

# Programme de Formation Angular 18

## Module 1: Introduction à Angular

- **Introduction à Angular**
  - Qu'est-ce qu'Angular?
  - Architecture d'Angular : Modules, Composants, et Services
  - Configuration de l'environnement de développement
  - Introduction à Angular CLI
- **Installation et Configuration**
  - Installation de Node.js et Angular CLI
  - Création d'une nouvelle application Angular
  - Structure des dossiers et fichiers dans une application Angular
  - Introduction au fichier `angular.json`

## Module 2: Les Bases d'Angular

- **Composants Angular**
  - Création d'un composant
  - Structure d'un composant (template, style, et logique)
  - Communication entre les composants (Input, Output)
  - Cycle de vie des composants
- **Templates et Data Binding**
  - Interpolation et Property Binding
  - Event Binding
  - Two-Way Data Binding avec `ngModel`
  - Directives structurelles
  - Directives d'attributs (`[ngStyle]`, `[ngClass]`)
- **Services et Dependency Injection**
  - Introduction aux services
  - Création et utilisation d'un service
  - Injection de dépendances dans Angular
  - Le rôle des providers

## Module 3: Gestion des Formulaires

- **Formulaires Template-Driven**
  - Création d'un formulaire
  - Validation des formulaires
  - Gestion des erreurs et affichage des messages d'erreur
  - Utilisation de `ngForm`, `ngModel`, et les directives associées
- **Formulaires Reactive**
  - Introduction aux formulaires réactifs
  - Création d'un formulaire réactif avec `FormBuilder`
  - Validation réactive et gestion des états des champs
  - `FormGroup`, `FormControl`, et `FormArray`

## Module 4: Routage et Navigation

- **Routage de base**

- Configuration des routes dans Angular
- Navigation entre les routes
- Paramètres de route et query parameters
- Protection des routes avec `AuthGuard`
- **Routage avancé**
  - Chargement paresseux (Lazy Loading) des modules
  - Préchargement des modules
  - Utilisation des routes enfants

## Module 5: Gestion des Données et HTTP

- **HTTP Client et API REST**
  - Introduction à `HttpClientModule`
  - Envoi de requêtes HTTP (GET, POST, PUT, DELETE)
  - Gestion des Observables avec `RxJS`
  - Traitement des erreurs HTTP
- **RxJS et Programmation Réactive**
  - Introduction à `RxJS`
  - Concepts de base : Observable, Observer, et Subscription
  - Opérateurs `RxJS` courants : `map`, `filter`, `mergeMap`, `switchMap`
  - Manipulation des flux de données asynchrones

## Module 6: Gestion de l'État avec `NgRx`

- **Introduction à `NgRx`**
  - Concepts de base : Store, Actions, Reducers, Selectors
  - Configuration de `NgRx` dans un projet Angular
  - Création et gestion du Store
  - Utilisation d'Effects pour gérer les effets secondaires

## Module 7: Tests Unitaires et End-to-End

- **Tests unitaires avec Jasmine et Karma**
  - Introduction à Jasmine et Karma
  - Écriture de tests pour les composants, services, et directives
  - Test des formulaires et des services HTTP
- **Tests End-to-End avec Protractor**
  - Introduction à Protractor
  - Écriture de tests E2E pour les composants et le routage
  - Configuration et exécution des tests

## Module 8: Optimisation et Déploiement

- **Optimisation de la Performance**
  - Optimisation du chargement des modules
  - Compression et minimisation des bundles
  - Utilisation de Change Detection Strategy
- **Déploiement d'une Application Angular**
  - Build et préparation pour la production
  - Déploiement sur Firebase, Netlify, ou un serveur web

- Configuration d'un serveur pour le déploiement (Nginx, Apache)

## Module 9: Projet Final

- **Développement d'une application complète**
  - Définition des exigences du projet
  - Mise en œuvre des concepts appris
  - Présentation du projet et retour d'expérience

## Durée et Méthodologie

- **Durée:** Le programme peut être étalé sur 8 à 10 semaines, en fonction du rythme d'apprentissage des participants.
- **Méthodologie:** Chaque module doit comprendre des sessions théoriques suivies de travaux pratiques. Les participants doivent être encouragés à poser des questions et à résoudre des exercices pour renforcer leur compréhension.

## Matériel Requis

- Ordinateur avec un environnement de développement (VS Code, Node.js, Angular CLI)
- Accès à une API REST pour les exercices pratiques (ou la création d'une API fictive)
- Documentation officielle Angular et ressources en ligne pour approfondir les sujets

### Fin de programme

## I. Introduction à Angular

### 1. Qu'est-ce qu'Angular?

Angular est un **framework JavaScript open-source** développé par Google pour créer des applications web dynamiques à page unique (**Single Page Applications - SPA**). Angular utilise **TypeScript** comme langage principal, une version typée de JavaScript, ce qui permet d'améliorer la maintenabilité et la lisibilité du code.

Angular se distingue par plusieurs fonctionnalités clés :

- **Data Binding bidirectionnel** : Les modifications apportées aux données dans le modèle se répercutent automatiquement dans l'interface utilisateur et vice-versa.
- **Injection de dépendances** : Un mécanisme qui permet de créer et de gérer les services dans une application de manière modulaire.
- **Modularité** : Angular segmente les fonctionnalités dans des modules, composants, services, etc., facilitant ainsi l'organisation et la réutilisation du code.

## 2. Architecture d'Angular : Modules, Composants, et Services

Angular suit une architecture basée sur les composants et les modules pour organiser l'application.

- **Modules** : Ce sont des conteneurs logiques qui regroupent des composants, des services et d'autres ressources. Le module principal est `AppModule` dans lequel tous les composants sont déclarés. D'autres modules peuvent être créés pour diviser l'application en sous-parties logiques (par exemple, `CustomerModule`, `AdminModule`).
- **Composants** : Ce sont des blocs de construction réutilisables de l'interface utilisateur. Chaque composant possède une vue (HTML), une logique (TypeScript), et un style (CSS). Le composant racine est `AppComponent`, à partir duquel tous les autres composants dérivent.
- **Services** : Les services fournissent des fonctionnalités partagées entre les différents composants, comme l'accès aux données ou la gestion de la logique métier. Ils sont généralement injectés dans les composants à l'aide de l'injection de dépendances.

## 4. Configuration de l'environnement de développement

Pour commencer avec Angular, il est nécessaire de configurer l'environnement de développement.

### Outils principaux :

- **Node.js et npm** : Angular dépend de Node.js pour exécuter certains outils en ligne de commande. npm (Node Package Manager) permet d'installer les bibliothèques nécessaires, y compris Angular lui-même. [lien node js](#)
- **Angular CLI (Command Line Interface)** : Il s'agit d'un outil qui permet de créer, générer et déployer des applications Angular avec des commandes simples.

# Installation global d'Angular CLI sur l'ordi : `npm install -g @angular/cli`

Il est aussi possible d'installer Angular dans un dossier spécifique dans votre ordinateur sans qu'il ne soit installé globalement

- 1- Créer un dossier
- 2- Ouvrez le avec VS code
- 3- Initialiser un projet node avec la commande : `npm init -y`
- 4- Installer CLI dans ce dossier : `npm install @angular/cli@18 --save-dev`
- 5- Vérifier la version : `npx ng version`
- 6- Créer un nouveau projet dans ce dossier : `npx ng new nom-du-nouveau-projet`
- 7- Déplacez vous dans le projet et vérifier la version : `npx ng version`

**Editeur de code** : Utilisez un éditeur moderne tel que **Visual Studio Code**, qui prend en charge TypeScript, et offre de nombreuses extensions pour améliorer la productivité.

## 5. Introduction à Angular CLI

L'**Angular CLI** (Command Line Interface) est un outil puissant qui facilite la gestion des projets Angular. Avec Angular CLI, vous pouvez générer des composants, des services, des modules et bien d'autres entités sans avoir à écrire la structure de base manuellement.

Voici quelques-unes des commandes de base :

**Créer un nouveau projet : `ng new mon-projet`**

**Servir l'application** (lancer un serveur de développement local) : `ng serve`

**Générer un composant** : `ng generate component mon-composant`

**Générer un service** : `ng generate service mon-service`

## II. Structure des dossiers et fichiers dans une application Angular

Après avoir généré un projet avec Angular CLI, vous trouverez une structure de fichiers bien définie :

- **src/** : C'est le dossier principal où se trouve le code source de l'application. Il contient :
  - **app/** : Le cœur de l'application avec le composant racine (`AppComponent`).
  - **assets/** : Utilisé pour stocker les fichiers statiques comme les images, les fichiers CSS globaux, etc.
  - **environments/** : Contient des fichiers de configuration d'environnement (ex. `environment.ts` pour le développement et `environment.prod.ts` pour la production).
- **app/** : Ce répertoire contient les modules et composants Angular :
  - **app.component.ts** : Le fichier TypeScript qui définit la logique du composant racine.
  - **app.component.html** : Le fichier de template HTML associé au composant.
  - **app.component.css** : Le fichier de style pour ce composant spécifique.
- **node\_modules/** : Contient toutes les dépendances installées via npm.
- **angular.json** : Fichier de configuration principal d'Angular (voir la section suivante).
- **package.json** : Décrit les dépendances du projet, les scripts npm, et la version du projet.
- **tsconfig.json** : Contient la configuration TypeScript.

### Introduction au fichier **angular.json**

Le fichier **angular.json** est un fichier crucial dans un projet Angular. Il définit la configuration du projet et des différents environnements. Il permet de personnaliser le comportement d'Angular CLI.

Voici quelques sections clés du fichier **angular.json** :

- **Projects** : Liste les applications et bibliothèques Angular incluses dans le projet. Chaque application a sa propre configuration.

```

json

{
  "projects": {
    "nom-du-projet": {
      ...
    }
  }
}

```

- **Architect** : Définit les configurations pour construire, tester, et lancer le projet. Par exemple, pour la construction de l'application :

```

json

"architect": {
  "build": {
    "builder": "@angular-devkit/build-angular:browser",
    "options": {
      "outputPath": "dist/nom-du-projet",
      "index": "src/index.html",
      "main": "src/main.ts",
      "polyfills": "src/polyfills.ts",
      "tsConfig": "tsconfig.app.json",
      "assets": ["src/assets", "src/favicon.ico"],
      "styles": ["src/styles.css"],
      "scripts": []
    }
  }
}

```

- **Serve** : Contient la configuration utilisée pour exécuter `ng serve` (pour lancer l'application en mode développement).

- **Build Options** : Définit les fichiers à inclure lors de la compilation, tels que les fichiers principaux TypeScript (`main.ts`), le fichier HTML (`index.html`), les styles, et les polyfills.

Le fichier **angular.json** est donc essentiel pour contrôler le comportement de l'application Angular lors de son développement et de son déploiement. Vous pouvez modifier les configurations pour répondre aux besoins spécifiques de votre projet, comme l'ajout de chemins de styles globaux ou de scripts externes.

## Les Bases d'Angular

### Composants Angular

Les composants sont les blocs de construction fondamentaux d'une application Angular. Chaque composant est une unité encapsulée de l'interface utilisateur, qui combine la logique, le template HTML, et les styles.

#### 1. Création d'un composant

Pour créer un composant, vous pouvez utiliser Angular CLI. Supposons que nous voulons créer un composant appelé `user-profile`.

- Commande pour créer un composant :

**ng generate component user-profile** ou en version courte :

**ng g c user-profile**

Cette commande génère un dossier `user-profile` avec les fichiers suivants :

- `user-profile.component.ts` : Fichier TypeScript contenant la logique du composant.
- `user-profile.component.html` : Fichier HTML pour le template du composant.
- `user-profile.component.css` : Fichier CSS pour les styles du composant.
- `user-profile.component.spec.ts` : Fichier pour les tests unitaires du composant.

#### 2. Structure d'un composant (template, style, et logique)

Chaque composant Angular se compose de trois parties principales : le template, les styles, et la logique.

**Template (HTML)** : Définit la structure HTML du composant.

**Exemple - `user-profile.component.html` :**

html

```
<div class="user-profile">
  <h1>{{ user.name }}</h1>
  <p>Email: {{ user.email }}</p>
</div>
```

**Styles (CSS) :** Contient les styles spécifiques au composant.

**Exemple - user-profile.component.css :**

CSS

```
.user-profile {
  border: 1px solid #ccc;
  padding: 16px;
  border-radius: 8px;
}

.user-profile h1 {
  color: #2c3e50;
}
```

- **Logique (TypeScript) :** Gère la logique du composant, les propriétés, et les méthodes.

**Exemple - user-profile.component.ts :**



typescript

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

@Component({
  selector: 'app-user-profile',
  templateUrl: './user-profile.component.html',
  styleUrls: ['./user-profile.component.css']
})
export class UserProfileComponent implements OnInit {
  user = { name: 'John Doe', email: 'john.doe@example.com' };

  constructor() { }

  ngOnInit(): void { }
}
```

### 3. Communication entre les composants (Input, Output)

La communication entre les composants peut se faire de différentes manières, principalement à l'aide des décorateurs @Input et @Output.

- **@Input** : Permet à un composant parent de passer des données à un composant enfant.

C'est dans le composant enfant qu'on doit mettre le décorateur **@Input**

**Exemple - Composant parent : app.component.html :**

html

```
<app-user-profile [user]="currentUser"></app-user-profile>
```

app.component.ts

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


@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  currentUser = { name: 'John Doe', email: 'john.doe@example.com' };
}

You, 1 second ago • Uncommitted changes
```

Exemple - Composant enfant : user-profile.component.ts :

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

@Component({
  selector: 'app-user-profile',
  templateUrl: './user-profile.component.html',
  styleUrls: ['./user-profile.component.css'],
  // Permet au composant d'être autonome (standalone),
  // utilisé sans être déclaré dans un module Angular.
  standalone: true,
  // Liste des imports nécessaires pour ce composant.
  imports: []
})
export class UserProfileComponent implements OnInit {
  user = Input.required<{ name: string; email: string; }>();
  constructor() {}
  ngOnInit(): void {}
}
```

 Copier le code

typescript

```
@Input()user:{name:string;email:string;}; // Angular 16/17  
user=input.required<{name:string;email:string}>(); // Angular 18
```

Le HTML du composant enfant :

html

```
<div class="user-profile">  
  <h1>{{ user.name }}</h1>  
  <p>Email: {{ user.email }}</p>  
</div>
```

**@Output** : Permet à un composant enfant d'émettre des événements vers son composant parent, toujours le decorateur doit etre dans l'enfant

**Exemple - Composant enfant : user-profile.component.ts :**

typescript

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

@Component({
  selector: 'app-user-profile',
  templateUrl: './user-profile.component.html',
  styleUrls: ['./user-profile.component.css']
})
export class UserProfileComponent {
  @Output() userClicked = new EventEmitter<void>();

  onUserClick() {
    this.userClicked.emit();
  }
}
```

```
userClicked = output<void>()
```

Pour Angular18

Exemple - Composant parent : app.component.html :

```
<app-user-profile (click)="handelUserClicked()"></app-user-profile> <
```

Exemple - Composant parent : app.component.ts :

typescript

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  currentUser = { name: 'John Doe', email: 'john.doe@example.com' };

  handleUserClick() {
    console.log('User profile clicked!');
  }
}
```

#### 4. Cycle de vie des composants

Les composants Angular passent par plusieurs phases au cours de leur cycle de vie. Angular fournit plusieurs hooks de cycle de vie qui permettent d'intervenir à différentes étapes du cycle de vie du composant.

- **ngOnInit()** : Appelé après l'initialisation des propriétés du composant.

**Exemple :**

typescript

```
ngOnInit(): void {
  // Initialisation de données
}
```

**ngOnChanges()** : Appelé lorsqu'une ou plusieurs propriétés d'entrée (@Input) du composant changent.

typescript

```
ngOnChanges(changes: SimpleChanges): void {  
    // Réagir aux changements des propriétés d'entrée  
}
```

**ngOnDestroy()** : Appelé juste avant que le composant ne soit détruit. Utilisé pour libérer les ressources ou annuler les abonnements.

**Exemple :**

typescript

```
ngOnDestroy(): void {  
    // Nettoyage, désabonnement  
}
```

**ngAfterViewInit()** : Appelé après que la vue du composant a été initialisée. Utile pour des manipulations de DOM ou des intégrations avec des bibliothèques tierces.

**Exemple :**

typescript

```
ngAfterViewInit(): void {  
    // Manipuler le DOM  
}
```

- **Templates et Data Binding**

## Interpolation

**Objectif :** Afficher une variable dans le template.

1. **Création du composant `simple` :**

Création du composant `simple` :

```
typescript

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

@Component({
  selector: 'app-simple',
  templateUrl: './simple.component.html',
  styleUrls: ['./simple.component.css']
})
export class SimpleComponent {
  title = 'Hello, Angular!';
}
```

- `simple.component.html` :

```
html

<h1>{{ title }}</h1>
```


Cela affichera "Hello, Angular!" dans un élément `<h1>`.

## Property Binding

**Objectif :** Lier une propriété d'un élément HTML à une variable du composant.

1. **Code du composant :**
  - `simple.component.ts` :

typescript

 Copier le code

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

@Component({
  selector: 'app-simple',
  templateUrl: './simple.component.html',
  styleUrls: ['./simple.component.css']
})
export class SimpleComponent {
  imageUrl = 'https://angular.io/assets/images/logos/angular/angular.svg';
}
```

Logo Angular : <https://angular.io/assets/images/logos/angular/angular.svg>

simple.component.html :

html

```
<img [src]="imageUrl" alt="Angular Logo">
```

Cela affichera une image dont la source est définie par la variable `imageUrl`.

## 2. Event Binding

**Objectif :** Réagir à un événement utilisateur en appelant une méthode du composant.

### 1. Code du composant :

- `simple.component.ts` :



typescript

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

@Component({
  selector: 'app-simple',
  templateUrl: './simple.component.html',
  styleUrls: ['./simple.component.css']
})
export class SimpleComponent {
  message = '';

  showMessage() {
    this.message = 'Button clicked!';
  }
}
```

simple.component.html:

html

```
<button (click)="showMessage()">Click me</button>
<p>{{ message }}</p>
```

Cela affichera un bouton, et lorsque vous cliquez dessus, le message "Button clicked!" sera affiché sous le bouton.

### 3. Two-Way Data Binding avec `ngModel`

**Objectif :** Lier une entrée utilisateur à une variable du composant et refléter les modifications.

#### 1. Code du composant :

- `simple.component.ts` :

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

@Component({
  selector: 'app-simple',
  templateUrl: './simple.component.html',
  styleUrls: ['./simple.component.css'],
  standalone: true, // Si vous utilisez des composants autonomes
  imports: [FormsModule]
})
export class SimpleComponent {
  name = '';
}
```

simple.component.html :

```
html

<input [(ngModel)]="name" placeholder="Enter your name">
<p>Your name is: {{ name }}</p>
```

Cela affichera un champ de saisie. Le texte entré sera affiché sous le champ en temps réel.

#### 4. Directives structurales (@If, @For, .....)

- @If

**Objectif :** Afficher un élément en fonction d'une condition.

##### 1. Code du composant :

- simple.component.ts :

typescript

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

@Component({
  selector: 'app-simple',
  templateUrl: './simple.component.html',
  styleUrls: ['./simple.component.css']
})
export class SimpleComponent {
  isVisible = true;
}
```

simple.component.html:

typescript

```
<button (click)="isVisible = !isVisible">Toggle Visibility</button>
@if (isVisible) {
  <p>This paragraph is visible only if isVisible is true.</p>
}
```

- @for

simple.component.ts:

```
public matieres : Matiere[] = [
  {nom : 'Java', code : '001', nbHeure : 21},
  {nom : 'C++', code : '002', nbHeure : 51},
  {nom : 'JS', code : '003', nbHeure : 41}
]
```

simple.component.ts :

```
<table>
  <tr>
    <th>Nom</th>
    <th>Code</th>
    <th>NbHeures</th>
  </tr>
  @for (matiere of matieres; track $index) {
    <tr>
      <td>{{matiere.nom}}</td>
      <td>{{matiere.code}}</td>
      <td>{{matiere.nbHeure}}</td>
    </tr>
  }
</table>
```

Le mot **track** est un identifiant unique pour chaque éléments de la boucle, angular vas se servir de la valeur pour identifier les éléments de façon unique, ici **index** est l'index de chaque élément dans le tableau, on pouvait mettre 'matiere.code' si 'code' est identifiant unique.

## 5. Directives d'attributs ([ngStyle], [ngClass])

Les directives d'attributs `ngStyle` et `ngClass` sont des outils puissants en Angular pour manipuler dynamiquement les styles CSS d'un élément HTML en fonction des données de votre application. Cela permet de créer des interfaces utilisateur plus interactives et réactives.

- `ngStyle` : Il faudra importer `ngStyle` d'abord dans le `ts`

```
standalone: true,
imports: [FormsModule, NgStyle],
```

Dans le html :

```
<h2 [ngStyle]="{'color': isImportant ? 'red' : 'black'}">{{message}}</h2>
```

En supposant que 'isImportant' est bien déclaré dans le ts comme boolean

- `ngClass` : Il faudra importer `ngClass` d'abord dans le ts

```
standalone: true,  
imports: [FormsModule, NgStyle, NgClass],
```

Dans le Html

```
<button [ngClass]="{'btn-primary': isFormValid, 'btn-danger': !isFormValid}">Soumettre</button>
```

Il faudra aussi que ces classes existent, soit vous les créer dans le fichier css du composant ou utiliser les classes de votre framework CSS installé dans le projet Angular.

- **Services et Dependency Injection**

## Introduction aux Services

En Angular, les services sont des classes qui encapsulent des fonctionnalités réutilisables, comme l'accès à des données, les appels d'API, ou la gestion d'événements. Ils permettent de séparer les préoccupations et de rendre votre code plus modulaire et testable.

## Création et Utilisation d'un Service

### 1. Création du Service

Créons un service nommé `EtudiantService` avec un composant `EtudiantComponent` qui simule la récupération d'informations sur des étudiants à partir d'une API :

La commande : `'ng g c Etudiant'` génère un composant `EtudiantComponent`

La commande : `'ng g s Etudiant'` génère un service `EtudiantService`

Dans le `etudiant.service.ts` :

```
import { Injectable, inject } from '@angular/core';
import { Observable, of } from 'rxjs';
import { Etudiant } from '../Utils/Personnes/Etudiant';
@Injectable({
  providedIn: 'root'
})
export class EtudiantService {
  constructor() { }

  public etudiants : Etudiant[] = [
    {nom: 'Doe', prenom: 'John', date: new Date(), matieres : ['commerce', 'foot']},
    {nom: 'Alex', prenom: 'Martin', date: new Date("01/01/1998"), matieres : ['vente', 'agricol']},
    {nom: 'Alane', prenom: 'Wilson', date: new Date('01/01/2002'), matieres : ['jouer', 'courire']},
  ]
  getEtudiants(): Observable<Etudiant[]> {
    return of(this.etudiants)
  }
}
```

Avec un fichier d'interface `etudiant` :

```
export interface Etudiant
{
  nom : string;
  prenom : string;
  date : Date;
  matieres : string[];
}
```

Dans le composant etudiant :

```
import { Component, OnInit, inject } from '@angular/core';
import moment from 'moment';
import { Etudiant } from '../Utils/Personnes/Etudiant';
import { Matiere } from '../Utils/Personnes/Matiere';
import { MatiereComponent } from '../matiere/matiere.component';
import { EtudiantService } from '../etudiant.service';
import { AsyncPipe } from '@angular/common';
@Component({
  selector: 'app-etudiant',
  standalone: true,
  imports: [MatiereComponent, AsyncPipe],
  templateUrl: './etudiant.component.html',
  styleUrls: ['./etudiant.component.css']
})
export class EtudiantComponent implements OnInit {
  etudiantListe : Etudiant[] = []
  private serviceEtudiant = inject(EtudiantService)
  items$ = this.serviceEtudiant.getEtudiants()
  constructor(){
  }
}
```

**items\$**: C'est une propriété du composant, souvent de type Observable. Le \$ à la fin est une convention courante pour indiquer qu'il s'agit d'un Observable. Un Observable est un objet qui émet des valeurs au fil du temps, ce qui est idéal pour gérer les données asynchrones comme les requêtes HTTP.

L'observable n'est toujours pas appelé d'abord tant que **items\$** n'est pas utilisé dans le HTML

Exemple :

```

<table>
  @for (item of items$ | async ; track $index) {
    <tr>
      <td>{{ item.nom }}</td>
      <td>{{ item.prenom }}</td>
      <td>{{ item.date }}</td>
    </tr>
    @for (item of item.matières; track $index) {
      <tr>
        <td>{{item}}</td>
      </tr>
    }
  }
</table>

```

Autre façon de faire aussi:

```

export class EtudiantComponent implements OnInit {
  etudiantListe : Etudiant[] = []
  // private serviceEtudiant = inject(EtudiantService)
  // items$ = this.serviceEtudiant.getEtudiants()
  constructor(private service : EtudiantService){
  }
  getEtudiant()
  {
    this.service.getEtudiants().subscribe({
      next:(response)=>{
        this.etudiantListe = response;
      },error:(err)=>{
        console.log(err);
      }
    })
  }
}

```



On remplace le `items$` par `'etudiantListe'` dans le HTML en enlevant le `async`, ça donne le même résultat

```
<table>
  @for (item of etudiantListe ; track $index) {
    <tr>
      <td>{{ item.nom }}</td>
      <td>{{ item.prenom }}</td>
      <td>{{ item.date }}</td>
    </tr>
    @for (item of item.matières; track $index) {
      <tr>
        <td>{{item}}</td>
      </tr>
    }
  }
</table>
```

Le fait de mettre la déclaration du service dans le constructeur fait que l'instance est automatiquement créée.

La méthode **`subscribe()`** est utilisée dans Angular (via RxJS) pour **consommer un Observable**. Lorsque vous effectuez une requête HTTP (ou toute autre opération asynchrone), elle retourne généralement un **Observable**, et vous utilisez `subscribe()` pour réagir aux événements émis par cet Observable, comme la réception de données ou une erreur.

`subscribe()` prend un objet comme paramètre avec plusieurs options, ici :

- **next** : Ce que l'Observable doit faire lorsqu'il reçoit une valeur (par exemple, la réponse de l'API).
- **error** : Ce que l'Observable doit faire lorsqu'une erreur survient (par exemple, si l'appel HTTP échoue).

Et il ne faut pas oublier de configurer le `app.config` en ajoutant la méthode `'provideHttpClient()'` :

You, 2 minutes ago | 1 author (You)

```
import { ApplicationConfig, provideZoneChangeDetection } from '@angular/core';
import { provideRouter } from '@angular/router';

import { routes } from './app.routes';
import { provideHttpClient } from '@angular/common/http';

export const appConfig: ApplicationConfig = {
  providers: [provideZoneChangeDetection({ eventCoalescing: true }),
    provideRouter(routes), provideHttpClient(),]
};
```

L'utilisation des verbes http exemple de get

**Le service :**

```
src > app > src > etudiant.service.ts > EtudiantService > getEtudiants

import { Injectable, inject } from '@angular/core';
import { Observable, of } from 'rxjs';
import { Etudiant, Student } from './Utils/Personnes/Etudiant';
import { HttpClient } from '@angular/common/http';

@Injectable({
  providedIn: 'root'
})
export class EtudiantService {
  constructor(private http : HttpClient) { }
  getEtudiants(): Observable<any> {
    return this.http.get('https://randomuser.me/api/');
    // renvoi toutes les données depuis un serveur
    // distant qu'on ignore carement la technologie
    // d'ou la notion d'API
  }
}
```

Le composant :

```
export class EtudiantComponent implements OnInit {
  etudiantListe : any[] = []
  constructor(private service : EtudiantService){
  }
  getEtudiant()
  {
    this.service.getEtudiants().subscribe({
      next:(response)=>{
        this.etudiantListe = response.results;
      },error:(err)=>{
        console.log(err);
      }
    })
  }
  ngOnInit(): void {
    this.getEtudiant()
  }
}
```

Le HTML :

```
@for (item of etudiantListe ; track $index) {
  <p>Gender : {{ item.gender }}</p>
  <p>Email : {{ item.email }}</p>
  <p>Phone : {{ item.phone }}</p>
}
```

## Module 3: Gestion des Formulaires

### Formulaires Template-Driven dans Angular

Les formulaires Template-Driven permettent de créer et de gérer des formulaires HTML de manière intuitive en utilisant des directives Angular. Dans ce type de formulaire, la logique de formulaire repose principalement sur le template (HTML).

#### Exemple basé sur l'interface `Etudiant`

L'interface `Etudiant` que nous allons utiliser ressemble à ceci :

```
etudiant: Etudiant = {  
  nom: '',  
  prenom: '',  
  date: new Date(),  
  matieres: [] // Initialisation à un tableau vide  
};
```

### 1. Création d'un formulaire

Voici comment créer un formulaire Template-Driven pour l'inscription d'un étudiant.

#### HTML

```
<form #etudiantForm="ngForm" (ngSubmit)="onSubmit(etudiantForm)">  
  <div>  
    <label for="nom">Nom :</label>  
    <input type="text" id="nom" name="nom" [(ngModel)]="etudiant.nom" required />  
  </div>  
  <div>  
    <label for="prenom">Prénom :</label>  
    <input type="text" id="prenom" name="prenom" [(ngModel)]="etudiant.prenom" required />  
  </div>  
  <div>  
    <label for="date">Date de Naissance :</label>  
    <input type="date" id="date" name="date" [(ngModel)]="etudiant.date" required />  
  </div>  
  <button type="submit" [disabled]="etudiantForm.invalid">Enregistrer</button>  
</form>
```

**ngModel** est une directive fournie par Angular qui joue un rôle essentiel dans la gestion des formulaires Template-Driven. Elle permet de lier les éléments du formulaire aux propriétés du modèle de données. Voici une explication détaillée de **ngModel** et de son fonctionnement.

### Qu'est-ce que **ngModel** ?

- **Two-Way Data Binding** : **ngModel** implémente le **binding bidirectionnel** entre le formulaire (ou les contrôles du formulaire) et les données dans le composant. Cela signifie que lorsque l'utilisateur modifie la valeur d'un champ de formulaire, cette valeur est automatiquement mise à jour dans le modèle de données du composant. Inversement, si le modèle change dans le composant, la vue (le formulaire) se met également à jour.

### Qu'est-ce que **#etudiantForm="ngForm"** ?

1. **Référence de template** : Le **#etudiantForm** est une variable de référence de template. Dans Angular, une référence de template permet d'accéder à une instance d'un objet ou d'un composant à l'intérieur du template. Dans ce cas, **etudiantForm** fait référence à l'objet du formulaire créé par Angular.
2. **Directive ngForm** : La partie **"ngForm"** indique que la variable de référence **etudiantForm** doit être associée à l'instance de la directive **ngForm**. Cela permet d'accéder à toutes les propriétés et méthodes du formulaire via la variable de référence.

### Comment cela fonctionne-t-il ?

Lorsque vous ajoutez **#etudiantForm="ngForm"** à votre balise **<form>**, Angular lie le formulaire à l'instance **ngForm**, ce qui vous permet de :

- **Accéder aux données du formulaire** : Vous pouvez récupérer les valeurs des contrôles du formulaire à partir de l'instance de **ngForm**.
- **Vérifier l'état de validité du formulaire** : Vous pouvez facilement vérifier si le formulaire est valide ou invalide.
- **Gérer les erreurs** : Vous pouvez afficher des messages d'erreur basés sur l'état du formulaire.

Il ne faut surtout pas oublier d'importer le modul '**FormsModule**' suivant :

```
@Component({
  selector: 'app-etudiant',
  standalone: true,
  imports: [FormsModule],
  templateUrl: './etudiant.component.html',
  styleUrls: ['./etudiant.component.css']
})
```

Quand on soumet le formulaire il faudra appeler une methode qui vas recevoir le formulaire et faire ce qu'on a besoin de faire comme ceci :

```
onSubmit(forme: any) {  
  if (forme.valid) {  
    const etudiant = forme.value; // Récupère l'étudiant soumis  
    // Récupération des étudiants déjà enregistrés  
    // (ou un tableau vide si aucun n'existe)  
    const etudiants = JSON.parse(localStorage.getItem('etudiants') || '[]');  
    // Ajout du nouvel étudiant au tableau  
    etudiants.push(etudiant);  
    // Enregistrement dans localStorage  
    localStorage.setItem('etudiants', JSON.stringify(etudiants));  
    console.log('Étudiant enregistré:', etudiant);  
  }  
}
```

Ici on a simuler une petite base de donnée qui enregistre les etudiants dans la session du navigateur, c'est du pure javascript que chacun peut facilement reconnaitre.

L'aperçu des donnée saisie dans le formulaire:

```
Angular is running in development mode.  
▼ Object { nom: "Sow", prenom: "Mamadou Dara", date: "2024-10-10" }  
  date: "2024-10-10"  
  nom: "Sow"  
  prenom: "Mamadou Dara"  
  ► <prototype>: Object { _ }
```

On peut faire de ces données tout ce qu'on voudra

## Gestion des erreurs et affichage des messages d'erreur :

Nous allons gérer des messages d'erreur personnalisés pour chaque champ, en fonction des erreurs spécifiques qui se produisent.

```

<div>
  <label for="nom">Nom :</label>
  <input type="text" id="nom" name="nom" minlength="3" [(ngModel)]="etudiant.nom" required />
  @if (etudiantForm.submitted && etudiantForm.controls['nom'].errors) {
    <div>
      @if (etudiantForm.controls['nom'].errors['required']) {
        <small class="text-danger">Le nom est obligatoire.</small>
      }
      @if (etudiantForm.controls['nom'].errors['minlength']) {
        <small class="text-danger">Le nom doit contenir au moins 3 caractères.</small>
      }
    </div>
  }
</div>

```

## Résumé du comportement général

- Si l'utilisateur tente de soumettre le formulaire sans remplir le champ `nom`, le message **"Le nom est obligatoire"** s'affichera sous le champ.
- Si l'utilisateur entre un nom de moins de 3 caractères et soumet le formulaire, le message **"Le nom doit contenir au moins 3 caractères"** sera affiché.
- Angular prend en charge la gestion de l'état des formulaires (`etudiantForm`) et de chaque champ (`etudiantForm.controls['nom']`), ce qui permet de personnaliser l'affichage des erreurs en fonction de la validation du formulaire.

## Formulaires Reactive

### 1. Introduction aux formulaires réactifs

Les formulaires réactifs offrent plus de contrôle et une meilleure gestion des validations, car ils sont pilotés par le code TypeScript plutôt que par le template.

### 2. Création d'un formulaire réactif avec FormBuilder

Voici un exemple d'un formulaire réactif pour un étudiant :

```
etudiantForm: FormGroup;

constructor(private fb: FormBuilder) {
  this.etudiantForm = this.fb.group({
    nom: ['', [Validators.required, Validators.minLength(3)]],
    prenom: ['', Validators.required],
    date: ['', Validators.required]
  });
}
```

## Explication des éléments :

1. **etudiantForm: FormGroup;**
  - **etudiantForm** est une instance de **FormGroup**.
  - Un **FormGroup** est une collection de **FormControl** (champs de formulaire), et il permet de regrouper plusieurs contrôles sous une seule entité.
  - Dans ce cas, le **FormGroup** va contenir les contrôles pour les champs **nom**, **prenom**, et **date**.
2. **constructor(private fb: FormBuilder)**
  - **FormBuilder** est un service fourni par Angular pour faciliter la création de formulaires réactifs.
  - Le **constructor** injecte ce service via **private fb: FormBuilder**.
  - Le **FormBuilder** permet de créer facilement des instances de **FormGroup** et **FormControl** sans avoir à les instancier manuellement.
3. **this.etudiantForm = this.fb.group({ ... })**
  - Ici, nous utilisons **this.fb.group()** pour définir les différents champs du formulaire (**nom**, **prenom**, **date**) ainsi que leurs **validators** (les règles de validation).
  - Chaque champ est défini comme un **FormControl** avec des valeurs initiales et des validateurs.

Il faut importer le **ReactiveFormsModule** dans le composant:

```
@Component({
  selector: 'app-etudiant',
  standalone: true,
  imports: [FormsModule, ReactiveFormsModule],
  templateUrl: './etudiant.component.html',
  styleUrls: ['./etudiant.component.css']
})
```



## L'aperçu du formulaire:

```
<form [formGroup]="etudiantForm" (ngSubmit)="onSubmit()">
  <div>
    <label for="nom">Nom :</label>
    <input type="text" id="nom" formControlName="nom" />
    @if (etudiantForm.get('nom')?.invalid && etudiantForm.get('nom')?.touched) {
      <div>
        @if (etudiantForm.get('nom')?.errors?.['required']) {
          <small class="text-danger">
            Le nom est obligatoire.
          </small>
        }
        @if (etudiantForm.get('nom')?.errors?.['minlength']) {
          <small class="text-danger">
            Le nom doit contenir au moins 3 caractères.
          </small>
        }
      </div>
    }
  </div>
</form>
```

Ainsi de suite pour tous les autres champs

## Explication des elements :

Le champ d'entrée utilise `formControlName="nom"`, ce qui signifie qu'il est relié à un contrôleur de formulaire appelé `nom` dans le groupe de formulaire (`FormGroup`) qui gère l'état et les validations de ce champ.

- `etudiantForm.get('nom')?.invalid && etudiantForm.get('nom')?.touched` : Cette condition vérifie si le champ `nom` est invalide **et** s'il a été touché (c'est-à-dire que l'utilisateur a interagi avec ce champ). Si les deux conditions sont vraies, une zone d'erreur s'affiche.
- `etudiantForm.get('nom')?.errors?.['required']` : Cette condition vérifie si l'erreur correspond à la validation `required`, c'est-à-dire que le champ est obligatoire. Si c'est le cas, le message "Le nom est obligatoire." est affiché.
- `etudiantForm.get('nom')?.errors?.['minlength']` : Cette condition vérifie si l'erreur correspond à la validation de longueur minimale. Si le champ contient moins de 3 caractères, le message "Le nom doit contenir au moins 3 caractères." est affiché.

Ici on n'a pas besoin d'envoyer le formulaire a la méthode appelée, quand le formulaire est soumis, automatiquement le formulaire est instancier dans la variable '**etudiantForm**',

Une fois que le formulaire soumis on a accès a toutes les données du formulaire s'il est valide, d'ailleurs c'est possible de désactiver le bouton de soumission tan que le formulaire n'es pas valide (voir le cours précédent).

```
onSubmit() {  
  if (this.etudiantForm.valid) {  
    console.log(this.etudiantForm.value);  
  } else {  
    console.log("Formulaire invalide");  
  }  
}
```

## FormArray :

Le **FormArray** dans Angular est une structure utilisée pour gérer un ensemble dynamique de contrôles de formulaire, souvent utilisée pour des listes d'éléments répétitifs. Elle permet de regrouper plusieurs contrôles similaires au sein d'un tableau, comme des champs pour des matières dans un formulaire étudiant, des lignes de commandes dans un bon de commande, etc.

Commençont par la logique du ts

```
etudiantForm: FormGroup;  
constructor(private fb: FormBuilder) {  
  this.etudiantForm = this.fb.group({  
    nom: ['teste', [Validators.required, Validators.minLength(3)]],  
    prenom: ['', Validators.required],  
    date: ['', Validators.required],  
    matieres: this.fb.array([]) // FormArray pour les matières  
  });  
}
```

Un getter qui retourne le tableau

```
// Getter pour accéder au FormArray des matières
get matieres(): FormArray {
    return this.etudiantForm.get('matieres') as FormArray;
}
```

### Rôle de la méthode `get matieres()`

1. **Accès pratique** : Cette méthode utilise un getter pour simplifier l'accès au **FormArray** dans le template HTML et dans le TypeScript. Plutôt que d'écrire à chaque fois :
2. **Retourne le **FormArray** correctement typé** : En utilisant `as FormArray`, la méthode s'assure que la valeur retournée est explicitement un **FormArray**, ce qui permet d'utiliser toutes les méthodes spécifiques à **FormArray** comme `push()`, `removeAt()`, ou `controls`. Cela est particulièrement utile pour éviter les erreurs de typage.

Ajout dynamique des matieres :

```
// Méthode pour ajouter une matière
ajouterMatiere() {
    const matiereGroup = this.fb.group({
        nom: ['', Validators.required],
        note: ['', [Validators.required, Validators.min(1), Validators.max(20)]]
    });
    this.matieres.push(matiereGroup);
}
```

### Rôle et Explication de `ajouterMatiere()`

1. **Création d'un **FormGroup** pour une matière** : La méthode commence par créer un **FormGroup** pour représenter une matière. Un **FormGroup** est un groupe de contrôles de formulaire qui contient plusieurs champs, ici les champs `nom` (pour le nom de la matière) et `note` (pour la note de la matière).
2. **Ajout du **FormGroup** au **FormArray**** : Une fois le **FormGroup** créé, il est ajouté au **FormArray** nommé `matieres`. Pour ajouter un élément à un **FormArray**, on utilise la méthode `push()`.
3. **Effet dans le formulaire** :

- Chaque fois que `ajouterMatiere()` est appelée, une nouvelle entrée pour une matière (avec les champs `nom` et `note`) sera ajoutée au formulaire.
- Cela est particulièrement utile si vous ne savez pas à l'avance combien de matières un étudiant aura, et que vous voulez permettre à l'utilisateur d'en ajouter autant qu'il en a besoin.

Si on peut ajouter il faut forcément pouvoir supprimer aussi !! 😊😊

```
// Méthode pour supprimer une matière
supprimerMatiere(index: number) {
  this.matieres.removeAt(index);
}
```

### Rôle de la méthode `supprimerMatiere(index: number)`

1. **Accès à l'indice de la matière :** La méthode prend un paramètre `index`, qui correspond à la position de la matière à supprimer dans le `FormArray` `matieres`. Chaque matière ajoutée dans le formulaire a un index unique dans ce tableau.
  - `index: number` : Ce paramètre représente la position exacte de l'élément dans le `FormArray`. Par exemple, si vous avez trois matières, les index iront de 0 à 2.
2. **Suppression d'un élément du `FormArray` :** La méthode utilise la fonction `removeAt()` fournie par Angular pour supprimer un élément à un index spécifique dans le `FormArray`.
3. **Effet dans le formulaire :**
  - Lorsque vous appelez `supprimerMatiere(index)`, la matière correspondante est supprimée du tableau des matières (`FormArray`) et du formulaire affiché à l'utilisateur.
  - Le formulaire est automatiquement mis à jour et ne montre plus l'élément supprimé.

### Utilisation dans le template HTML :

```

<div formArrayName="matieres">
  <button type="button" (click)="ajouterMatiere()">Ajouter une matière</button>
  @for (matiere of matieres.controls; track $index) {
    <div [formGroupName]="$index">|
      <label for="matiere-nom-{{ $index }}">Nom de la matière :</label>
      <input type="text" id="matiere-nom-{{ $index }}" formControlName="nom" />
      @if (matiere.get('nom')?.invalid && matiere.get('nom')?.touched) {
        <small class="text-danger">Le nom de la matière est obligatoire.</small>
      }<label for="matiere-note-{{ $index }}">Note :</label>
      <input type="number" id="matiere-note-{{ $index }}" formControlName="note" />
      @if (matiere.get('note')?.invalid && matiere.get('note')?.touched) {
        @if (matiere.get('note')?.errors?.['required']) {
          <small class="text-danger">La note est obligatoire.</small>
        }@if (matiere.get('note')?.errors?.['min']) {
          <small class="text-danger">La note doit être au moins 1.</small>
        }@if (matiere.get('note')?.errors?.['max']) {
          <small class="text-danger">La note ne peut pas dépasser 20.</small>
        }
      } <button type="button" (click)="supprimerMatiere($index)">Supprimer</button>
    </div>
  }
</div>

```

Cette partie est un formulaire pour les matieres mais qui est inclus dans le grand formulaire, une capture total du formulaire va donner une image trop grande aussi.

Aperçu du rendu:

Nom : teste  
 Prénom :  
 Date de Naissance : jj / mm / aaaa  
 Ajouter une matière  
 Nom de la matière : Note : Supprimer  
 Enregistrer

Logs Data:

```
onSubmit() {  
  if (this.etudiantForm.valid) {  
    console.log(this.etudiantForm.value);  
  } else {  
    console.log("Formulaire invalide");  
  }  
}
```

Angular is running in development mode.

```
▼ Object { nom: "Sow", prenom: "Mamadou Dara", date: "1984-01-01", matieres: (1) [...] }  
  date: "1984-01-01"  
  ▼ matieres: Array [ {...} ]  
    ▼ 0: Object { nom: "Angular", note: 9.75 }  
      nom: "Angular"  
      note: 9.75  
      ► <prototype>: Object { _ }  
      length: 1  
      ► <prototype>: Array []  
    nom: "Sow"  
    prenom: "Mamadou Dara"  
    ► <prototype>: Object { _ }
```