qui interagissent avec les **formulaires**.

