




ib
cegos

Dynamiser des pages web avec Angular



Programme de la formation



1. Introduction

- ☐ Evolution du web
- ☐ Les nouveaux Frameworks
- ☐ Single Page Application

2. ES6+ et TypeScript

- ☐ Rappels Javascript ES6+
- ☐ Typescript


Programme de la formation



3. Angular: Principes et fonctionnement

- ☐ Présentation du framework
- ☐ Architecture de Angular
- ☐ Classe composant
- ☐ Cycle de vie d'un composant
- ☐ Interpolation et data binding
- ☐ Directives
- ☐ Les variables de template
- ☐ Les composants enfants
- ☐ UI: Bootstrap

Programme de la formation

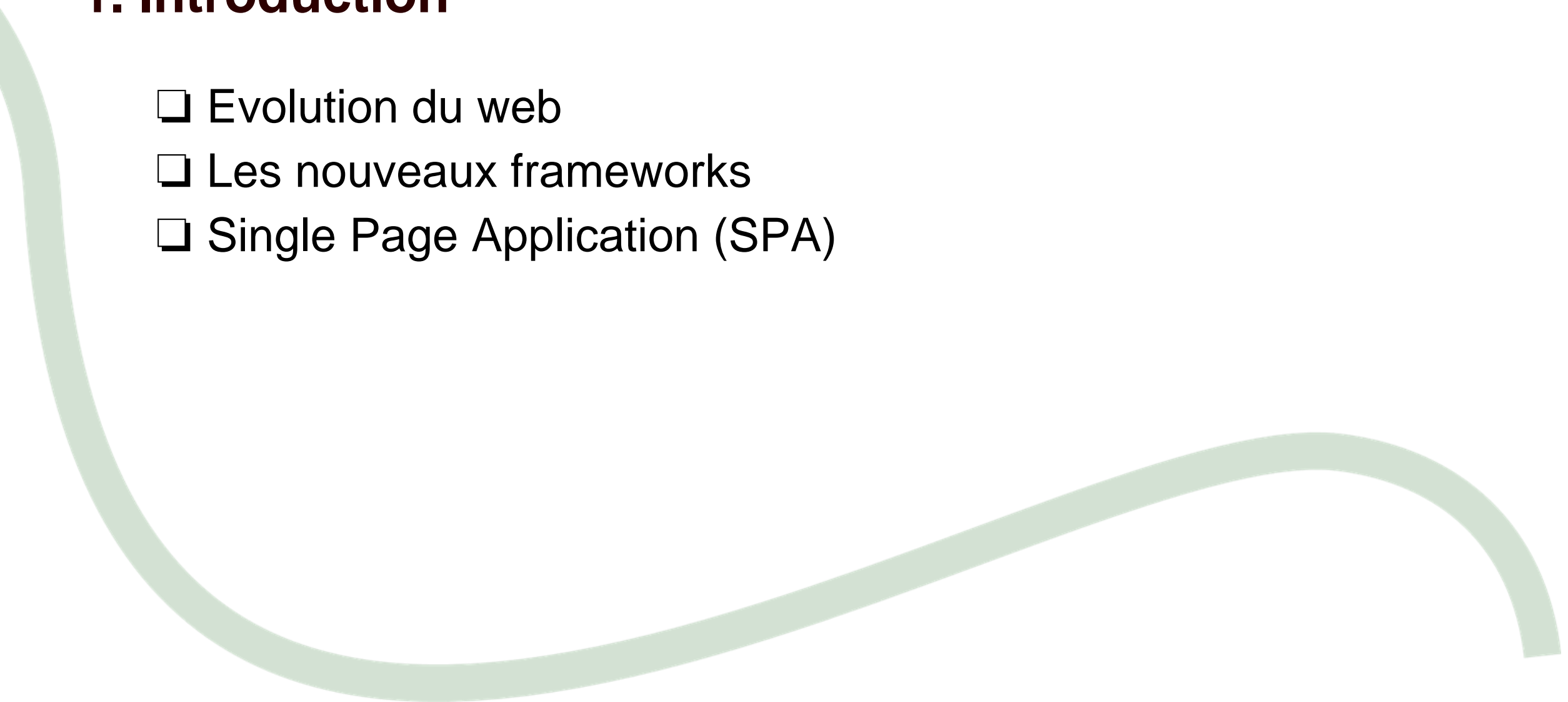


4. Angular: gestion de l'environnement

- ☐ Formulaires: Template Driven Form et Data Driven Form
- ☐ Validation des formulaires
- ☐ Les observables
- ☐ Les services
- ☐ Le router
- ☐ Les pipes
- ☐ Le Guard
- ☐ Interceptor



1. Introduction

- ❑ Evolution du web
 - ❑ Les nouveaux frameworks
 - ❑ Single Page Application (SPA)
- 



❑ Evolution du web

Ces dernières années nous sommes passés des sites web traditionnels aux applications web plus interactives et complexes.



Dans le but d'offrir des expériences utilisateurs plus riches, plus rapides, plus sécurisées et plus accessibles.

Ceci grâce à l'utilisation des technologies avancées et à l'adoption de meilleurs pratiques de développement.

- **Single Page Application (SPA):** chargement dynamique du contenu du page sans recharger toute une page.
- **Frameworks JS avancé:** structure robuste, fonctionnalités avancées
- **APIs avancées:** Permettent aux clients une interaction avec des données.



❑ Les nouveaux frameworks

Un framework est une structure logicielle conçue pour faciliter et accélérer le processus de développement d'applications web.

Il fournit un ensemble de bibliothèques, de modules, de conventions et de bonnes pratiques prédéfinies pour simplifier les tâches courantes (CLI, State, requêtes HTTP, etc).

- **NestJS**: Framework Nodejs pour la construction d'applications serveur robustes et évolutives.
- **ReactJS**: Framework permettant de construire des interfaces utilisateurs spécifiques.
- **Angular**: Framework puissant et polyvalent qui simplifie le processus de développement

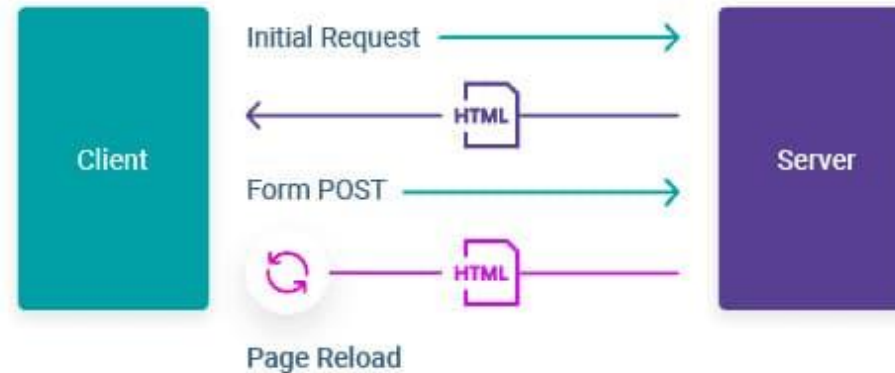


❑ Single Page Application (SPA)

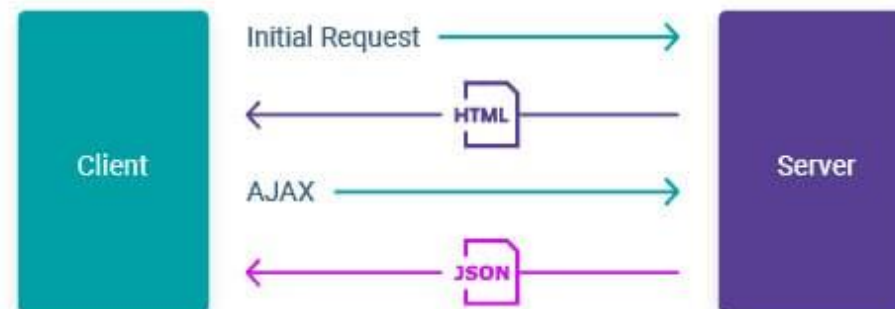
Une application monopage (SPA) est une application web qui consiste en une seule page HTML.

Au lieu de rafraîchir la page entière après chaque interaction de l'utilisateur, seules les données qui doivent être mises à jour déclenchent un rafraîchissement partiel.

Multi-Page Lifecycle

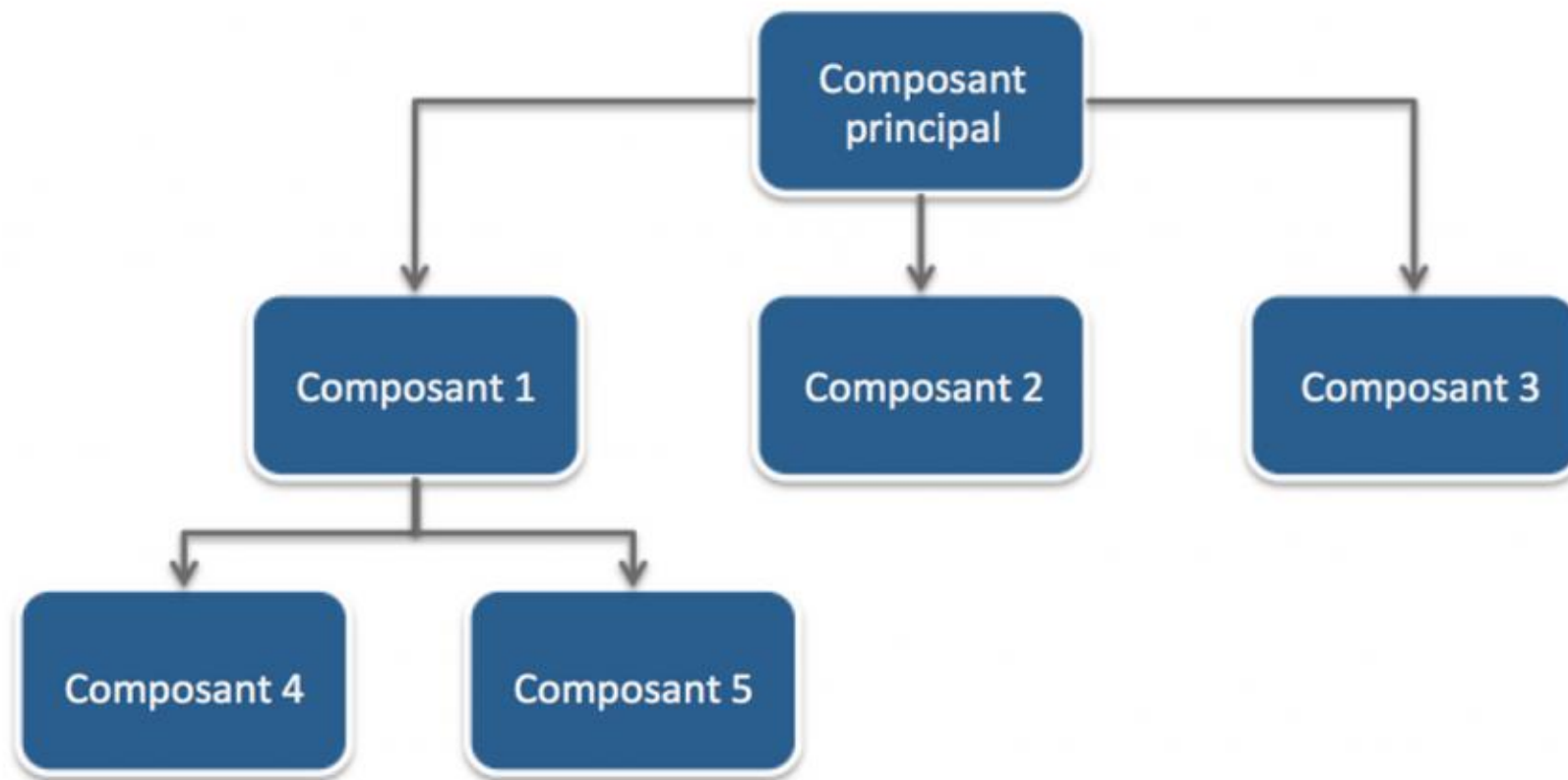


SPA Lifecycle



❑ Single Page Application (SPA)

Généralement un SPA est un ensemble de petits blocs (composants) qui sont affichés dynamiquement selon les interactions de l'utilisateur (formulaire, changement de pages via le routing, etc)





Single Page Application (SPA)

B Pricing example Features Enterprise Support Pricing

Pricing

Quickly build an effective pricing table for your potential customers with this Bootstrap example. It's built with default Bootstrap components and utilities with little customization.

Free	Pro	Enterprise
\$0/mo 10 users included 2 GB of storage Email support Help center access Sign up for free	\$15/mo 20 users included 10 GB of storage Priority email support Help center access Get started	\$29/mo 30 users included 15 GB of storage Phone and email support Help center access Contact us

B © 2017–2024

Features	Resources	About
Cool stuff	Resource	Team
Random feature	Resource name	Locations
Team feature	Another resource	Privacy
Stuff for developers	Final resource	Terms

Header

Banner

Container

Card

Footer

2. ES6 et TypeScript

- ❑ Rappels Javascript ES6+
 - ❑ TypeScript
- 



❑ Rappels Javascript ES6+

JavaScript a été créé par Brendan Eich en 1985. Javascript ES6 (ECMAScript) est la sixième version majeure du standard pour javascript.

```
<!DOCTYPE html>
<html>
  <head>
    <title>My Website Title</title>
    <script src="app.js"></script>
  </head>
  <body>
    <h1></h2>
  </body>
</html>
```



❑ Rappels Javascript ES6+

JavaScript a été créé par Brendan Eich en 1985. Javascript ES6 (ECMAScript) est la sixième version majeure du standard pour javascript.

ES6+ apporte de nombreuses nouvelles fonctionnalités et améliorations au langage, ce qui en fait une mise à jour importante et largement adoptée.

→ Déclarations des variables avec **let** et **const**.

```
2  const YEAR = '2024';  
3  
4  let carName = 'Peugeot';
```

→ Arrow functions

```
6  const add1 = function (a,b) {  
7    return a + b;  
8  }  
9  
10 // arrow function  
11 const add2 = (a, b) => a + b;  
12
```

→ Les paramètres par défaut

```
2  const increment = function (nber, inc = 1) {  
3    return nber + inc;  
4  }  
5  
6  increment(3) // 4  
7  increment(3, 3) // 6  
8
```



❑ Rappels Javascript ES6

→ Destructuring: extraction des données

```
2  const data = {  
3    |   carName: "Peugeot",  
4    |   year: "2024"  
5  }  
6  
7  // classique  
8  let carName1 = data.carName; // Peugeot  
9  
10 // avec Destructuring  
11 let { carName } = data; // Peugeot  
12 // personnalisation de la variable  
13 let { carName: newName } = data; // Peugeot  
14
```

→ Spread Operator: opérateur de propagation permettant de décomposer des données

```
2  const sum = function (a,b,c) {  
3    |   return a + b + c;  
4  }  
5  // classique  
6  const sum1 = sum(1, 2, 3) // 6  
7  
8  // avec le Spread Operator (...)  
9  const numbers = [1, 2, 3];  
10 const sum2 = sum(...numbers) // 6  
11
```

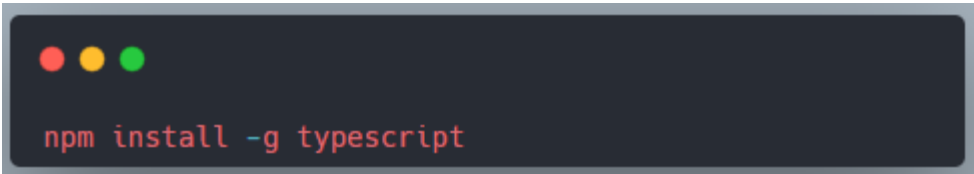


❑ TypeScript

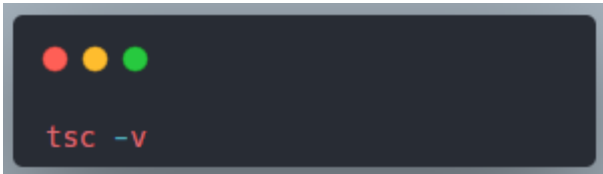
TypeScript est un langage de programmation développé par Microsoft qui a pour but d'améliorer et de sécuriser la production de code JavaScript.

Typescript a un typage fort qui facilite le débogage. Un code javascript valide est un code typescript valide.

Pour installer typescript, vous devez avoir au préalable **node** installé sur votre PC.



```
npm install -g typescript
```



```
tsc -v
```

Un code Typescript s'écrit dans des fichiers avec une extension **.ts** par exemple **app.ts**. Typescript vient avec un compilateur (TSC)



TypeScript

Le code TypeScript est transpilé en JavaScript, pouvant ainsi être interprété par n'importe quel navigateur web ou moteur JavaScript.



```
tsc app.js  
// tsc app.js - outDir ./dist
```



```
<!DOCTYPE html>  
<html>  
<head>  
  <title>My Website Title</title>  
</head>  
<body>  
  <h1>Hello World!</h1>  
  <script src="dist/app.js" ></script>  
</body>  
</html>
```

tsconfig.json est un fichier de configuration pour les projets TypeScript. Il vous permet de spécifier les options du compilateur, d'inclure/exclure des fichiers et de configurer d'autres paramètres pour votre projet TypeScript.



```
tsc --init
```




TypeScript

→ Création d'une variable

Mot clé	Nom de la variable	Type de la variable	Valeur de la variable
let	maVariable	number	= 10

→ Typescript supporte plusieurs types

```
let num: number = 10;
let str: string = "Hello";
let bool: boolean = true;
let anyValue: any = 10; // Utilisation de 'any'

let nullValue: null = null;
let undefinedValue: undefined = undefined;

let arr: number[] = [1, 2, 3];

enum Color { Red, Green, Blue };
let color: Color = Color.Red;
```



La vérification des types s'effectue à la compilation et non à l'exécution



TypeScript

→ Les fonctions

```
function addition(x: number, y: number): number {  
    return x + y;  
}  
  
const result: number = addition(1, 20);  
  
console.log(result); // 21
```

→ Paramètre optionnel et par défaut

```
function sayHello(name?: string) {  
    if(name) {  
        console.log(`Hello ${name}!`);  
    } else {  
        console.log(`Hello world!`);  
    }  
}  
  
sayHello(); // Hello world!  
sayHello("Test"); // Hello Test!
```

```
function sayHello(name: string = "word") {  
    console.log(`Hello ${name}!`);  
}  
  
sayHello(); // Hello world!  
sayHello("Test"); // Hello Test!
```



❏ TypeScript

→ Modularité du code

```
// file: greeting.ts
export function greet(name: string): string { // export file
  return `Hello, ${name}`;
}

// file: app.ts
import { greet } from './greeting.ts'; // import file

console.log(greet("John")); // Output: Hello, John!
```

- 🔔 Le mot clé **export** permet de rendre une fonction, une variable, une classe accessible pour un import dans un autre fichier.
- 🔔 Le mot clé **import** permet d'importer un élément depuis un autre fichier.



TypeScript

TypeScript supporte les concepts de POO permettant une bonne structuration et organisation du code.

→ Classe

```
class User {
  name: string;

  constructor (name: string) {
    this.name = name;
  }

  displayName() {
    console.log("Hello, my name is " + this.name);
  }
}

const user = new User('Alice');
user.displayName(); // sous la console: Hello, my name is Alice
```

→ Interface

```
interface IUser {
  name: string;
  age: number;
};

// via une classe
class Employee implements IUser {
  name: string;
  age: number;
}

// via les paramètres d'une fonction
function introduceEmployee(data: IUser) {
  console.log(`Mon nom est ${data.name} et âgé de ${data.age} ans.`);
}
```

En TypeScript:

- Les **classes** sont utilisées pour créer des objets avec des propriétés et des méthodes.
- Les **interfaces** sont utilisées pour décrire la structure d'un objet sans fournir d'implémentation.



Atelier

Activité



– TD.1 Transpiler un code Typescript

- Dans ce TD, vous allez transpiler un code typescript donné et rajouter une fonctionnalité.



3. Angular: Principes et fonctionnement

- ❑ Présentation du framework
- ❑ Architecture Angular
- ❑ Classe composant
- ❑ Cycle de vie d'un composant
- ❑ Interpolation et data binding
- ❑ Directives
- ❑ Variable de template
- ❑ Les composants enfants
- ❑ UI: Bootstrap



❑ Présentation du framework

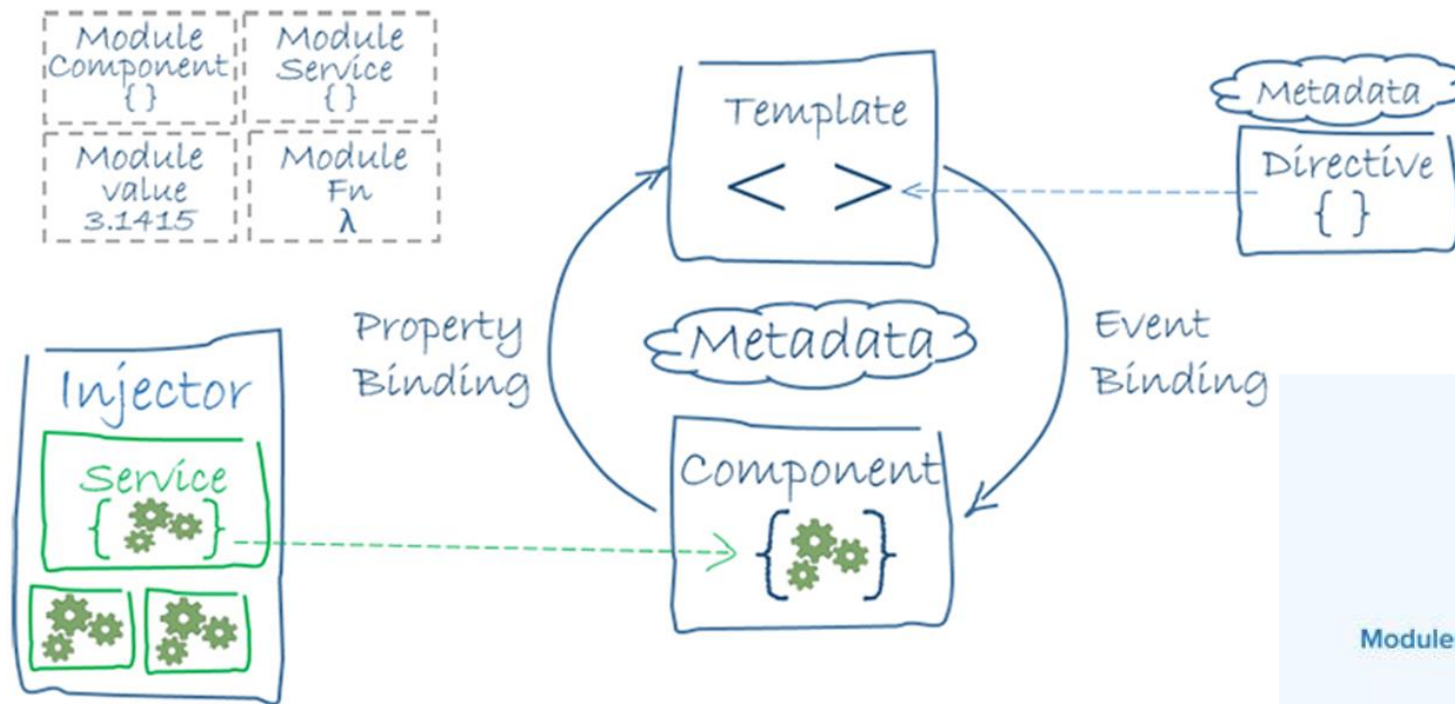
- Angular est un framework (créé en 2016) open source développé et maintenu par google.
- Largement utilisé pour le développement d'applications web et mobiles côté client, Angular est basé sur le concept de composants.
- Le langage utilisé pour implémenter des applications Angular est Typescript (depuis Angular 2).
- Angular CLI est un outil de ligne de commande de Angular qui facilite le processus de développement d'une application.
- La CLI propose des commandes permettant de générer des composants, modules, services, etc..
- Pour installer la CLI, on exécute la commande suivante

```
npm install -g @angular/cli  
// npm install -g @angular/cli@16.2.0
```

```
ng version
```

Architecture Angular

Les éléments de base de Angular sont les composants



❏ Architecture Angular

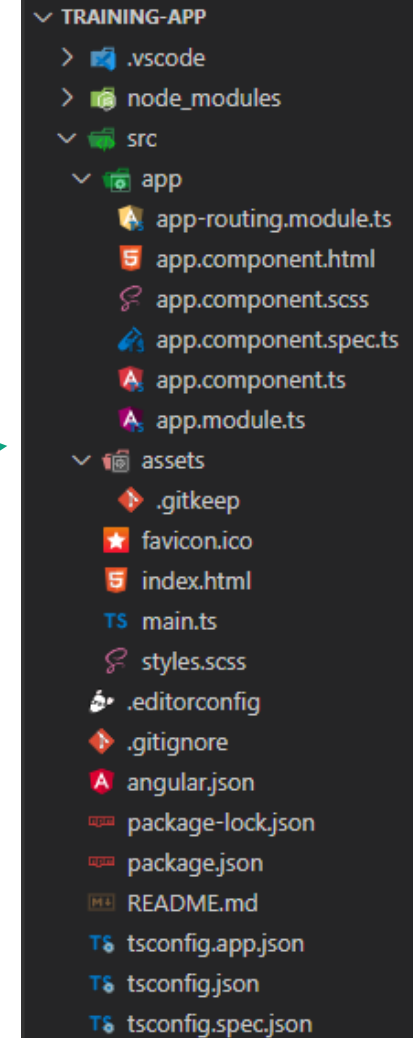
→ Pour créer un projet angular on utilise la commande

```
ng new app_name  
ng new app_name --style scss --routing
```

→ Voici la structure par défaut d'une application "training-app" généré

→ Pour lancer une application on utilise la commande suivante:

```
// cd app_name  
npm start
```



The screenshot shows a file explorer view of a project named 'TRAINING-APP'. The structure is as follows:

- TRAINING-APP
 - .vscode
 - node_modules
 - src
 - app
 - app-routing.module.ts
 - app.component.html
 - app.component.scss
 - app.component.spec.ts
 - app.component.ts
 - app.module.ts
 - assets
 - .gitkeep
 - favicon.ico
 - index.html
 - main.ts
 - styles.scss
 - .editorconfig
 - .gitignore
 - angular.json
 - package-lock.json
 - package.json
 - README.md
 - tsconfig.app.json
 - tsconfig.json
 - tsconfig.spec.json

Exercice:

Créez un projet Angular et lancez le. Regardez la structure des fichiers et répertoires.



❑ Classe composant

- Angular organise l'interface utilisateur en composants réutilisables.
- Chaque composant encapsule du code, des modèles et des styles spécifiques à une fonctionnalité de l'application. Les composants communiquent entre eux via des entrées et des sorties.
- Un composant est défini par une classe en utilisant le mot-clé **class** en TypeScript
- Un composant est composé généralement:
 - La vue appelée **template** : c'est la partie HTML du composant,
 - La classe : c'est la classe Typescript comportant la logique du composant,
 - Le CSS : c'est un fichier de style du composant,
 - Le fichier **spec.ts** : c'est un fichier où les tests unitaires sont écrits
- Pour créer un composant on utilise la commande ci-dessous:

```
ng generate component component_name  
// avec alias: ng g c component_name  
// example: ng generate component layout/header
```



Lors de la création avec la CLI, celle-ci va automatiquement ajouter le composant dans le module principal

❑ Classe composant

→ Un composant créé a le code par défaut suivant:

```
@Component({
  selector: 'app-header',
  templateUrl: './header.component.html',
  styleUrls: ['./header.component.scss']
})
export class HeaderComponent {
}
```

→ Pour des simples vues (template) on peut opter de mettre le code html dans la classe

→ Pour utiliser ce composant dans un autre composant (parent) on utilise le **selector** (sélecteur) de la classe. Ce sélecteur doit être unique dans l'application.

```
@Component({
  selector: 'app-header',
  template: `
    <header>
      | Hello Header!
    </header>
  `,
  styleUrls: ['./header.component.scss']
})
export class HeaderComponent {
}
```

```
2
3   <app-header></app-header>
4
```

Exercice:

Créez un composant **header** dans un dossier **layout**, mettez un message "**Hello Word!**" sur sa vue suivant les 2 façons donc nous avons vu.



❑ Cycle de vie d'un composant

En Angular, les composants ont un cycle de vie. Cela signifie qu'il y a un moment où ils sont créés, mis à jour, détruits.

Tout cela, est géré par le framework. Il existe des méthodes à utiliser durant les différents moments de vie des composants: ce sont des hooks.

↪ **ngOnChanges**

Premier hook exécuté, il est appelé à chaque fois qu'un ou plusieurs @Input() changent.

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

@Component({
  selector: 'app-header',
  templateUrl: './header.component.html',
  styleUrls: ['./header.component.scss']
})
export class HeaderComponent implements OnChanges {
  @Input() username!: string;

  ngOnChanges(changes: SimpleChanges) {
    if ('username' in changes) {
      console.log(changes['username'].currentValue)
    }
  }
}
```

❑ Cycle de vie d'un composant

↪ **ngOnInit()**

Appelée une seule fois, à la création du composant.

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

@Component({
  selector: 'app-items-list',
  templateUrl: './items-list.component.html',
  styleUrls: ['./items-list.component.scss']
})
export class ItemsListComponent implements OnInit {

  ngOnInit() {
    // mettre vos appels backend ici
  }
}
```

↪ **ngAfterViewInit()**

Appelée après que la vue d'un composant est initialisée.

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

@Component({
  selector: 'app-items-list',
  templateUrl: './items-list.component.html',
  styleUrls: ['./items-list.component.scss']
})
export class ItemsListComponent implements AfterViewInit {

  ngAfterViewInit() {
    // Ajout du code afin de modifier les éléments de la vue
    // périmètre du composant
  }
}
```



❑ Cycle de vie d'un composant

↪ **ngOnDestroy()**

Elle est appelée juste avant la destruction d'un composant.

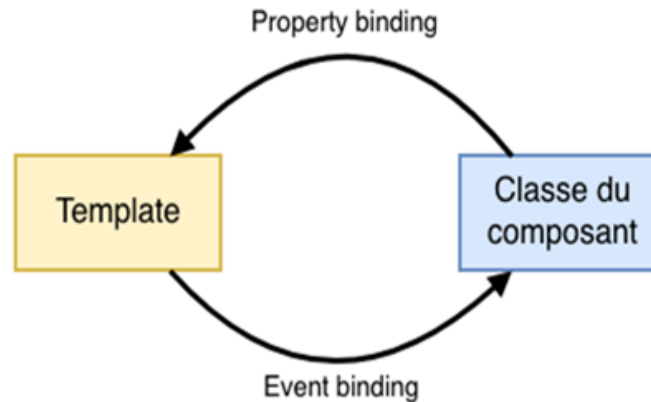
```
import { Component, OnDestroy } from '@angular/core';

@Component({
  selector: 'app-items-list',
  templateUrl: './items-list.component.html',
  styleUrls: ['./items-list.component.scss']
})
export class ItemsListComponent implements OnDestroy {

  ngOnDestroy () {
    // ici on peut:
    // se désabonner des observables, supprimer les écouteurs d'événements
  }
}
```

❑ Interpolation et data binding

Pour rendre dynamique l'affichage de la vue d'un composant, Angular permet d'implémenter des interactions entre un template et la classe d'un composant.



↪ Interpolation

Elle permet d'afficher une valeur dans le template (via l'utilisation des markups `{{ }}`).

```
@Component({
  selector: 'app-example',
  template: `
    <header>
      <div> Bonjour {{ username }}! </div>
    </header>
  `,
  styleUrls: ['./example.component.scss']
})
export class ExampleComponent {
  username: string = "Joe";
}
```



❑ Interpolation et data binding

↪ Property binding

Il permet de lier une propriété d'un élément HTML à une valeur du composant (via l'utilisation des []).

```
@Component({
  selector: 'app-example',
  template: `
    <div>
      <button [disabled]="isDisabled">Valider</button>
    </div>
  `,
  styleUrls: ['./example.component.scss']
})
export class ExampleComponent {
  isDisabled: boolean = true;
}
```

↪ Event binding

Il permet de lier un événement d'un élément HTML à une méthode du composant.

```
@Component({
  selector: 'app-example',
  template: `
    <div>
      <button (click)="onClick()">Valider</button>
    </div>
  `,
  styleUrls: ['./example.component.scss']
})
export class ExampleComponent {

  onClick() {
    console.log("Bouton cliqué!");
  }
}
```




❑ Interpolation et data binding

↪ Two-way binding

Il permet de créer une connexion bidirectionnelle entre l'élément HTML et un composant. On utilise la syntaxe [()].

```
@Component({
  selector: 'app-example',
  template: `
    <div>
      <input type="text" [(ngModel)]="name" />
      <p>Mon prénom est {{ name }}</p>
    </div>
  `,
  styleUrls: ['./example.component.scss']
})
export class ExampleComponent {

  name = "Joe";
}
```

Exercice:

Créez un composant **example** dans le dossier **components**, mettez un message **"Hello {{name}}!"** sur la vue. Ajoutez un élément HTML input qui vous permettra de changer la valeur de name. Par défaut name a une valeur **"world"**.

PS. Si vous avez une erreur, importez **FormsModule** du @angular/forms dans le AppModule.



❏ Directives

Ce sont des classes qui ajoutent un comportement supplémentaire aux éléments du DOM.

↳ Les directives structurelles

→ **ngIf/else**: permet de conditionner un élément du DOM

```
<div>
  <span *ngIf="userLoggedIn else offline">
    Welcome, {{ username }}
  </span>
  <ng-template #offline>
    <span>Non connecté</span>
  </ng-template>
</div>
```

→ **ngFor**: permet de boucler sur une collection d'éléments afin de générer dynamiquement du contenu HTML.

```
@Component({
  selector: 'app-example',
  template: `
    <ul>
      <li *ngFor="let item of items">{{ item }}</li>
    </ul>
  `,
  styleUrls: ['./example.component.scss']
})
export class ExampleComponent {
  items: string[] = ["Citron", "Orange", "Cérise"];
}
```



❏ Directives

→ **ngSwitch**: permet d'évaluer une expression et d'effectuer un rendu conditionnel

```
@Component({
  selector: 'app-example',
  template: `
    <div [ngSwitch]="color">
      <p *ngSwitchCase="'red'">Red color</p>
      <p *ngSwitchCase="'blue'">Blue color</p>
      <p *ngSwitchCase="'green'">Green color</p>
      <p *ngSwitchDefault>Unknown color</p>
    </div>
  `,
  styleUrls: ['./example.component.scss']
})
export class ExampleComponent {
  color: string = "red";
}
```

Exercice:

Créer un composant dans le dossier **components**, vous allez créer un tableau de 5 fruits (ayant orange) dans votre composant et les afficher avec un **ngFor** dans votre vue. Si on est sur le fruit "**orange**" alors affichez le en gras (****).

□ Directives

↳ Les directives d'attributs

→ **ngClass**: permet d'ajouter des classes css grâce aux property binding.

```
@Component({
  selector: 'app-example',
  template: `
    <p [ngClass]="{'text-error': isError}">Une erreur est survenue</p>
  `,
  styleUrls: ['./example.component.scss']
})
export class ExampleComponent {
  isError: boolean = true;
}
```

→ **ngStyle**: permet d'ajouter du style sous forme de valeur ou sous forme d'objet de style.

```
@Component({
  selector: 'app-example',
  template: `
    <p [ngStyle]="{'color': color}">Une erreur est survenue</p>
  `,
  styleUrls: ['./example.component.scss']
})
export class ExampleComponent {
  color: string = 'red';
}
```



❏ Directives

↔ Création d'une directive

Pour créer une directive, on utilise la commande suivante:

```
ng generate directive directive_name  
// avec alias: ng g d directive_name  
// example: ng generate directive directives/directive_name
```



```
import { Directive } from '@angular/core';  
  
@Directive({  
  selector: '[appBold]'  
})  
export class BoldDirective {  
  
  constructor() { }  
  
}
```

Exemple de contenu d'une directive et utilisation dans une vue

```
import { Directive, ElementRef, Renderer2, AfterViewInit } from '@angular/core';  
  
@Directive({  
  selector: '[appBold]'  
})  
export class BoldDirective implements AfterViewInit{  
  
  constructor(private el: ElementRef, private renderer: Renderer2) { }  
  
  ngAfterViewInit() {  
    this.el.nativeElement.style.fontWeight = 'bold';  
  }  
}
```

```
<p appBold >Hello world!</p>
```



❑ Variable de template

Les variables locales et les variables de template sont des moyens de référencer des éléments du DOM ou des valeurs spécifiques dans les templates

↪ **Variable locale:** référencement avec le symbole # suivi du nom

```
<div>
  <input #myInput type="text">
  <button (click)="logValue(myInput.value)">Log Value</button>
</div>
```

↪ **Variable de template:** stockage des valeurs spécifique afin de les réutiliser dans le contexte d'exécution

```
<div *ngFor="let item of items; let i = index">
  {{ i }}: {{ item }}
</div>
```



❑ Les composants enfants

Dans Angular, un composant peut contenir des sous composants sous une hiérarchie de parents-enfants.

```
@Component({
  selector: 'app-items-list',
  template: `
    <div>
      <p>Liste des éléments</p>
      <ul>
        <li>Un</li>
        <li>Deux</li>
      </ul>
      <app-example></app-example>
    </div>
  `,
  styleUrls: ['./items-list.component.scss']
})
export class ItemsListComponent {
}
```

❖ La projection de contenu (Content projection)

C'est un mécanisme permettant d'insérer du contenu dynamique dans un composant depuis l'extérieur. on en distingue 2 types de projection de contenu.

❑ Les composants enfants

- ↪ **La projection avec contenu brut:** on insère du contenu à l'intérieur du composant. on utilise la balise `<ng-content>` pour spécifier ou peut être placer le composant externe.

```
@Component({
  selector: 'app-items-list',
  template: `
    <div>
      <ng-content></ng-content>
    </div>
  `,
  styleUrls: ['./items-list.component.scss']
})
export class ItemsListComponent {
}
```

```
<!-- app.component.html -->
<app-items-list>
  <div>
    <p>Liste des éléments</p>
    <ul>
      <li>Un</li>
      <li>Deux</li>
    </ul>
  </div>
</app-items-list>
```

- ↪ **La projection avec des sélecteurs:** on insère du contenu à l'intérieur du composant en fonction des secteurs défini.

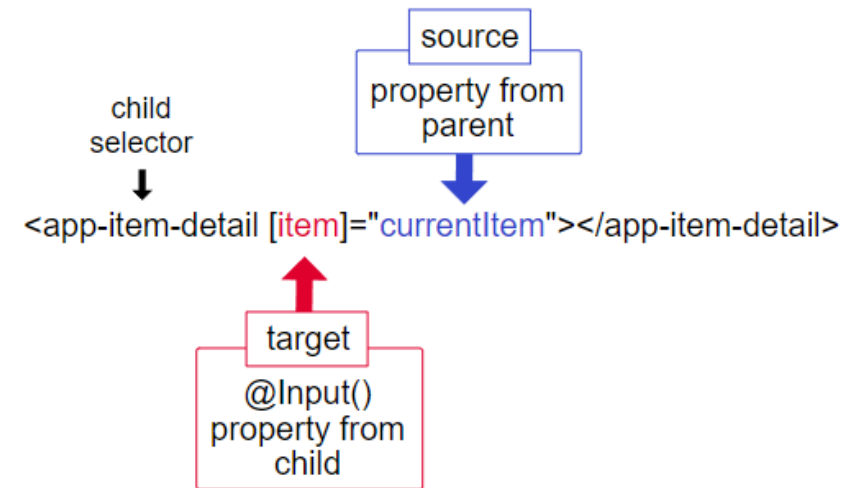
```
@Component({
  selector: 'app-items-list',
  template: `
    <div>
      <ng-content select="[list]"></ng-content>
      <ng-content select="[other]"></ng-content>
    </div>
  `,
  styleUrls: ['./items-list.component.scss']
})
export class ItemsListComponent {
}
```

```
<app-items-list>
  <div list>
    <p>Liste des éléments</p>
    <ul>
      <li>Un</li>
      <li>Deux</li>
    </ul>
  </div>
  <app-example other></app-example>
</app-items-list>
```


❑ Les composants enfants

❖ Communication parent-enfant

- Le composant parent peut injecter un ou plusieurs paramètres dans le composant enfant en utilisant un property binding.
- Le composant enfant doit déclarer des propriétés exposées comme paramètres d'entrée (avec **@Input()**).



Parent

Enfant

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

@Component({
  selector: 'app-root',
  template: `
    <div>
      <p>Liste des éléments</p>
      <app-items-list [items]="items"></app-items-list>
    </div>
  `,
  styleUrls: ['./app.component.scss']
})
export class AppComponent {
  items: string[] = ["Cérise", "Kiwi", "Orange"]
}
```

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

@Component({
  selector: 'app-items-list',
  template: `
    <div>
      <ul>
        <li *ngFor="let item of items" > {{ item }}</li>
      </ul>
    </div>
  `,
  styleUrls: ['./items-list.component.scss']
})
export class ItemsListComponent {
  @Input() items:string[] = [];
}
```

❑ Les composants enfants

❖ Communication enfant-parent

- Le composant enfant communique avec le parent par des événements (**@Output()**)
- Output() est utilisé avec **EventEmitter** pour émettre des événements personnalisés

Enfant



Parent

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

@Component({
  selector: 'app-form-item',
  template: `
    <div>
      <input #input type="text" />
      <button (click)="onAdd(input.value)"> Ajouter</button>
    </div>
  `,
  styleUrls: ['./form-item.component.scss']
})
export class FormItemComponent {
  @Output() addItemClick = new EventEmitter();

  onAdd(item: string) {
    this.addItemClick.emit(item);
  }
}
```

```
@Component({
  selector: 'app-root',
  template: `
    <div>
      <p>Liste des éléments</p>
      <app-form-item (addItemClick)="onAddItemClick($event)"></app-form-item>
      <app-items-list [items]="items"></app-items-list>
    </div>
  `,
  styleUrls: ['./app.component.scss']
})
export class AppComponent {
  items: string[] = ["Cérise", "Kiwi", "Orange"]

  onAddItemClick(newItem: string) {
    this.items.push(newItem);
  }
}
```



❑ UI: Comment ajouter bootstrap

→ C'est une boîte à outils puissantes et riche en fonctionnalités permettant de réaliser des interfaces graphiques.

→ Installation de bootstrap

```
npm install bootstrap bootstrap-icons
```

→ Configuration de bootstrap

- Option 1: Configuration du **angular.json**

```
"styles": [  
  "node_modules/bootstrap/dist/css/bootstrap.min.css",  
  "src/styles.css"  
],
```

- Option 2: Import dans le fichier **src/style.css**

```
/* You can add global styles to this file, and also import other style files */  
@import '~bootstrap/dist/css/bootstrap.min.css';
```

→ Documentation

<https://getbootstrap.com/docs/5.3/getting-started/introduction/>



Atelier

Activité



— TD 2: Création des composants et communication

- TD 2.1, Vous allez structurer une application existante en composant et créer de nouveaux composants pour afficher une liste de données.
- TD 2.2, Vous allez créer une application qui permet d'augmenter et de baisser la valeur d'un nombre.

4. Angular: Gestion de l'environnement

- ❑ Formulaires: Template Driven Form (TDF) et Data Driven Form (DDF)
- ❑ Validation des formulaires
- ❑ Les observables
- ❑ Les services
- ❑ Le router
- ❑ Le Guard
- ❑ Interceptor
- ❑ Les Pipes

📄 Formulaires

Il y'a deux méthodes pour manipuler les formulaires et chacune d'elles sont incompatibles entre elles.

➡ Template driven forms (TDF)

C'est un formulaire entièrement géré par le template. On utilise les directives **ngModel**, **ngSubmit**, **ngForm** depuis la vue afin de faire la liaison entre les champs du formulaire et une propriété.

```
@Component({
  selector: 'app-login',
  template: `
    <form (ngSubmit)="onSubmit()">
      <div>
        <input type="text" [(ngModel)]="credentials.email" name="email">
      </div>
      <div>
        <input type="password" [(ngModel)]="credentials.password" name="password">
      </div>
      <div>
        <button type="submit">Se connecter</button>
      </div>
    </form>
  `,
  styleUrls: ['./login.component.scss']
})
export class LoginComponent {
  credentials = {
    email: '',
    password: ''
  }

  onSubmit() {
    console.log(this.credentials);
  }
}
```

🚨 On peut rajouter une variable locale **#form** dans le formulaire (<form>)

🚨 Ne pas oublier d'importer le **Forms Module** depuis **@angular/forms**

Formulaires

➔ Reactive forms (DDF)

Un **reactive forms** permet de créer un objet qui représente le formulaire et ensuite cet objet sera lié à la balise **<form>** du template.

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

@Component({
  selector: 'app-login',
  template: `
    <form [formGroup]="form" (ngSubmit)="onSubmit()">
      <div>
        <input type="text" formControlName="email">
      </div>
      <div>
        <input type="password" formControlName="password">
      </div>
      <div>
        <button type="submit">Se connecter</button>
      </div>
    </form>
  `,
  styleUrls: ['./login.component.scss']
})
export class LoginComponent {
  form = new FormGroup(
    {
      email: new FormControl(""),
      password: new FormControl("")
    }
  );

  onSubmit() {
    console.log(this.form.value);
  }
}
```

🚨 Le formulaire est créé avec **Formgroup** qui le représente et ses sous éléments (champs) sont fait avec le **FormControl**. Sur la vue la liaison est faite avec **formControlName**.

🚨 Ne pas oublier d'importer le **ReactiveForms Module** depuis **@angular/forms**

❑ Validation des formulaires

- ➔ **Template driven forms (TDF)**
- ➔ Validation faite dans template
- ➔ création des variables locale liées aux directives (**ngForm**, **ngModel**)
- ➔ Personnalisation des erreurs

Exercice:

Créez un composant dans lequel vous allez ajouter un formulaire (**TDF**) ayant les champs suivants: **firstname** (obligatoire), **lastname** (obligatoire), **email** (obligatoire). Si lors de la soumission il y'a des erreurs alors les afficher sous les champs concernés sinon afficher un message alert.

```
import { Component, ViewChild } from '@angular/core';
import { NgForm } from '@angular/forms';

@Component({
  selector: 'app-login2',
  template: `
    <form #form="ngForm" (ngSubmit)="onSubmit(form.value)">
      <div>
        <input type="text" [(ngModel)]="credentials.email" required name="email" #emailRef="ngModel">
        <div *ngIf="emailRef.invalid && emailRef.touched">
          <div *ngIf="emailRef.errors?.['required']">Email obligatoire </div>
        </div>
      </div>
      <!-- ..... -->
    </form>
  `,
  styleUrls: ['./login2.component.scss']
})
export class Login2Component {
  @ViewChild('form') form!: NgForm;
  credentials = {
    email: '',
    password: ''
  }

  onSubmit(e: {}) {
    if (this.form.invalid) {
      return;
    }
    console.log(e);
  }
}
```


❑ Validation des formulaires

➔ Reactive forms (DDF)

- ➔ Ajout des validateurs (Validators)
- ➔ Facilité de maintenance

Exercice:

Créer un composant dans lequel vous allez ajouter un formulaire (DDF) ayant les champs suivants: firstname (obligatoire), lastname (obligatoire), email (obligatoire, email format). Si lors de la soumission il y'a des erreurs alors les afficher sous les champs concernés sinon afficher un message alert.

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

@Component({
  selector: 'app-login',
  template: `
    <form [formGroup]="form" (ngSubmit)="onSubmit()">
      <div>
        <input type="text" formControlName="email" >
        <div *ngIf="email?.invalid && (email?.dirty || email?.touched)">
          <div *ngIf="email?.errors?.['required']">Email obligatoire </div>
          <div *ngIf="email?.errors?.['email']">Mauvais format email </div>
        </div>
      </div>
      <!-- ..... -->
      <div>
        <button type="submit">Se connecter</button>
      </div>
    </form>
  `,
  styleUrls: ['./login.component.scss']
})
export class LoginComponent {
  form = new FormGroup({
    {
      email: new FormControl("", [Validators.required, Validators.email]),
      password: new FormControl("", Validators.required)
    }
  });

  get email() { return this.form.get('email'); }

  onSubmit() {
    if (this.form.invalid) {
      return;
    }
    console.log(this.form.value);
  }
}
```

❑ Les observables

- Un Observable est un flux d'évènements sur lequel on peut s'abonner pour réagir à chaque fois qu'un évènement se déclenche.
- Le flux des Observable sont, par défaut, fermés. Cela signifie que si on ne s'abonne pas à un Observable, on ne recevra pas les évènements.
- C'est un concept central dans la programmation réactive et notamment utilisé dans les bibliothèques comme **RxJS**.

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

@Component({
  selector: 'app-example',
  templateUrl: './example.component.html',
  styleUrls: ['./example.component.scss']
})
export class ExampleComponent {
  // Création
  monObservable: Observable<string> = new Observable(observer => {
    observer.next('Salut !'); // émission d'une valeur
    observer.complete(); // émission terminée
  });

  // Abonnement
  monAbonnement = this.monObservable.subscribe((message) => console.log(`messaga: ${message}`));
}
```

- Dans angular, la responsabilité des observables va généralement être dans un service et depuis les composants vous allez vous abonner pour recevoir des données.

<https://rxjs.dev/guide/overview>



Les services

- C'est un objet dont l'instanciation est gérée par Angular.
- Cet objet est unique dans toute l'application et il peut être injecté dans n'importe lequel de vos composants ou bien encore dans un autre service.
- C'est une classe (**class**) décorée par **@Injectable()**
- Il permet:
 - ◆ Partager des fonctionnalités, propriétés entre plusieurs composants
 - ◆ Communiquer entre les composants
 - ◆ Développer des fonctionnalités techniques de l'application

```
ng generate service service_name  
// avec alias: ng g s service_name  
// example: ng generate service services/service_name
```

```
import { Injectable } from '@angular/core';  
  
@Injectable({  
  providedIn: 'root'  
})  
export class ExampleService {  
  
  constructor() { }  
}
```

❏ Les services

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

@Injectable({providedIn: 'root'})
export class ExampleService {
  constructor(private http: HttpClient) { }

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

Depuis un composant, voici un exemple d'utilisation

```
import { Component, OnInit } from '@angular/core';
import { ExampleService } from '../services/example.service';
@Component({
  selector: 'app-example',
  templateUrl: './example.component.html',
  styleUrls: ['./example.component.scss']
})
export class ExampleComponent implements OnInit {
  constructor(private exampleService: ExampleService) { }

  ngOnInit(): void {
    this.exampleService.getExample().subscribe((data) => {
      // traitement
    });
  }
}
```

🚨 Ne pas oublier d'importer le module **HttpClientModule** venant de **@angular/common/http** dans la section **imports** du module principale de l'application.

🚨 HttpClient est un utilitaire qui permet de faire des requêtes Http en GET, POST, DELETE, PUT.

🚨 <https://angular.io/api/common/http/HttpClient#usage-notes>

❑ Le router

En Angular, c'est ce qui permet de relier une url à un composant

Lors de la création d'un projet angular vous avez la possibilité de le rajouter, cela générera un fichier app-routing.modules (router) qui informera le module principal des routes de l'application.

```
const routes: Routes = [
  { path: '', component: HomeComponent},
  { path: 'items', component: ItemsListComponent},
  { path: '**', component: NotFoundComponent }
  // si vous n'avez de page not-found alors vous pouvez forcer
  // les redirections vers la page par défaut
  // { path: '**', redirectTo: '' }
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

Pour indiquer l'emplacement d'insertion du composant, il faut utiliser la directive **<router-outlet>** directement dans le "root component" (par exemple **AppComponent**), cela peut aider si on souhaite mettre en place un menu et générer des liens vers les pages.

```
@Component({
  selector: 'app-root',
  template : `
    <div>
      <ul>
        <li>
          <a routerLink="/">Accueil</a>
        </li>
        <li>
          <a routerLink="/items">Liste des Items</a>
        </li>
      </ul>
      <router-outlet></router-outlet>
    </div>
  `,
  styleUrls: ['./app.component.scss']
})
export class AppComponent {
}
```

Le router

Pour définir un paramètre d'une route et le récupérer on agit ainsi:

```
const routes: Routes = [
  { path: '', component: HomeComponent },
  { path: 'items', component: ItemsListComponent },
  { path: 'items/:id', component: ItemComponent },
  { path: '**', component: NotFoundComponent }
  // si vous n'avez de page not-found alors vous pouvez forcer
  // les redirections vers la page par défaut
  // { path: '**', redirectTo: '' }
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

Pour récupérer le paramètre d'une url on procède de la façon suivante:

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

@Component({
  selector: 'app-item',
  templateUrl: './item.component.html',
  styleUrls: ['./item.component.scss']
})
export class ItemComponent implements OnInit {
  constructor(private route: ActivatedRoute) {}

  ngOnInit() {
    const id = this.route.snapshot.paramMap.get('id');
    console.log('id', id)
  }
}
```

Pour générer une route avec paramètre depuis une vue

```
<a [routerLink]="['/items', id]"></a>
```



❑ Le router

→ Pour effectuer une redirection depuis un composant:

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

@Component({
  selector: 'app-item',
  templateUrl: './item.component.html',
  styleUrls: ['./item.component.scss']
})
export class ItemComponent implements OnInit {
  constructor(private route: ActivatedRoute, private router: Router) {}

  ngOnInit() {
    const id = this.route.snapshot.paramMap.get('id');
    if (!id) {
      this.router.navigate(['']) // redirection vers la page d'accueil ayant l'url ''
    }
  }
}
```

→ Pour une bonne structure en sous routes on peut écrire nos routes ainsi

```
const routes: Routes = [
  { path: '', component: HomeComponent },
  {
    path: 'items',
    children: [
      { path: '', component: ItemsListComponent },
      { path: ':id', component: ItemComponent }
    ]
  },
  { path: '**', component: NotFoundComponent }
  // si vous n'avez de page not-found alors vous pouvez forcer
  // les redirections vers la page par défaut
  // { path: '**', redirectTo: '' }
];
```



❑ Les pipes

Un pipe dans Angular est un moyen simple de transformer, formater ou filtrer une valeur directement dans un template

Lorsque vous utilisez un pipe dans un template, vous le faites suivre d'une barre verticale (|) et du nom du pipe. Il existe des pipes natifs à angular: DatePipe, UpperCasePipe, LowerCasePipe.

```
<p>{{ 'hello world' | uppercase }}</p>
```

```
ng generate pipe pipe_name
// avec alias: ng g p pipe_name
// example: ng generate pipe pipes/adult
```

```
@Pipe({
  name: 'adult'
})
export class AdultPipe implements PipeTransform {
  transform(value: unknown, ...args: unknown[]): unknown {
    return null;
  }
}
```

Exemple de pipe :

```
@Pipe({
  name: 'adult'
})
export class AdultPipe implements PipeTransform {
  transform(value: number, ...args: unknown[]): unknown {
    return value >= 18 ? 'Adult' : 'Minor';
  }
}
```

```
<p> {{ 21 | adult }} </p>
```




❑ Le Guard

Un Guard permet de contrôler l'accès à des routes spécifiques dans votre application.

```
ng generate guard guard_name
// avec alias: ng g g guard_name
// exemple: ng generate guard guards/auth
```

```
import { CanActivateFn } from '@angular/router';

export const authGuard: CanActivateFn = (route, state) => {
  return true;
};
```

Cas d'utilisation:

```
import { AuthGuard } from '../guard/auth.guard';

const routes: Routes = [
  { path: '', component: HomeComponent },
  {
    path: 'items',
    canActivate: [AuthGuard],
    children: [
      { path: '', component: ItemsListComponent },
      { path: ':id', component: ItemComponent }
    ]
  },
  { path: '**', component: NotFoundComponent }
];
// si vous n'avez de page not-found alors vous pouvez forcer
// les redirections vers la page par défaut
// { path: '**', redirectTo: '' }
```



En créant un guard avec la CLI, vous avez le choix sur quel type de guard vous souhaitez créer

```
? Which type of guard would you like to create? (Press <space> to select, <a> to toggle all,
<i> to invert selection, and <enter> to proceed)
>(*) CanActivate
  ( ) CanActivateChild
  ( ) CanDeactivate
  ( ) CanMatch
```



Le Guard

➡ CanActivate

Permet de vérifier si on peut accéder à une route

➡ CanDeactivate

Permet de vérifier si on peut quitter une route

➡ CanMatch

Permet de vérifier si une route peut être activée tout comme CanActivate, mais si ce n'est pas le cas alors le router va essayer de matcher une autre route avec le même path

➡ CanActivateChildGuard

Permet de vérifier si on peut accéder à une route enfant

```
export const AuthGuard: CanActivateFn = (route, state) => {  
  const router = inject(Router);  
  const isAuthenticated = true // ou false;  
  
  if (isAuthenticated) {  
    return true;  
  }  
  
  return router.parseUrl("/login");  
};
```

Exercice:

Créer un nouveau projet angular avec routing. vous allez créer 3 composants (Home, Register, Login) et créer des routes qui pointent sur ces composants via un menu.



Interceptor

- Un interceptor permet de traiter les requêtes et les réponses HTTP avant qu'elles ne soient envoyées ou reçues par le serveur.
- Il peut être utilisé pour ajouter, modifier ou supprimer des données dans les entêtes de la requête ou de la réponse.



```
ng generate interceptor interceptor_name  
// example: ng generate interceptor interceptors/auth
```



```
import { Injectable } from '@angular/core';  
import { HttpInterceptor, HttpRequest, HttpHandler, HttpEvent } from '@angular/common/http';  
import { Observable } from 'rxjs';  
  
@Injectable()  
export class AuthInterceptor implements HttpInterceptor {  
  
  intercept(request: HttpRequest<unknown>, next: HttpHandler): Observable<HttpEvent<unknown>> {  
    // Récupération du token d'authentification  
    const token = 'token-value';  
  
    // Ajout du token dans les entêtes de la requête  
    const authReq = request.clone({  
      setHeaders: {  
        Authorization: `Bearer ${token}`  
      }  
    });  
  
    // Envoi de la requête avec les nouvelles entêtes  
    return next.handle(authReq);  
  }  
}
```



```
// .....  
  
@NgModule({  
  declarations: [  
    // .....  
  ],  
  imports: [  
    // .....  
  ],  
  providers: [  
    {  
      provide: HTTP_INTERCEPTORS,  
      useClass: AuthInterceptor,  
      multi: true  
    }  
  ],  
  bootstrap: [  
    // .....  
  ]  
})  
export class AppModule { }
```



Atelier

Activité



– TD 3: Création d'une application web

- TD 3, Vous allez créer une application qui va manipuler les données d'une API (Jsonplaceholder).

<https://jsonplaceholder.typicode.com/>

Liens utiles

<https://apprendre.bonjour-angular.com/cest-quoi/angular/>

<https://angular.fr/>

https://angular.fr/services-et-http/observable.html#_2-observer-qui-ecoute-l-observable

<https://kinsta.com/knowledgebase/what-is-typescript/>