

Introduction à Angular (8 heures)

Professeur Titulaire : Millimono Sory (AI Seacher)

Plan Du Cours



○ **Introduction à Angular, pourquoi l'utiliser.**

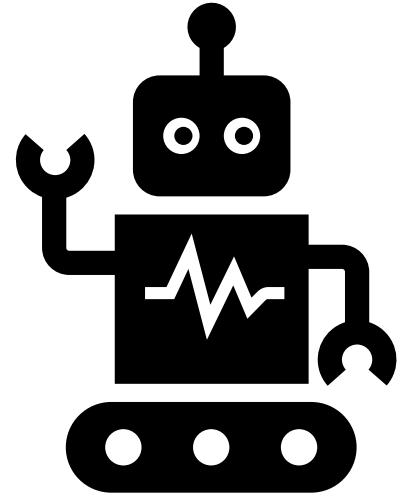
○ **Installation et configuration de l'environnement de développement.**

○ **Découverte de l'architecture Angular : Modules, composants, templates.**

○ **Premiers pas : Type Script.**

○ **Premiers pas : Création d'une application simple.**

Historique et philosophie de Angular



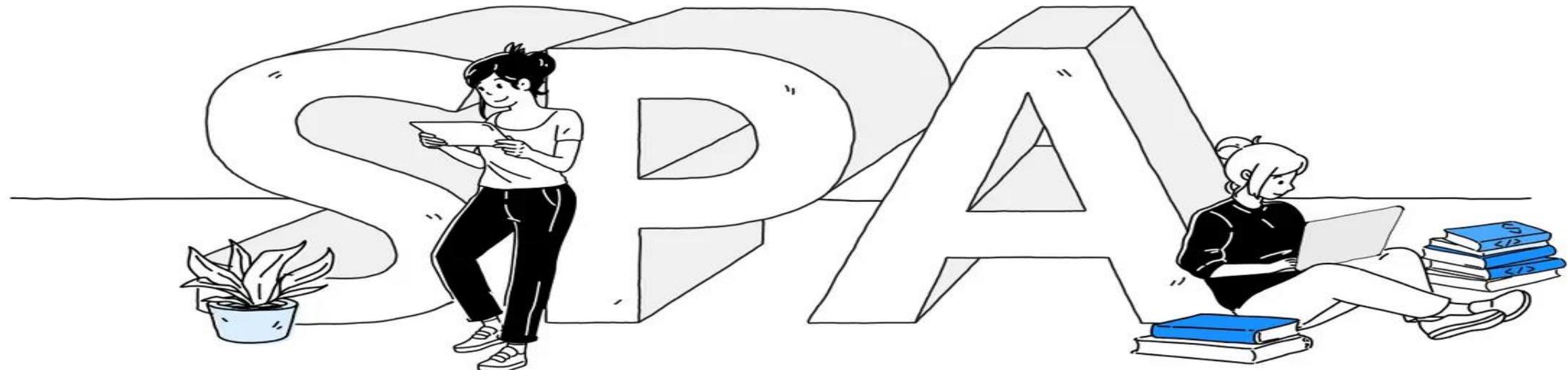
Historique d'Angular :

Angular, initialement appelé AngularJS, a été lancé en 2010 par Miško Hevery chez Google. AngularJS utilisait principalement JavaScript et visait à simplifier le développement et les tests d'applications à une seule page en proposant un cadre structuré pour le code côté client.

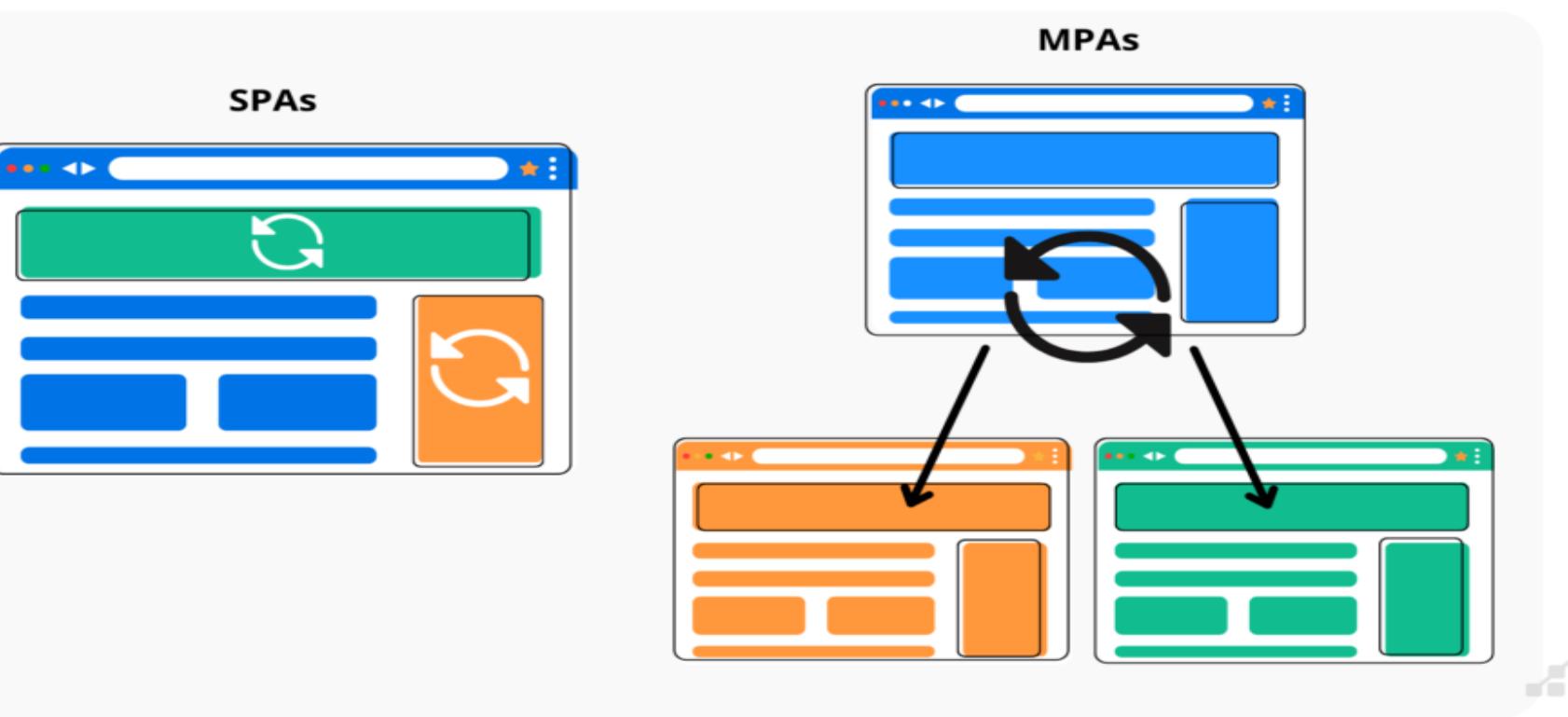
En 2016, Angular 2 a été publié, marquant une refonte complète du framework, avec une transition de JavaScript à TypeScript, qui est une surcouche de JavaScript offrant des fonctionnalités typées et orientées objet. Depuis lors, Angular a continué d'évoluer avec des mises à jour régulières, améliorant ses performances, ses outils et ses fonctionnalités.

Angular est un framework de développement front-end pour la création d'applications web dynamiques. Développé et maintenu par Google, il a été lancé initialement sous le nom AngularJS en 2010, puis complètement réécrit en 2016 sous le simple nom d'Angular.

Angular est particulièrement adapté pour développer des applications web à grande échelle et des SPA (Single Page Applications) qui nécessitent une architecture robuste, une maintenance facile, et des fonctionnalités interactives avancées.



Single-page Applications VS Multiple-page Applications



Avant les frameworks d'applications monopages (SPA), le web se composait majoritairement d'applications multipages (MPA) en HTML statique, utilisant des technologies serveur comme PHP ou Java. L'ajout d'AJAX a permis des mises à jour de pages sans rechargement, préfigurant les SPA. jQuery a innové en UI mais sans gestion avancée des données, problème que Knockout.js a résolu avec la liaison de données MVVM, séparant les vues des données.

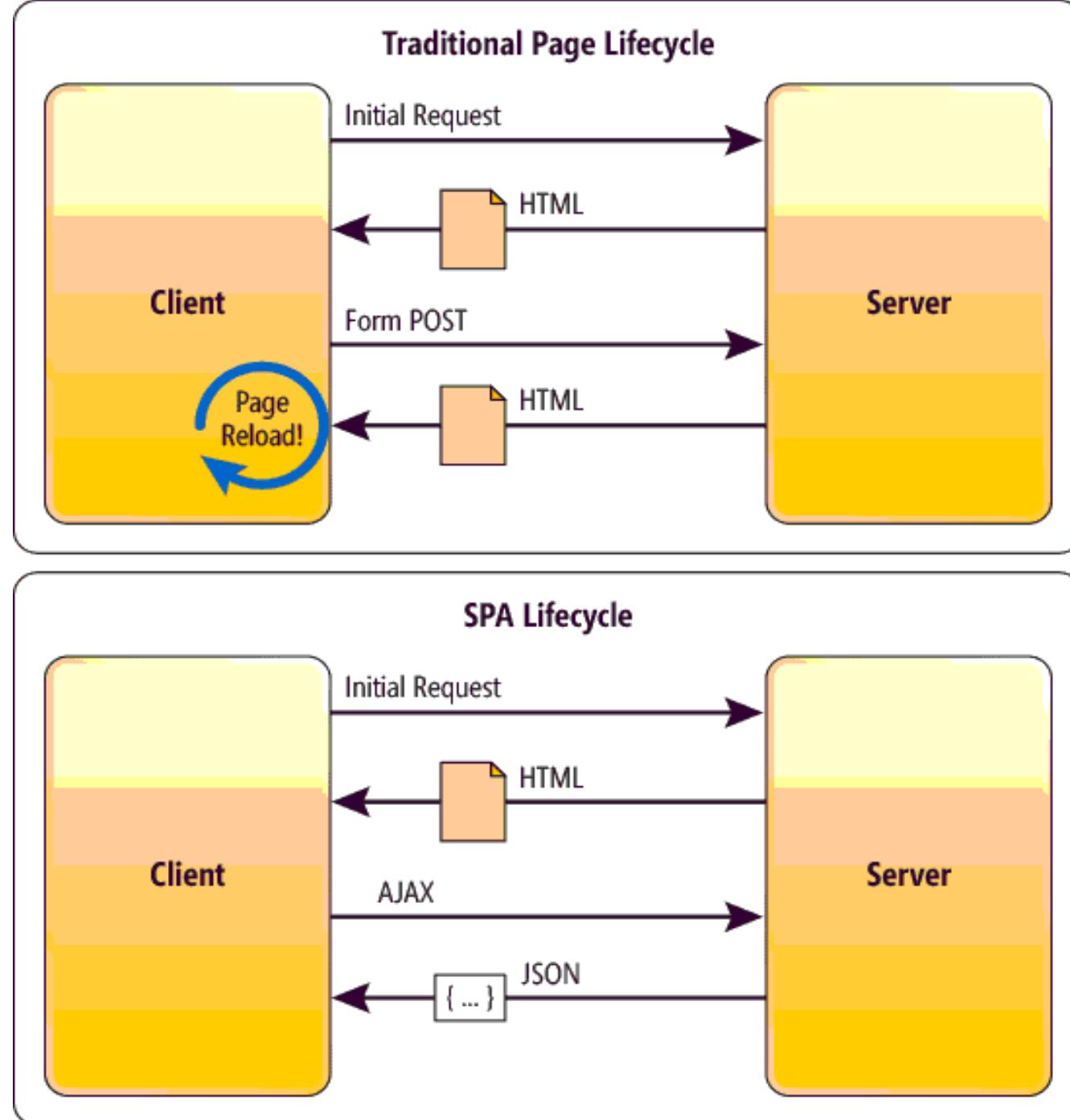
En 2009, Backbone.js offrait un framework client léger pour les SPA, exigeant cependant effort et code répétitif. Angular.js a révolutionné en 2010, unifiant MVC, liaison de données bidirectionnelle, modèles et injection de dépendances en une solution SPA complète.

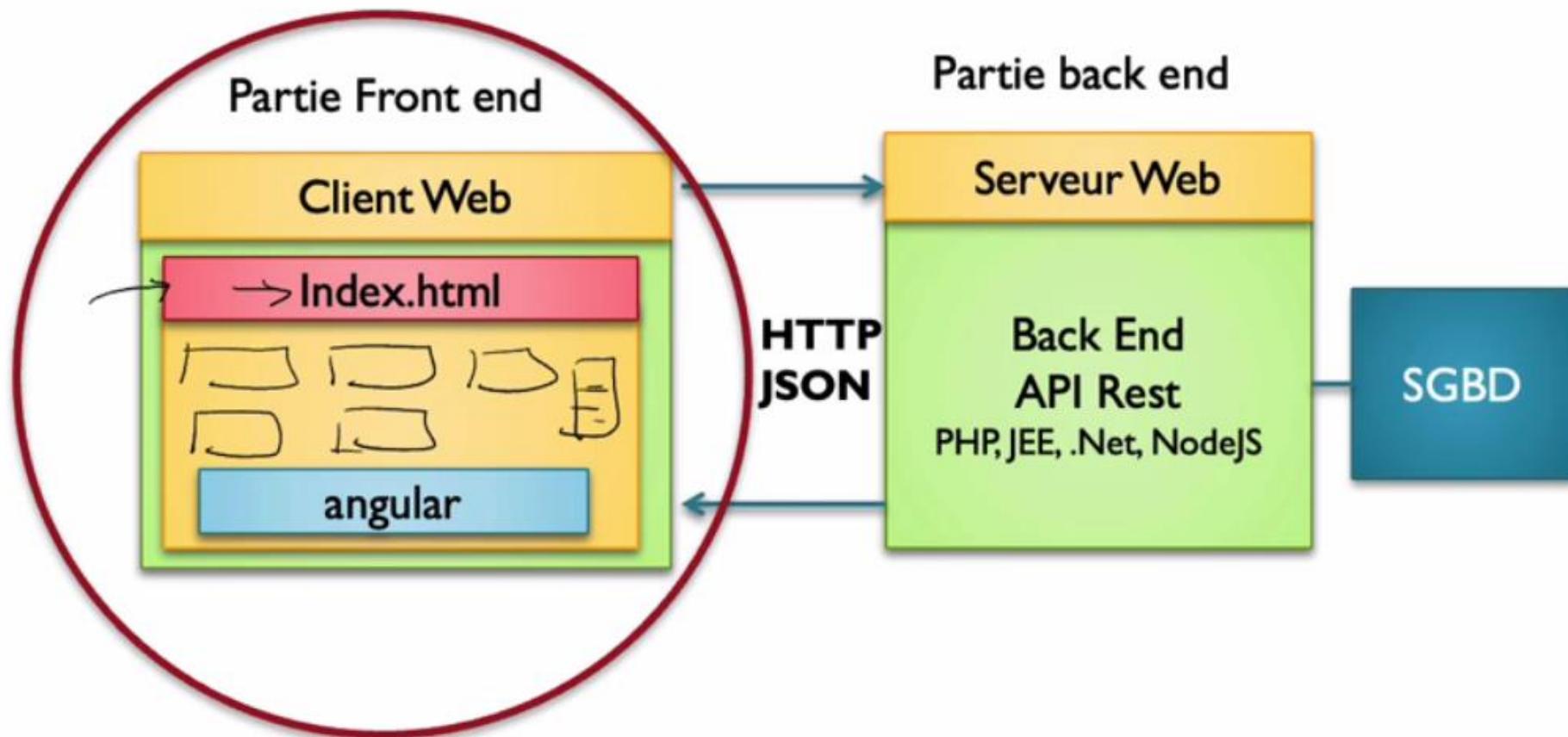
Les applications multipages dominent encore largement, souvent préférées pour leur efficacité éprouvée dans de nombreux cas d'utilisation, comme Stack Overflow et Amazon.

Single-page Applications VS Multiple-page Applications

	SPAs	MPAs
PERFORMANCE	Faster loading time	Slower loading time
DEBUGGING	More difficult	Well supported by debugging tools
DEVELOPMENT	Fast	Slower, more complex
MAINTENANCE	Fast & easy	Slower
SECURITY	simplified	More challenging
SEO	Limited	Easier & more effective
COST	More expensive	Less expensive
SCALABILITY	Not scalable	Scalable







Exemples d'application d'une seule page



ANGULAR

Companies Using Single-page Applications



Twitter



Instagram



Airbnb



Facebook



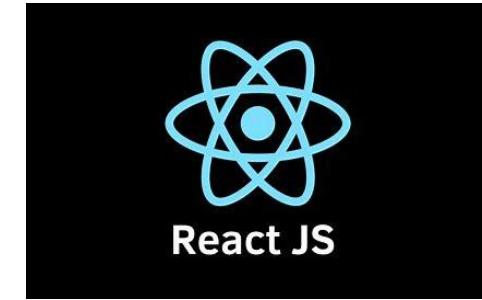
Netflix



Gmail



Uber



SVELTE



Opter pour des applications monopages (SPA) peut offrir une expérience utilisateur plus riche, comme le montrent les refontes de Pandora et Gmail.

Cependant, les avantages ne sont pas automatiques et la sélection d'une technologie doit être réfléchie, en considérant les compétences nécessaires, les coûts, la stabilité, le référencement, et la sécurité.

Pour des projets spécifiques, une approche hybride, comme celle utilisée par Facebook, qui combine un cadre multipage avec des éléments JavaScript pour améliorer l'expérience utilisateur, peut être plus appropriée.

AngularJS / Angular

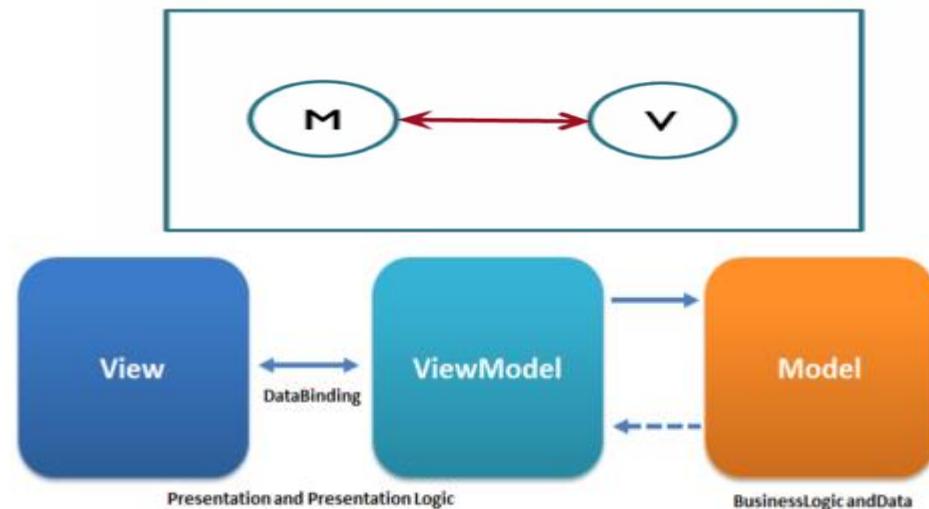
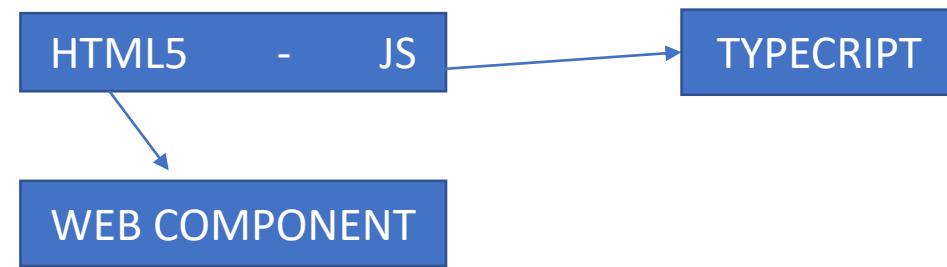
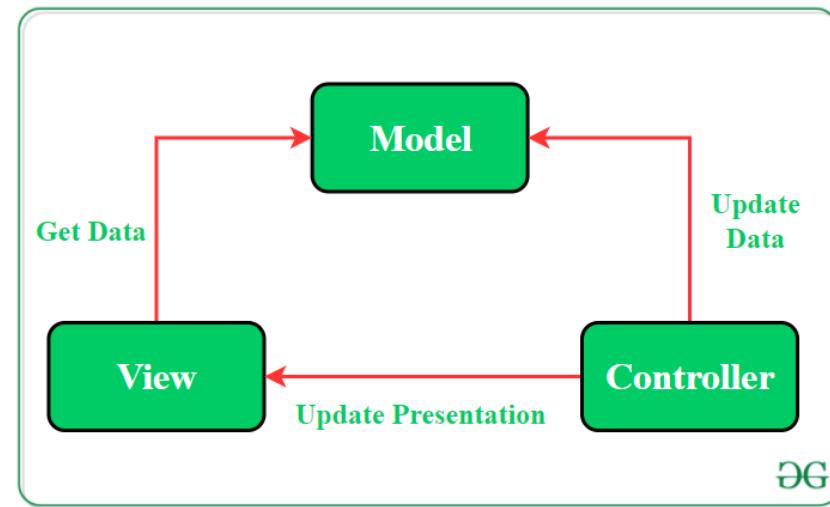
- **Angular 1 (Angular JS) :**

- Première version de Angular qui est la plus populaire.
- Elle est basé sur une architecture MVC coté client.
Les applications Angular 1 sont écrite en Java Script.

- **Angular 2 (Angular) :**

- Est une réécriture de Angular 1 qui est plus performante, mieux structurée et représente le futur de Angular.
- Les applications de Angular2 sont écrites en Type Script qui est compilé et traduit en Java Script avant d'être exécuté par les Browsers Web.
- Angular 2 est basée sur une programmation basée sur les Composants Web (Web Component)

- **Angular 4, 5, 6, ..., 16**



Angular est souvent décrit comme utilisant une architecture qui ressemble à MVVM (Model-View-ViewModel), mais il est plus précis de dire qu'il implémente une architecture basée sur des composants qui incorpore des éléments du pattern Model-View-Controller (MVC) et du MVVM.

MVC (Model-View-Controller)

Dans le pattern MVC, l'application est divisée en trois composants principaux:

Model: Représente la logique de données et les règles d'affaires.

View: Est responsable de l'affichage des données (le UI).

Controller: Agit comme un intermédiaire entre le Model et la View, récupérant les données du Model et les affichant dans la View.

MVVM (Model-View-ViewModel)

MVVM est une adaptation du MVC où le **Controller** est remplacé par le **ViewModel**. Le **ViewModel** est responsable de la liaison des données (**data-binding**) entre la **View** et le **Model**, et il abstrait le code de manipulation de la **View** pour se concentrer sur la logique métier.

Architecture Angular

Angular ne suit pas strictement le pattern MVVM, mais il en partage certains aspects, notamment:

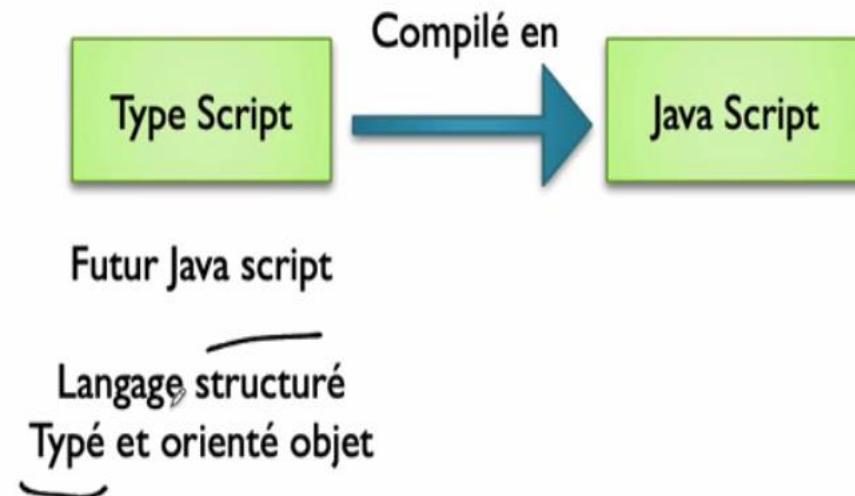
Model: Dans Angular, le Model est généralement représenté par des classes qui définissent la structure des données. Ces données peuvent provenir directement de la base de données ou d'une API.

View: Dans Angular, la View est définie par des templates, qui sont du HTML augmenté de balises et attributs spéciaux d'Angular pour le binding.

Component: Les composants dans Angular jouent le rôle de **contrôleur** et de **ViewModel**. Ils préparent les données pour être affichées dans la View (comme un ViewModel) et peuvent aussi inclure une logique de contrôle (comme un Controller).

TYPE SCRIPT

TypeScript est un langage de programmation développé par Microsoft, souvent décrit comme un sur-ensemble typé de JavaScript. Il ajoute des fonctionnalités de typage statique et des concepts de programmation orientée objet plus avancés à JavaScript, ce qui le rend particulièrement adapté pour développer de grandes applications avec une base de code plus facile à maintenir et à évoluer.



Avantages d'Angular :

- 1. Architecture basée sur les composants :** Angular utilise une architecture modulaire basée sur des composants, facilitant la réutilisation du code et la gestion des projets de grande envergure.
- 2. Ecosystème riche :** Avec le support de Google, Angular bénéficie d'un écosystème riche en bibliothèques, outils, et extensions développés par la communauté.
- 3. Productivité :** Angular CLI (Interface de Commande) offre des outils puissants qui simplifient l'initialisation, le développement, et le déploiement des projets.
- 4. Prise en charge du TypeScript :** TypeScript améliore la maintenabilité du code grâce à un typage statique et des fonctionnalités orientées objet.
- 5. Binding bidirectionnel :** Cette fonctionnalité permet une synchronisation automatique des données entre le modèle et la vue, simplifiant la gestion des mises à jour de l'interface utilisateur.

Cas d'utilisation typiques :

- Applications web dynamiques
- Applications de gestion (ERP, CRM)
- Plateformes de commerce électronique
- Applications mobiles via Angular avec des frameworks comme Ionic
- Etc ...

Comparaison avec d'autres frameworks :

Angular vs. React :

- React est une bibliothèque développée par Facebook, principalement axée sur la construction d'interfaces utilisateur. Il est plus léger que Angular mais nécessite l'intégration de bibliothèques supplémentaires pour des fonctionnalités complètes.
- Angular offre un ensemble complet d'outils et de fonctionnalités prêts à l'emploi pour le développement d'applications, ce qui peut accélérer le développement mais aussi augmenter la courbe d'apprentissage.

Angular vs. Vue :

- Vue est souvent perçu comme étant plus flexible et plus facile à apprendre que Angular, avec une intégration progressive qui permet aux développeurs de l'adopter partiellement.
- Angular, avec son architecture rigide et son écosystème riche, est souvent préféré pour de grands projets d'entreprise nécessitant des fonctionnalités robustes de gestion d'état et de routage.

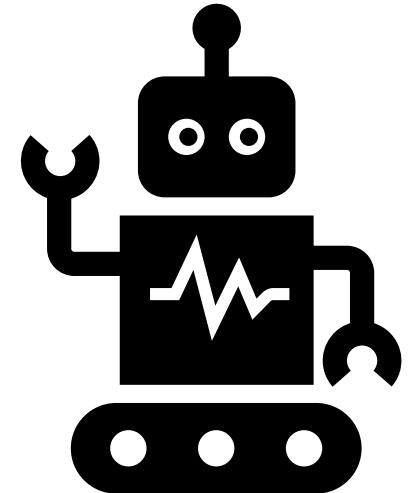
Importance d'Angular dans le développement web moderne :

Angular continue de jouer un rôle clé dans le paysage du développement web moderne grâce à sa stabilité, son soutien par Google, et son adoption massive dans les entreprises.

Pour des projets d'envergure nécessitant des solutions complètes, Angular offre une solution intégrée qui peut réduire la complexité et améliorer la productivité, faisant de lui un choix privilégié dans de nombreux scénarios de développement web d'entreprise.

Guide pas à pas pour installer Node.js et Angular CLI

Configuration de l'environnement de développement (IDE, extensions utiles)





Étape 1 : Installation de Node.js

1. Télécharger Node.js :

1. Rendez-vous sur le site officiel de Node.js (nodejs.org) et téléchargez la version LTS (Long Term Support) qui offre une meilleure stabilité et support à long terme.

2. Installer Node.js :

1. Windows : Exécutez le fichier d'installation téléchargé et suivez les instructions.
2. macOS : Ouvrez le package téléchargé et suivez les étapes d'installation.
3. Linux : Vous pouvez utiliser un gestionnaire de paquets selon votre distribution, par exemple pour Ubuntu :

```
sudo apt update  
sudo apt install nodejs  
sudo apt install npm
```



L'installation de Node.js est une étape essentielle pour utiliser Angular, bien que techniquement, Angular en tant que framework côté client ne nécessite pas Node.js pour exécuter les applications construites avec lui. Cependant, Node.js est crucial pour plusieurs aspects du développement avec Angular, notamment :

Gestion des Dépendances

Node.js inclut npm (Node Package Manager), qui est utilisé pour gérer les dépendances d'un projet Angular. Quand vous travaillez avec Angular, vous avez souvent besoin de divers packages et librairies tiers (comme Angular lui-même, RxJS, zone.js, et autres), et npm est l'outil standard pour gérer ces packages dans l'écosystème JavaScript.

Angular CLI

Angular CLI (Command Line Interface) est un outil puissant qui aide à automatiser le processus de développement d'applications Angular. Angular CLI dépend de Node.js et npm pour son installation et son fonctionnement.

Serveur de Développement

Pour tester et visualiser une application Angular pendant le développement, Angular CLI utilise un serveur de développement (basé sur Node.js) qui permet de servir l'application localement sur votre navigateur. Ce serveur supporte également le chargement en direct (live reloading), qui rafraîchit automatiquement votre application dans le navigateur lorsque vous modifiez le code source.

Build et Compilation

Environnement d'Exécution Uniforme

Build et Compilation

Angular utilise TypeScript, qui doit être transpilé en JavaScript pour être exécuté par les navigateurs. Node.js est nécessaire pour exécuter les outils de build et de compilation (comme Webpack, utilisé sous le capot par Angular CLI) qui transforment le code TypeScript et les templates Angular en JavaScript efficace et optimisé pour la production.

Environnement d'Exécution Uniforme

Utiliser Node.js assure que le développeur dispose d'un environnement uniforme pour la gestion des packages, l'exécution des scripts, et le déploiement, indépendamment du système d'exploitation (Windows, macOS, Linux). Cela simplifie les configurations et réduit les problèmes potentiels entre différents environnements de développement.

Vérifier l'installation :

- Ouvrez une console ou un terminal et tapez les commandes suivantes pour vérifier que Node.js et npm (le gestionnaire de paquets de Node) sont correctement installés :

node --version

npm --version

Étape 2 : Installation d'Angular CLI

Installer Angular CLI :

Utilisez npm pour installer Angular CLI globalement sur votre système :

- `npm install -g @angular/cli`

Vérifier l'installation d'Angular CLI :

Pour confirmer que Angular CLI est installé correctement, tapez :

- `ng version`

Étape 3 : Configuration de l'environnement de développement

1. Choisir un IDE :

- Visual Studio Code (VS Code) est fortement recommandé pour le développement Angular en raison de sa légèreté, de sa personnalisation, et de son intégration étroite avec TypeScript.
- Téléchargez et installez VS Code depuis code.visualstudio.com.

2. Installer des extensions utiles dans VS Code :

- **Angular Language Service** : Offre une auto-complétion intelligente, des erreurs de diagnostic, etc.
- **Angular Snippets** : Fournit des extraits de code pour Angular.
- **ESLint** : Un linter puissant pour JavaScript et TypeScript.
- **Prettier** : Un formateur de code pour maintenir un style cohérent.

3. Configurer VS Code :

- Ouvrez VS Code et allez dans l'onglet "Extensions" (icône des blocs sur la barre latérale gauche).
- Recherchez et installez les extensions mentionnées ci-dessus.
- Configurez Prettier et ESLint selon vos préférences de codage en modifiant les fichiers de configuration .prettierrc et .eslintrc respectivement.

Étape 4 : Créer et exécuter une nouvelle application Angular

1. Créez une nouvelle application :

Ouvrez un terminal et tapez :

- `ng new mon-app`

Suivez les instructions pour configurer le routing et les styles.

2. Démarrer l'application :

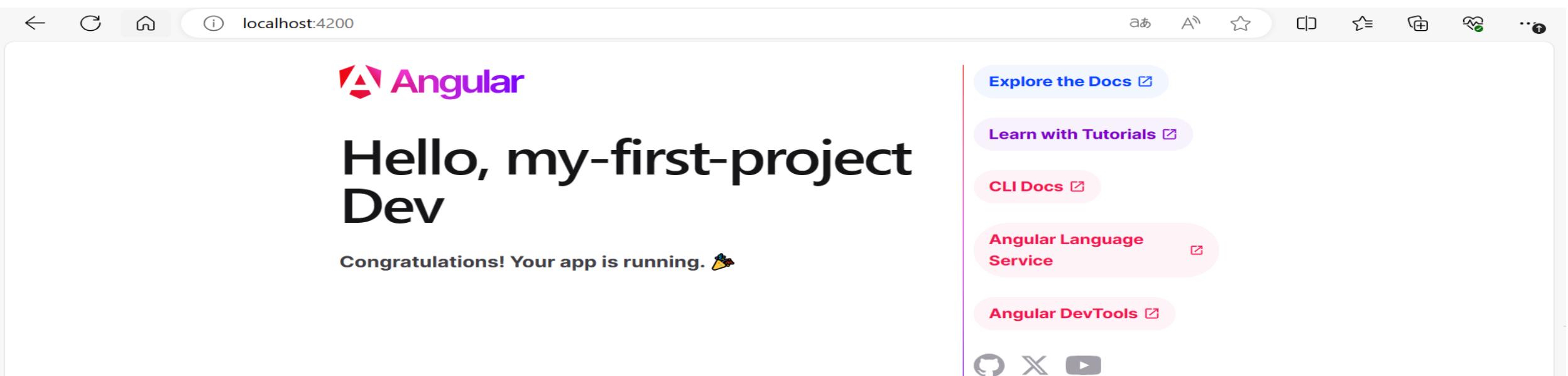
Naviguez dans le répertoire de votre nouvelle application et lancez le serveur :

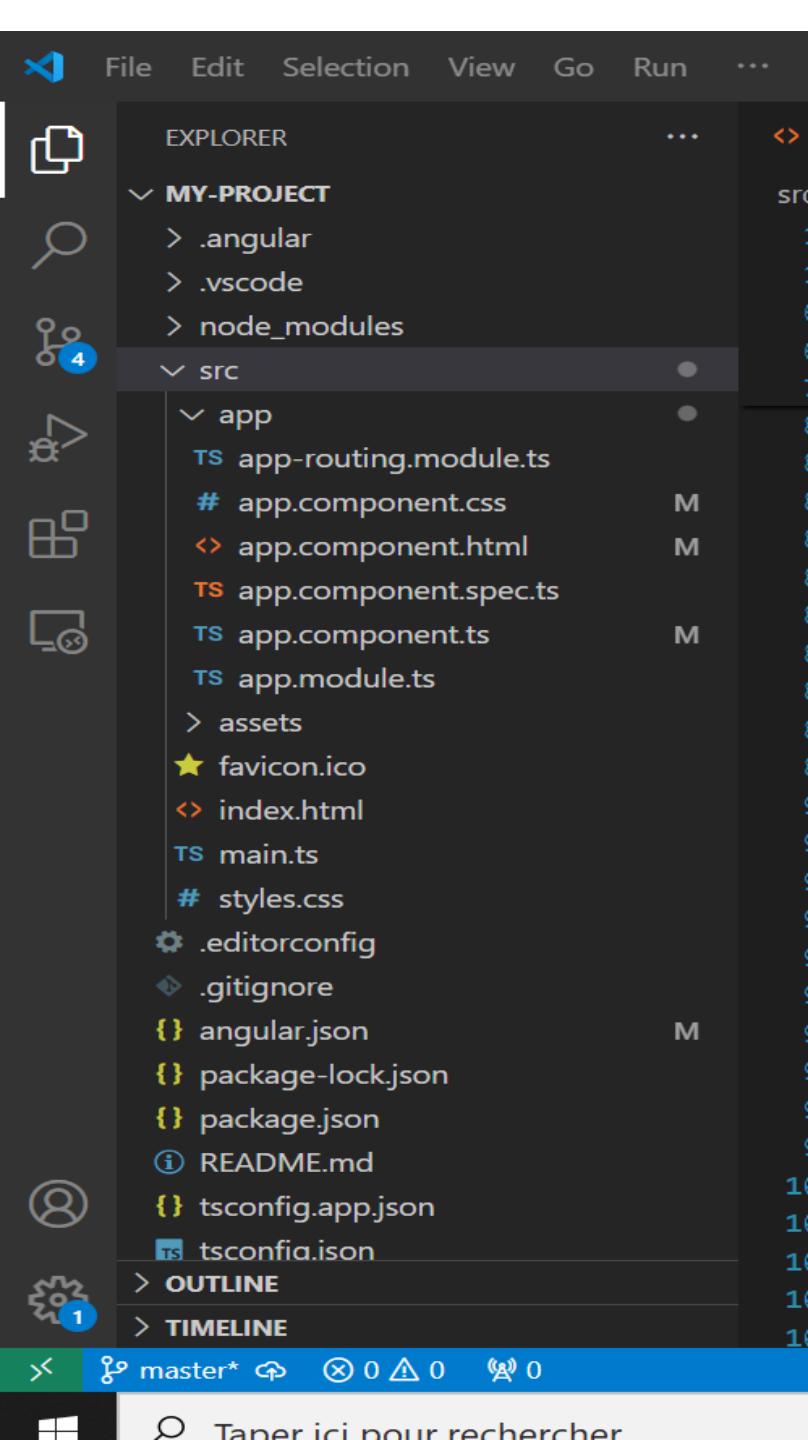
- `cd mon-app`
- `ng serve --open`

`ng new frontend-ang --directory=../ --no-standalone`

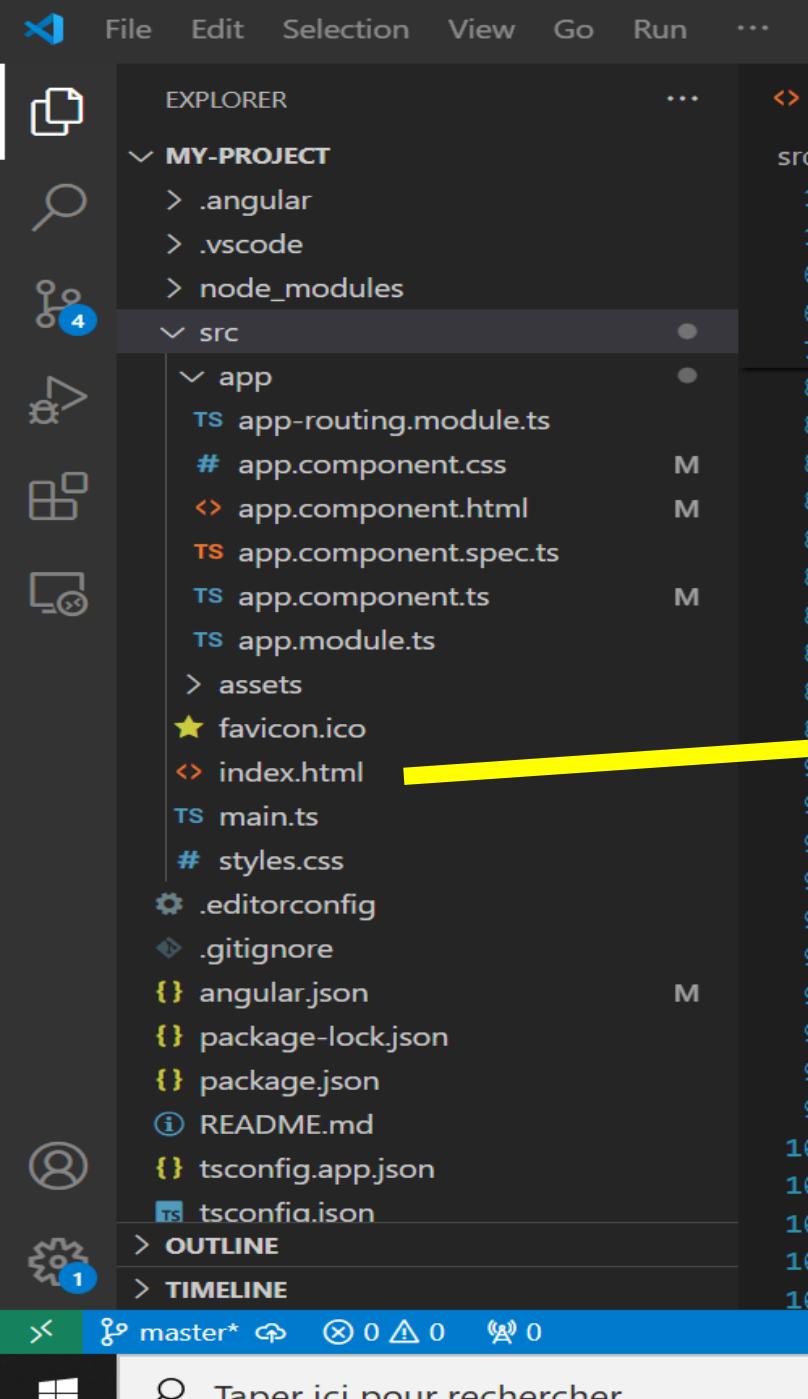
- > **ng serve**

- Cette commande compile le code source du projet pour transpiler le code TypeScript en Java Script et en même temps démarre un serveur Web local basé sur Node JS pour déployer l'application localement.
- Pour tester le projet généré, il suffit de lancer le Browser et taper l'url : `http://localhost:4200`
- Dans l'étape suivante, nous allons regarder la structure du projet généré par Angular CLI.

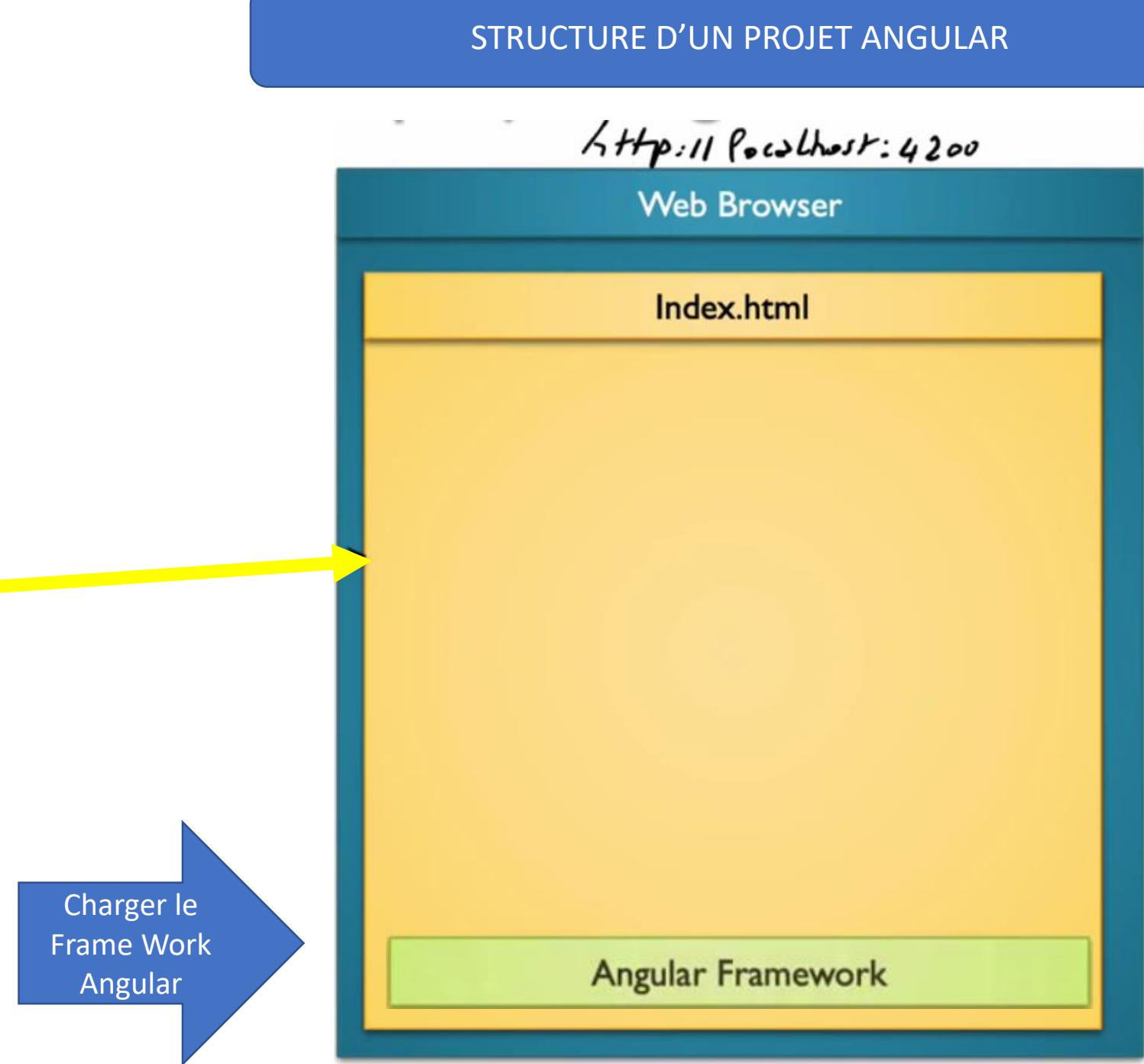




STRUCTURE D'UN PROJET ANGULAR



STRUCTURE D'UN PROJET ANGULAR



File Edit Selection View Go Run ...

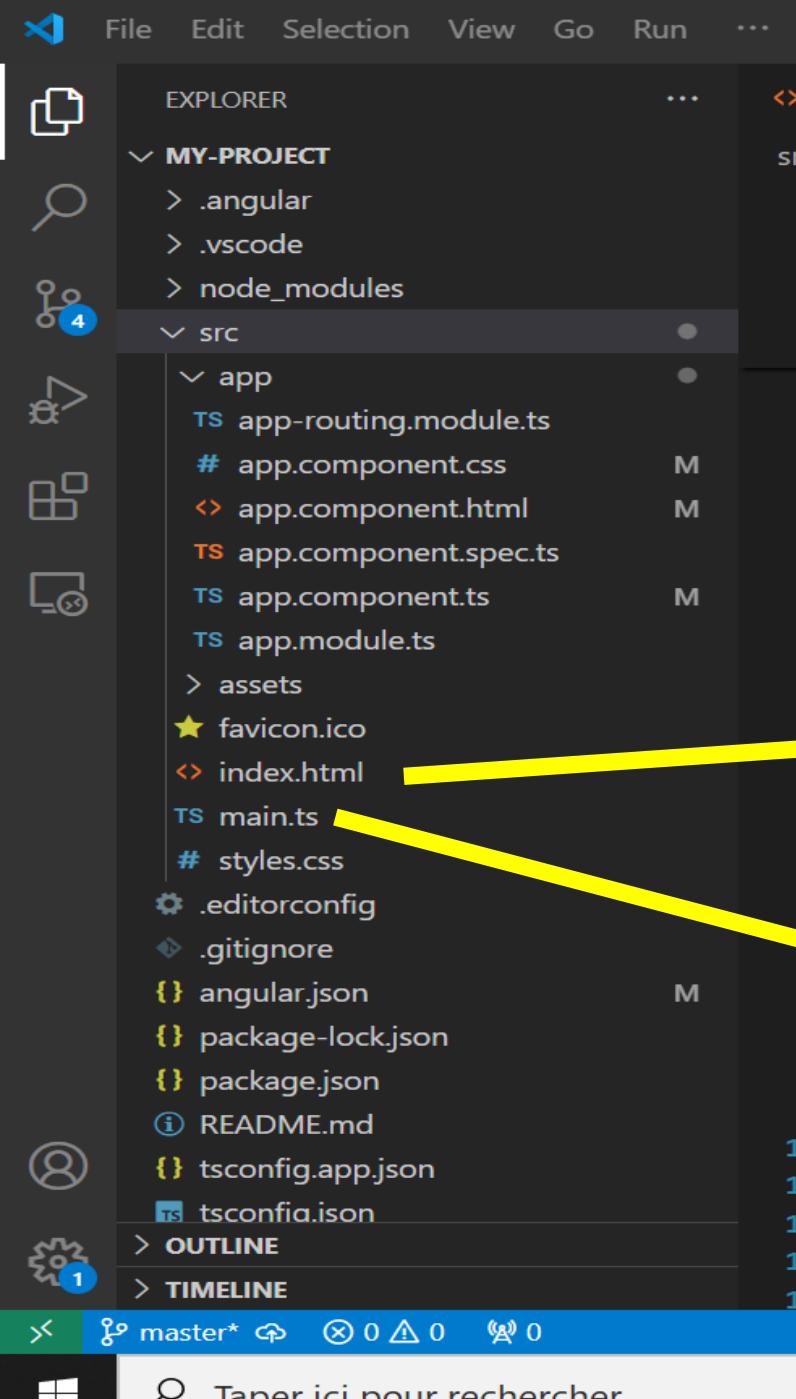
EXPLORER

MY-PROJECT

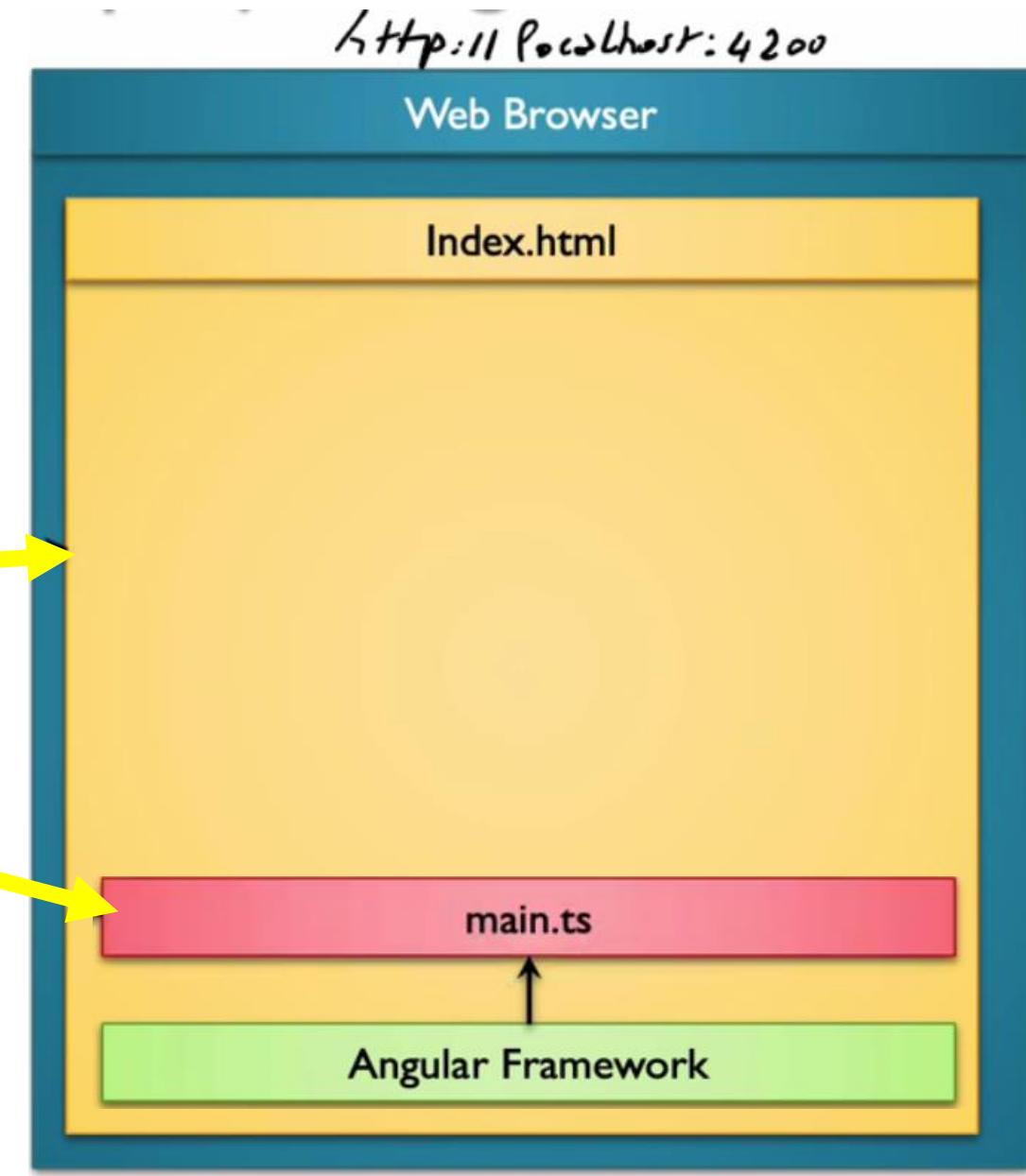
- > .angular
- > .vscode
- > node_modules
- src
 - app
 - TS app-routing.module.ts
 - # app.component.css M
 - <> app.component.html M
 - TS app.component.spec.ts
 - TS app.component.ts M
 - TS app.module.ts
 - > assets
 - ★ favicon.ico
 - <> index.html
 - TS main.ts
 - # styles.css
 - .editorconfig
 - .gitignore
 - { } angular.json
 - { } package-lock.json
 - { } package.json
 - (i) README.md
 - { } tsconfig.app.json
 - TS tsconfia.json
- > OUTLINE
- > TIMELINE

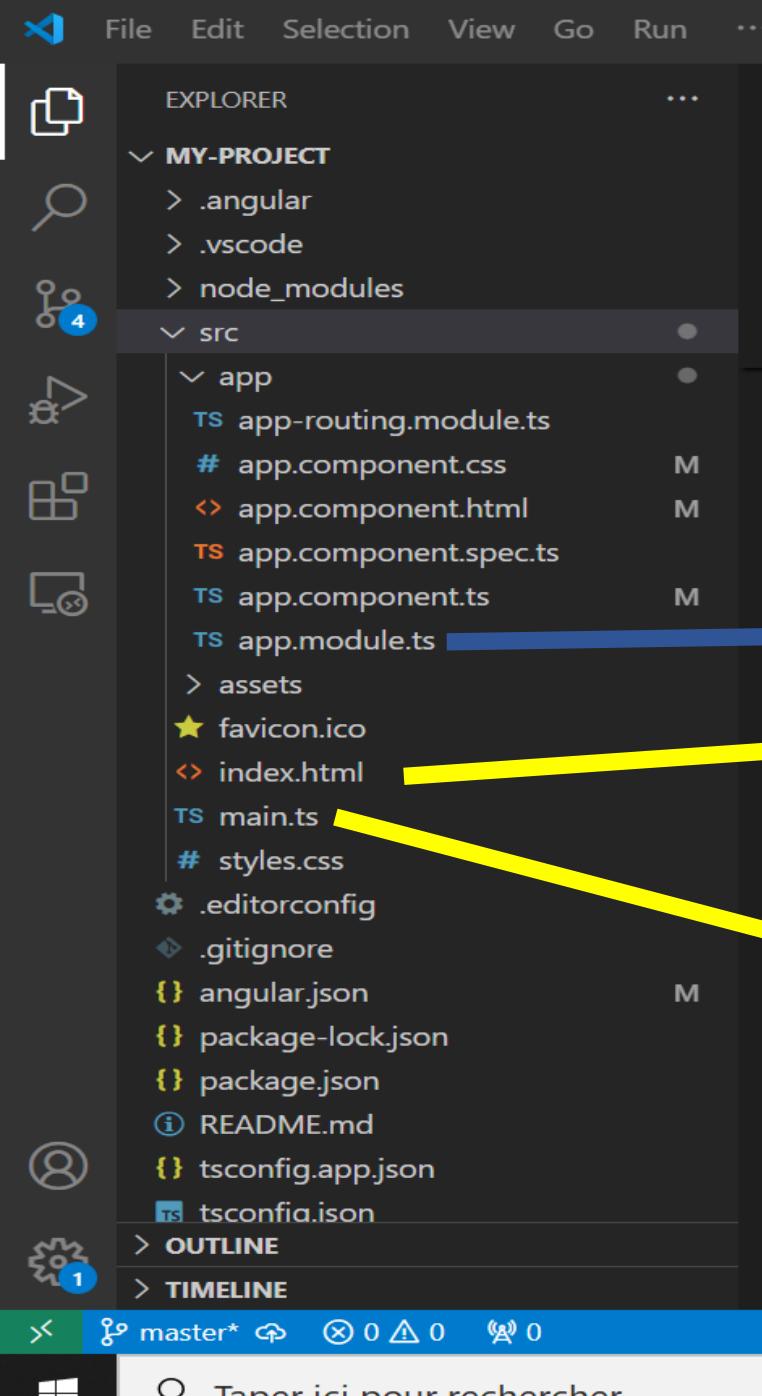
master* ↗ 0 ▲ 0 0 0

Taper ici pour rechercher

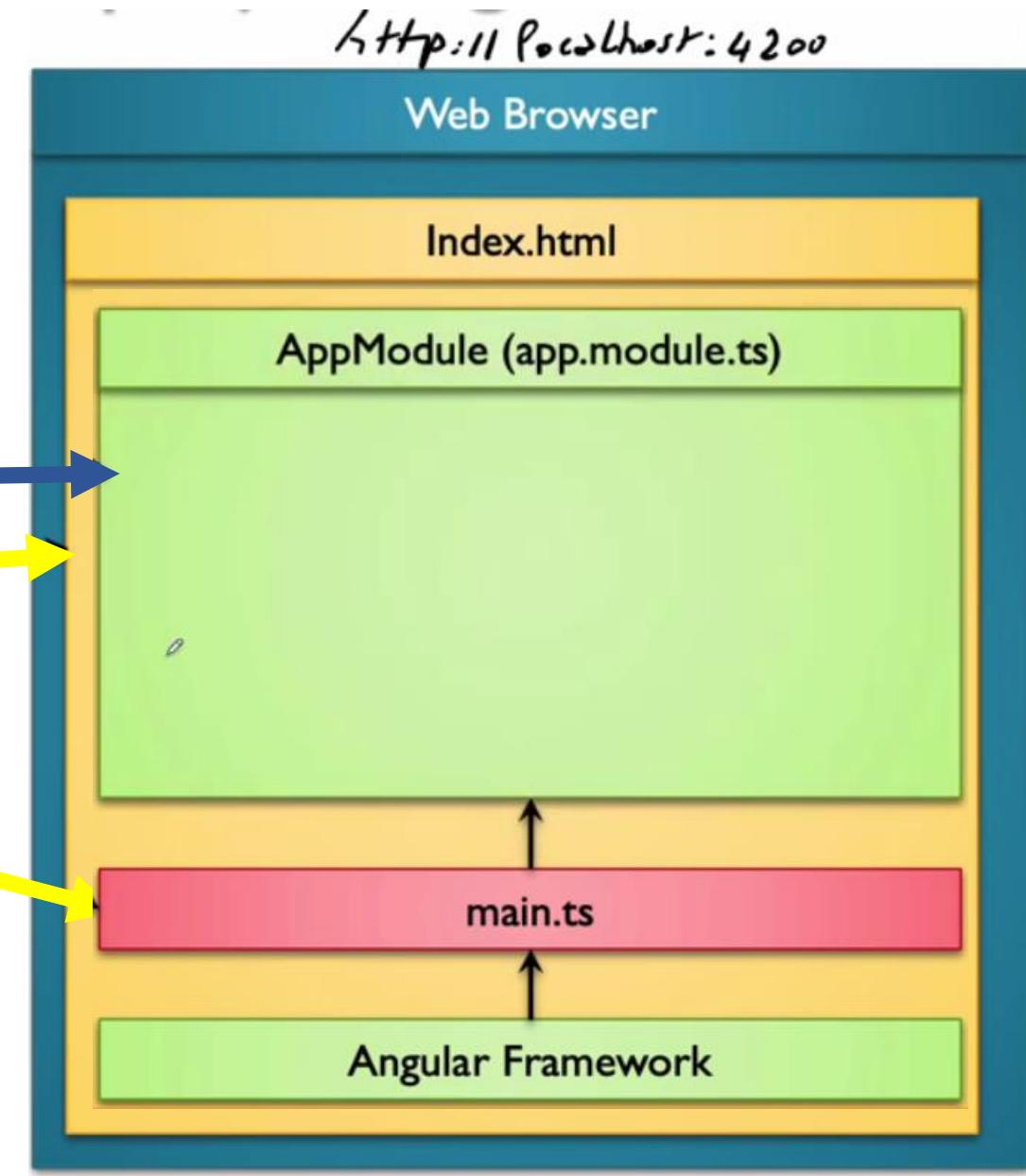


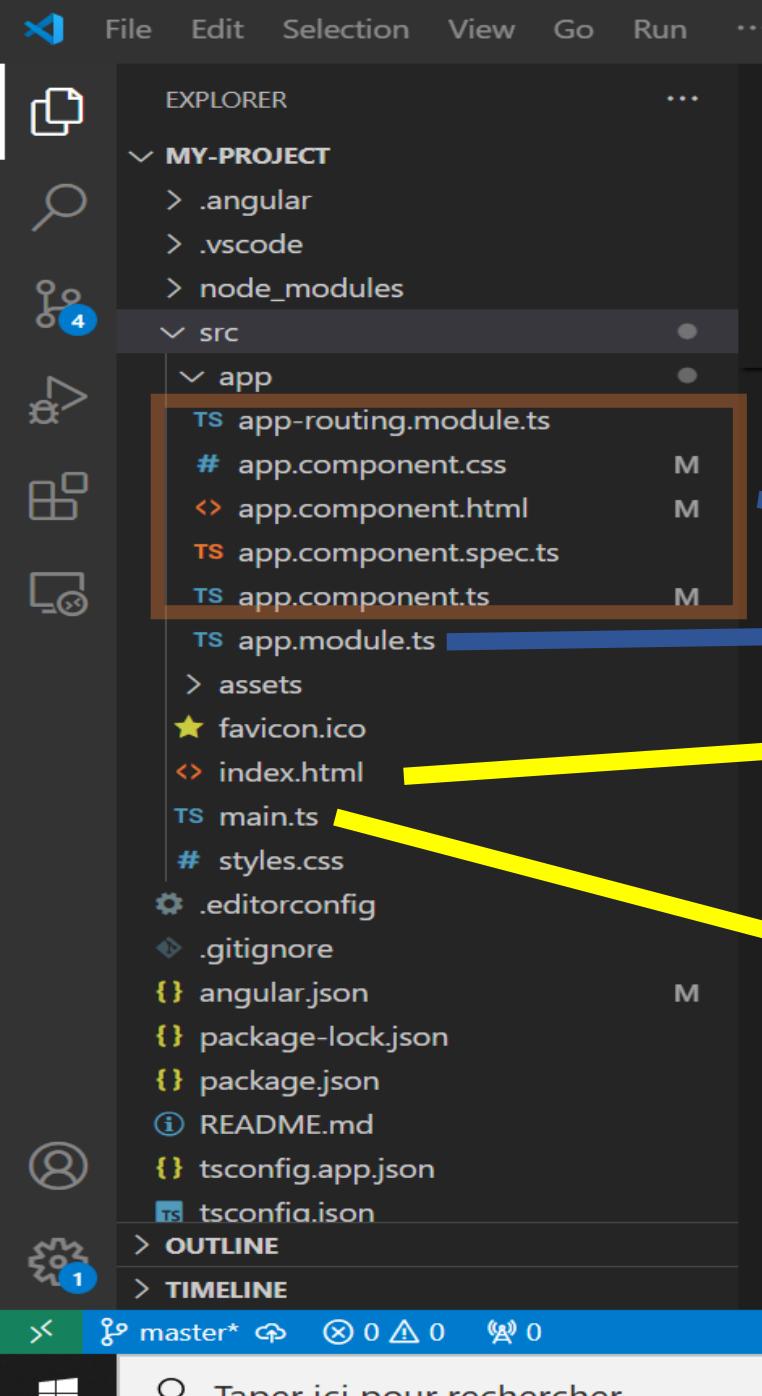
STRUCTURE D'UN PROJET ANGULAR



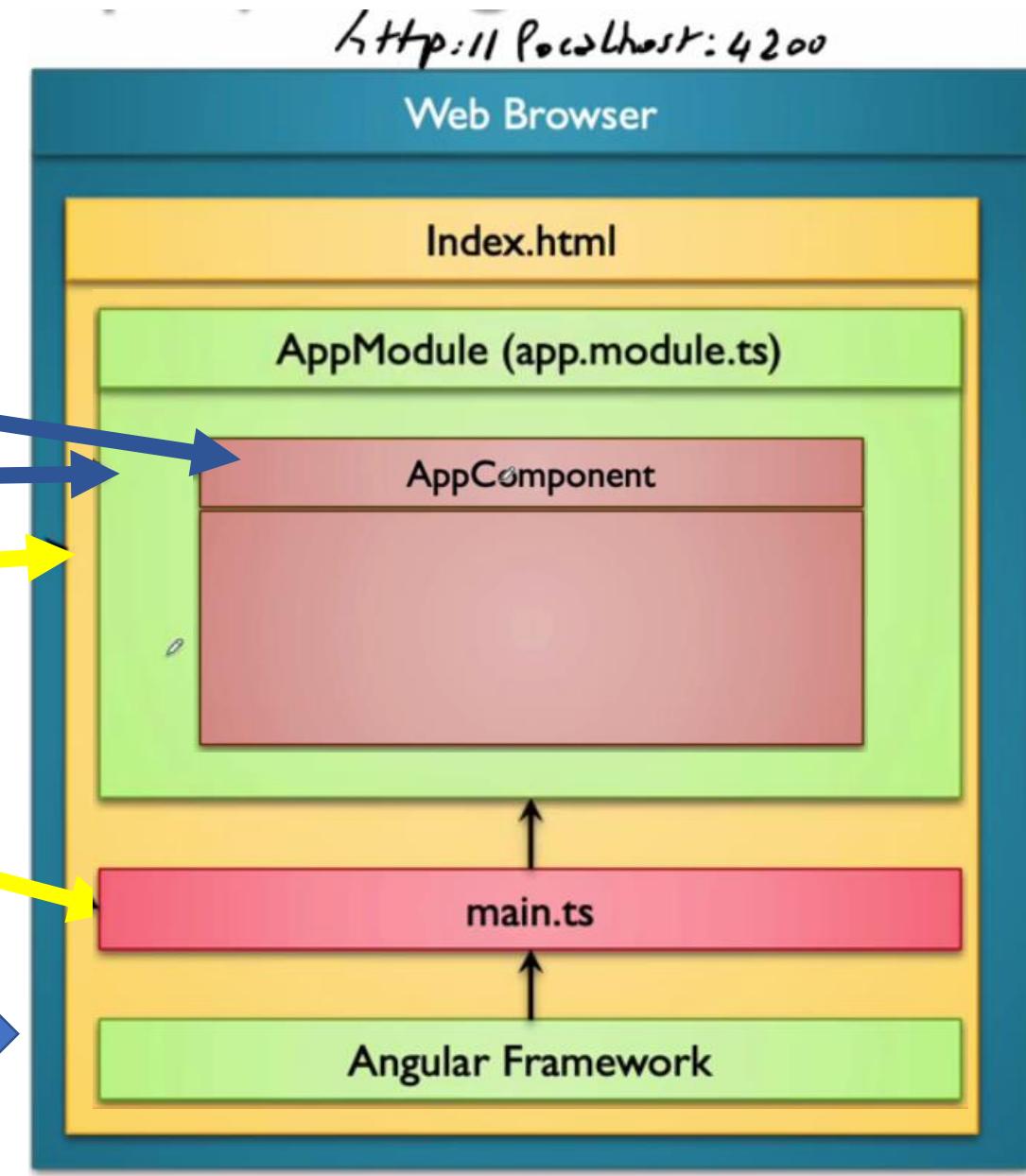


STRUCTURE D'UN PROJET ANGULAR

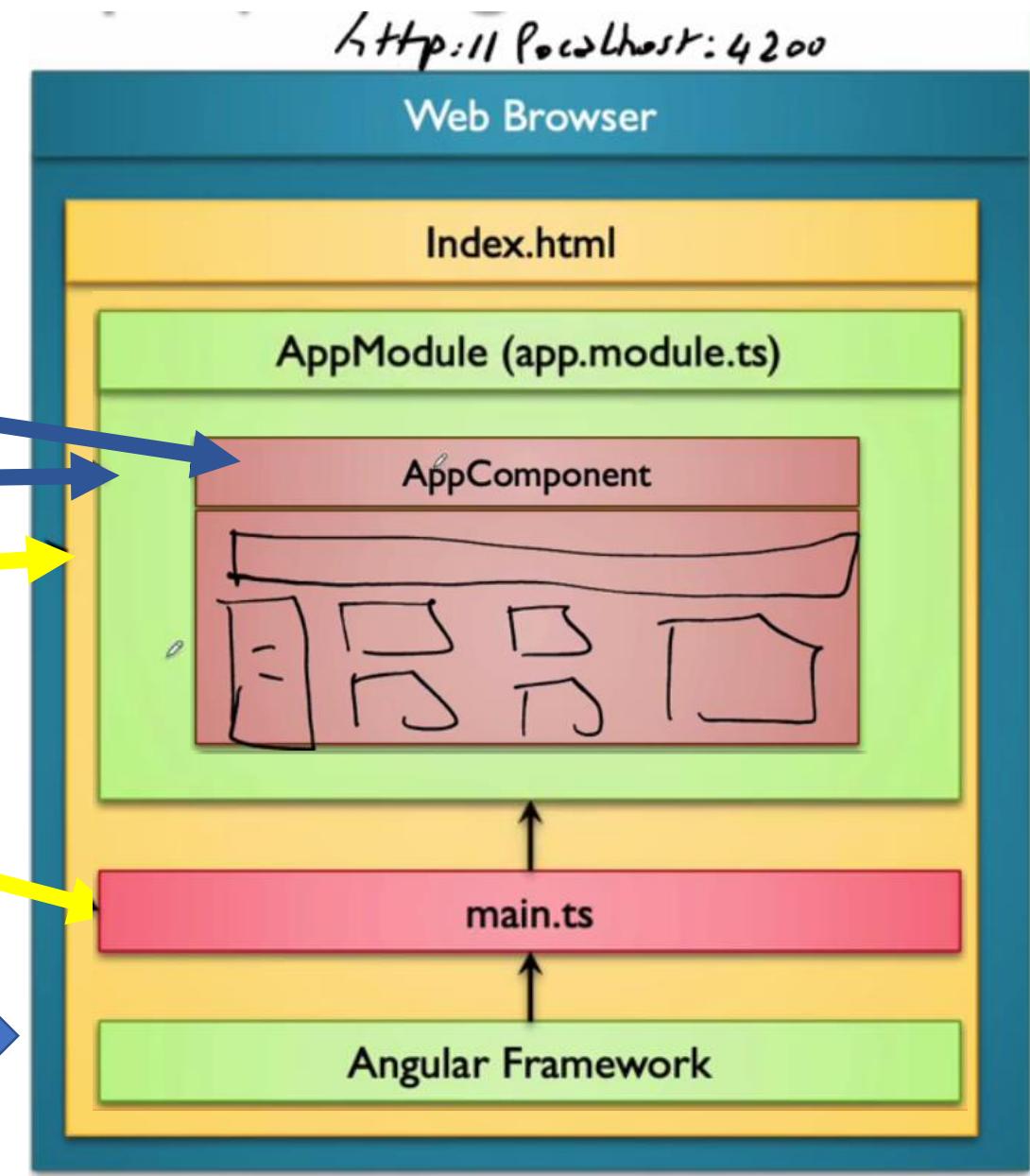
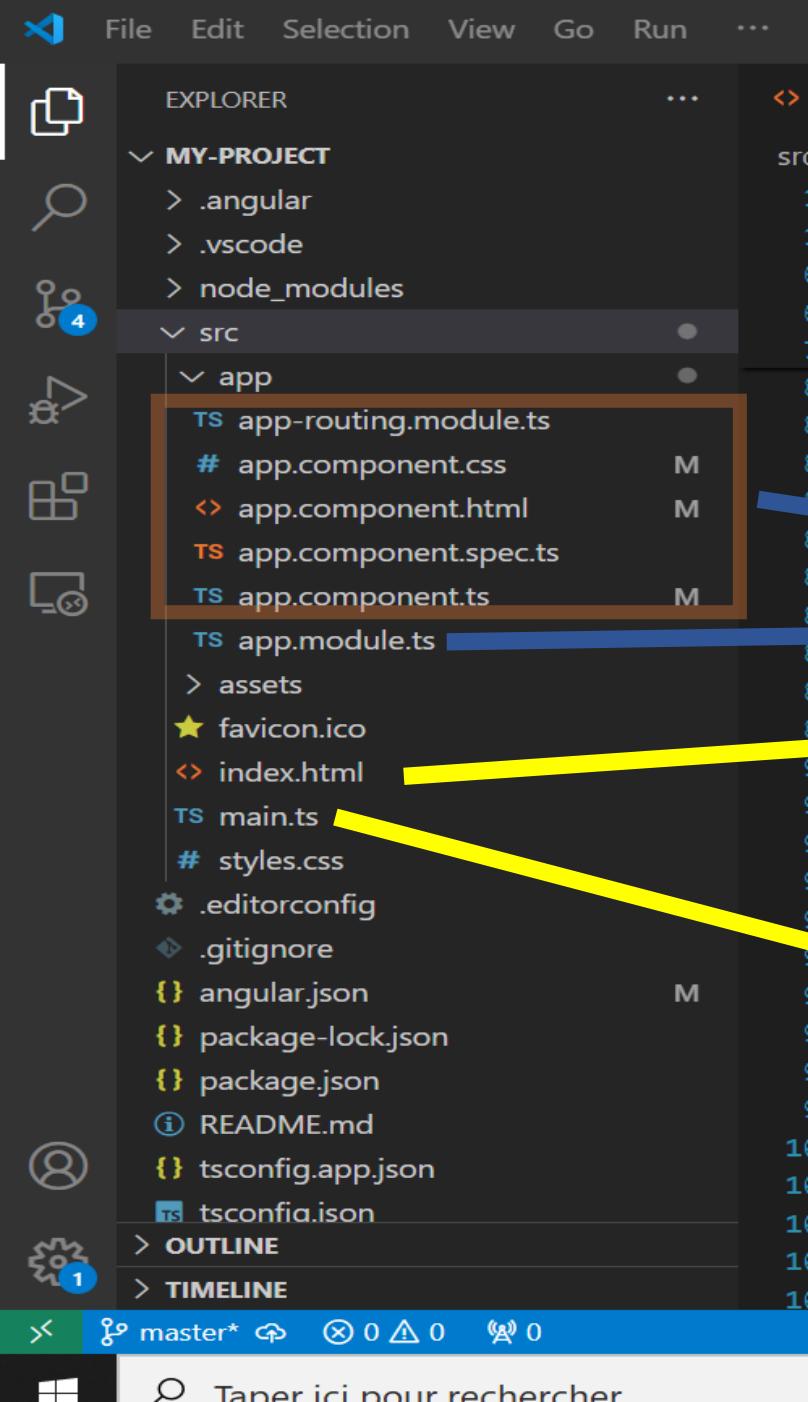




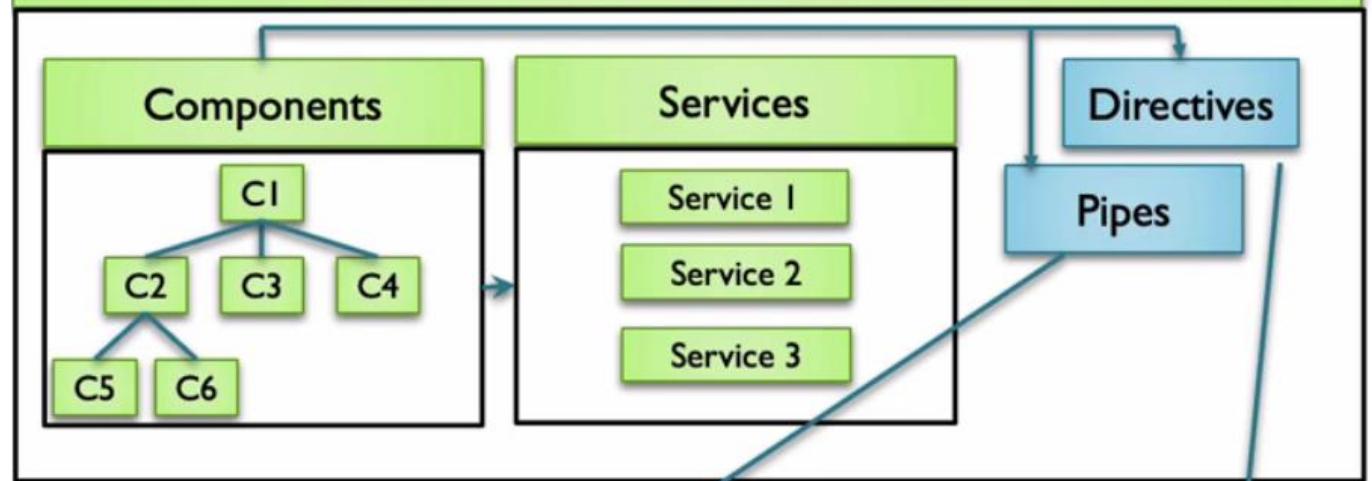
STRUCTURE D'UN PROJET ANGULAR



STRUCTURE D'UN PROJET ANGULAR



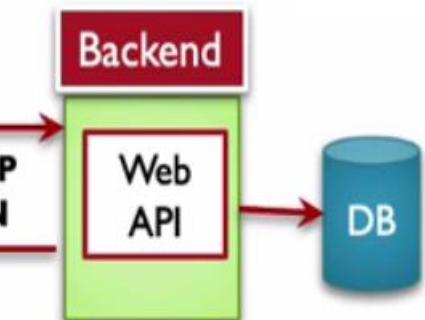
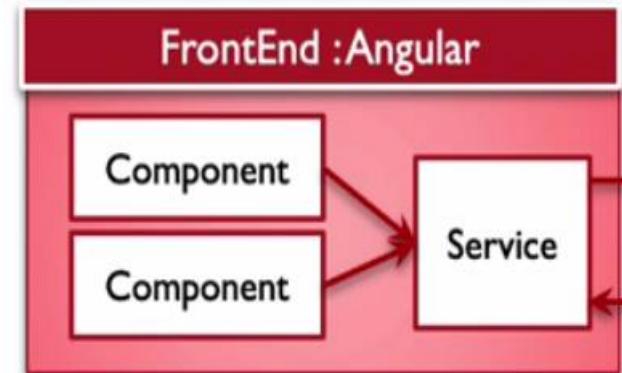
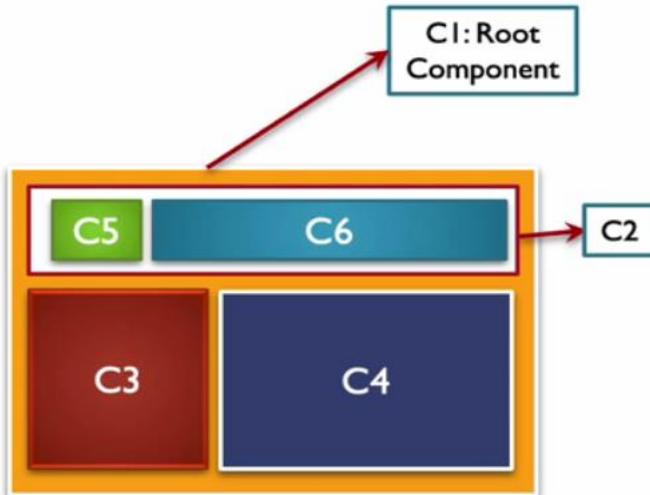
Module Angular : app.module.ts



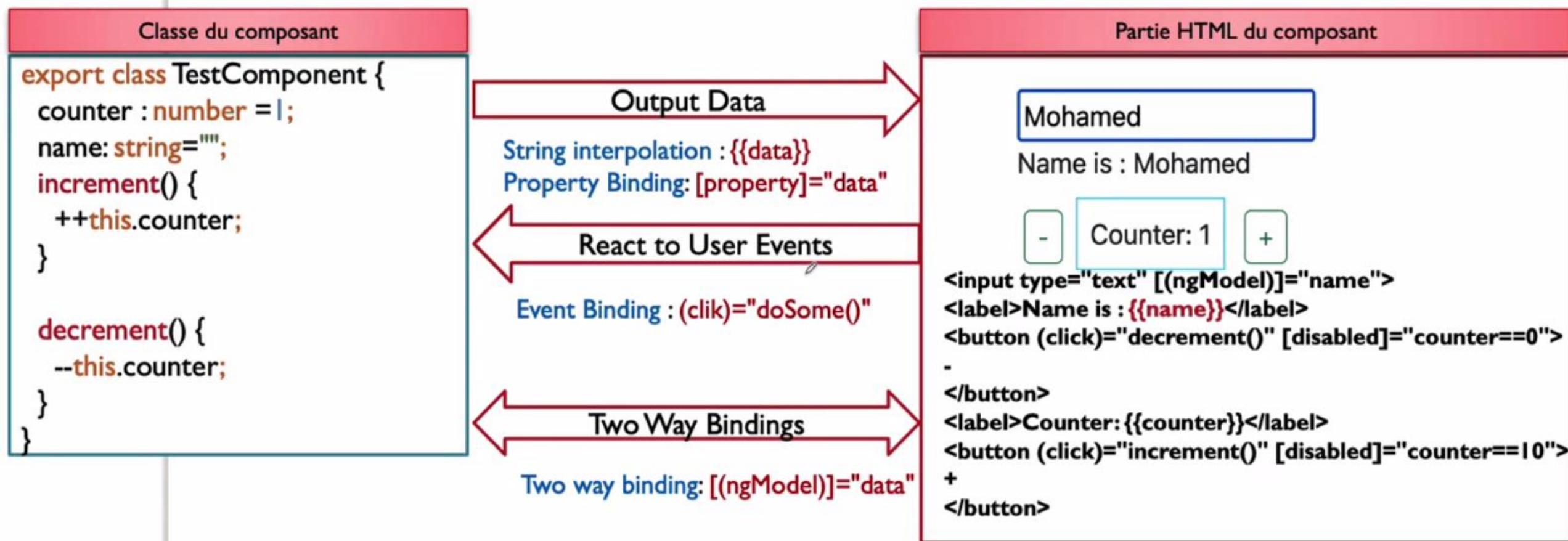
```
{ {dateNaissance | date : 'dd/MM/yyyy' } }
```

```
<input type="text" required />
```

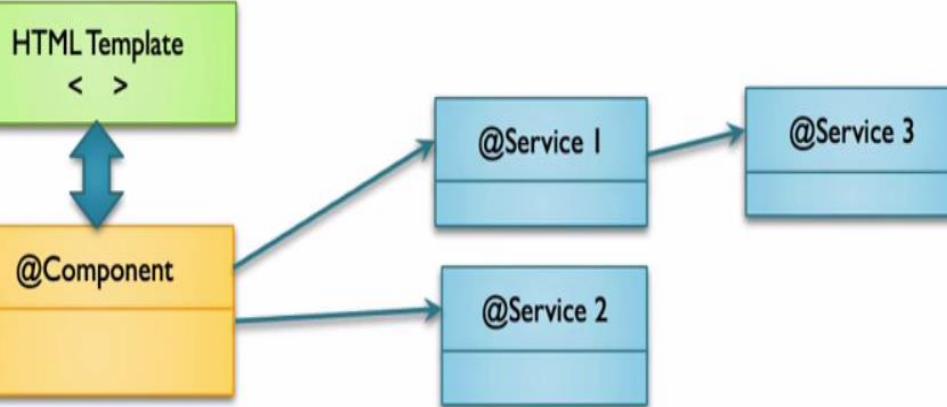
C1: Root Component



DATA BIDING



SERVICES



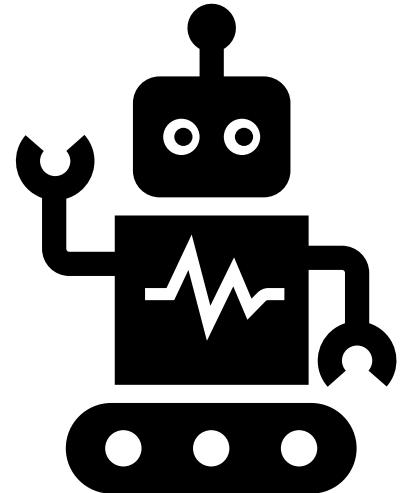
exemple.service.ts

```
import { Injectable } from '@angular/core';
@Injectable()
export class ExempleService {
  constructor() { }
  saveData(data) {
    console.log('saving data...');
  }
  getData() {
    return 'getting data...';
  }
}
```

exemple.component.ts

```
import { Component, OnInit } from '@angular/core';
import { ExempleService } from "../exemple.service";
@Component({
  selector: 'app-exemple',
  templateUrl: './exemple.component.html',
  styleUrls: ['./exemple.component.css']
})
export class ExempleComponent {
  result:string
  constructor(private exempleService:ExempleService) { }
  onSave(data) {
    this.exempleService.saveData(data);
  }
  onGetData() {
    this.result = this.exempleService.getData();
  }
}
```


Explication des modules Angular
Compréhension des composants et de
leur rôle
Utilisation des templates pour afficher
des données



Dans Angular, l'architecture de l'application est principalement construite autour de trois concepts fondamentaux : **les modules, les composants et les templates.**

Modules Angular

Un module Angular encapsule un ensemble cohérent de fonctionnalités qui sont liées et fonctionnent ensemble. Chaque application Angular a au moins un module racine, généralement appelé AppModule, qui sert à démarrer et à assembler l'application. Les modules sont définis en utilisant le décorateur @NgModule, qui prend en charge plusieurs propriétés pour configurer le module :

- **declarations** : Liste les composants, directives et pipes que le module gère.
- **imports** : Importe d'autres modules dont les composants ou services sont nécessaires.
- **providers** : Déclare les services que l'application utilise.
- **bootstrap** : Indique le composant racine qui doit être chargé lorsque l'application démarre.
- **exports** : Permet de rendre certaines parties du module disponibles pour d'autres modules.
- Les modules facilitent la modularité de l'application, la réutilisabilité du code, et peuvent être chargés de manière précieuse pour améliorer les performances de l'application.

Composants

Les composants sont les blocs de construction fondamentaux des applications Angular. Ils contrôlent une partie de l'écran appelée vue. Chaque composant est défini avec une classe contenant des données et des logiques, décorée par `@Component`, qui associe un template et éventuellement des styles spécifiques. les principaux aspects d'un composant :

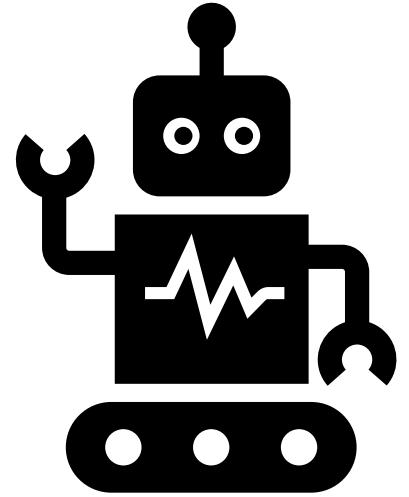
- **Template** : Le HTML qui définit la vue que le composant contrôle.
- **Class** : Gère les données et les comportements de la vue.
- **Metadata** : Fournies par le décorateur `@Component`, définissent la manière dont le composant doit être traité par Angular.
- Les composants sont généralement organisés hiérarchiquement, permettant une encapsulation efficace des fonctionnalités et une gestion simplifiée de l'interface utilisateur.

Templates et Data Binding

Les templates d'Angular utilisent le HTML augmenté de syntaxes spéciales qui permettent de lier les éléments de l'interface utilisateur aux données des composants. Angular supporte plusieurs types de data binding :

- **Interpolation** : Utilise `{{ value }}` pour insérer des valeurs dynamiques dans le HTML.
- **Property Binding** : Lie une propriété d'un élément HTML à une propriété du composant via `[property]="value"`.
- **Event Binding** : Permet au template d'écouter et de réagir aux actions de l'utilisateur via `(event)="handler()"`.
- **Two-way Binding** : Combine le property et event binding pour permettre une interaction bidirectionnelle entre le template et le composant, utilisant `[(ngModel)]="property"`.
- Les templates servent donc non seulement à définir la structure de l'interface utilisateur, mais aussi à établir une communication interactive entre l'interface et la logique de l'application. Cela rend les applications Angular très réactives et dynamiques.

TypeScript.



TypeScript est apprécié par les développeurs familiers avec les langages à typage statique comme Java et est le langage principal d'Angular.

Il offre des avantages similaires à ceux de Java/C#, tels que l'amélioration de l'autocomplétion, la détection précoce des erreurs, et une communication plus efficace au sein du code.

Bien que proche des langages POO en termes de fonctionnalités, il est important de comprendre comment TypeScript diffère de ces langages pour éviter les erreurs communes et améliorer la qualité du code JavaScript.

Installing & Using TypeScript

The screenshot shows a web browser window displaying the TypeScript download page at <https://www.typescriptlang.org/download/>. The browser's address bar and various icons are visible at the top. The main content area features a large heading "Download TypeScript" and two sections: "TypeScript in Your Project" and "via npm" and "with Visual Studio". The "TypeScript in Your Project" section contains text about using TypeScript per-project basis. The "via npm" and "with Visual Studio" sections provide links for installation. At the bottom, there's a search bar and a taskbar with several pinned application icons.

Download TypeScript

TypeScript can be installed through three installation routes depending on how you intend to use it: an npm module, a NuGet package or a Visual Studio Extension.

If you are using Node.js, you want the npm version. If you are using MSBuild in your project, you want the NuGet package or Visual Studio extension.

TypeScript in Your Project

Having TypeScript set up on a per-project basis lets you have many projects with many different versions of TypeScript, this keeps each project working consistently.

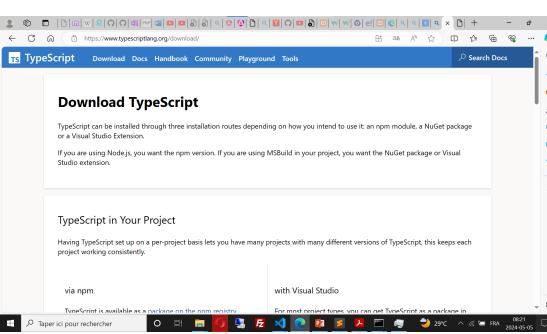
via npm

with Visual Studio

TypeScript is available as a [package on the npm registry](#).

For most project types, you can get TypeScript as a package in

Installing & Using TypeScript



npm install -g typescript

```
npm install typescript --save-dev
```

via npm

TypeScript is available as a [package on the npm registry](#) available as `."typescript"`

You will need a copy of [Node.js](#) as an environment to run the package. Then you use a dependency manager like [npm](#), [yarn](#) or [pnpm](#) to download TypeScript into your project.

npm install typescript --save-dev

npm yarn pnpm

All of these dependency managers support lockfiles, ensuring that everyone on your team is using the same version of the language. You can then run the TypeScript compiler using one of the following commands:

npx tsc

npm yarn pnpm

npx tsc --version

with Visual Studio

For most project types, you can get TypeScript as a package in Nuget for your MSBuild projects, for example an ASP.NET Core app.

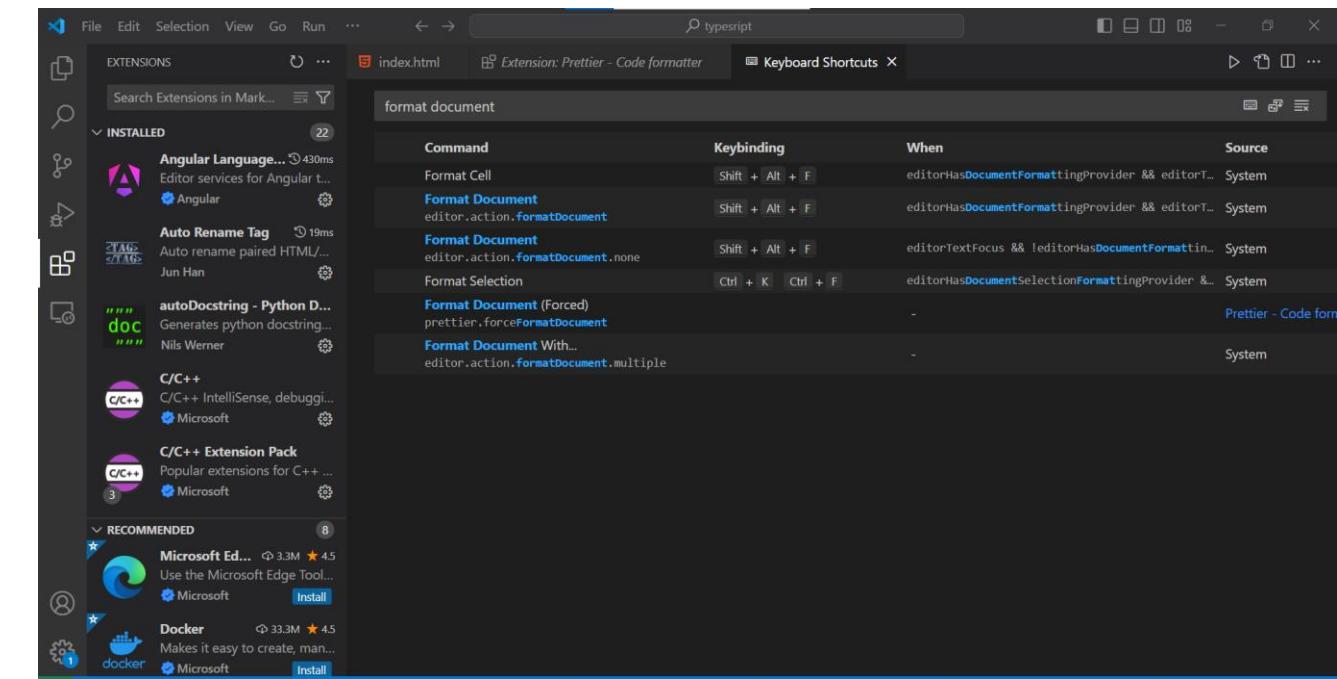
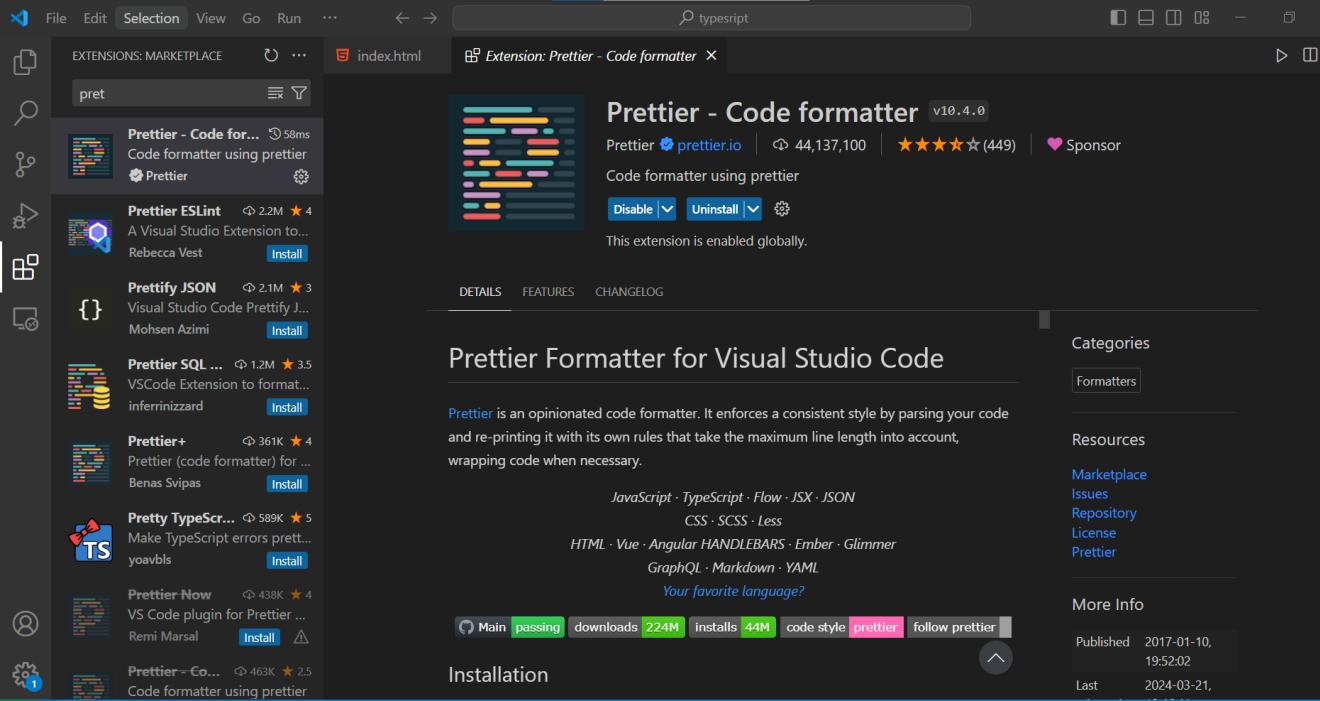
When using Nuget, you can [install TypeScript through Visual Studio](#) using:

- The Manage NuGet Packages window (which you can get to by right-clicking on a project node)

- The Nuget Package Manager Console (found in Tools > NuGet Package Manager > Package Manager Console) and then running:

`Install-Package Microsoft.TypeScript.MSBuild`

For project types which don't support Nuget, you can use the [TypeScript Visual Studio extension](#). You can [install the extension](#) using in Visual Studio. Extensions > Manage Extensions



Using Types

Types intégrés

JavaScript définit 8 types intégrés :

Type	Explication
Number	une virgule flottante IEEE 754 double précision.
String	une chaîne UTF-16 immuable.
BigInt	entiers dans le format de précision arbitraire.
Boolean	<code>true</code> et <code>false</code>
Symbol	Valeur unique généralement utilisée comme clé.
Null	équivalent au type d'unité.
Undefined	également équivalent au type d'unité.
Object	similaires aux enregistrements.

TypeScript a des types primitifs correspondants pour les types intégrés :

- `number`
- `string`
- `bignint`
- `boolean`
- `symbol`
- `null`
- `undefined`
- `object`

Using Types

Autres types TypeScript importants

Type	Explication
unknown	le type supérieur.
never	le type inférieur.
littéral d'objet	Eg { property: Type }
void	pour les fonctions sans valeur de retour documentée
T[]	tableaux mutables, également écrits Array<T>
[T, T]	les tuples, qui sont de longueur fixe mais modifiables
(t: T) => U	Fonctions

Using Types

Working with Numbers, Strings & Booleans

Using Types

Type Assignment & Type Inference

Using Types

Object Types

Nested Objects & Types

Using Types

Arrays Types

Using Types

Working with Tuples

Les tuples en TypeScript permettent de stocker un tableau de valeurs de types multiples, mais définis. La longueur et le type de chaque élément dans le tuple sont fixés dès la déclaration. Cela est utile pour avoir un contrôle strict sur le nombre d'éléments et leurs types, ce qui n'est pas possible avec les tableaux simples.

```
let myTuple: [number, string, boolean] = [5, "Hello", true];
```

Using Types

Working with Enums

Les enums sont une fonctionnalité qui permet de définir un ensemble de constantes nommées sous un seul type. Ils sont utiles pour attribuer des noms descriptifs à des ensembles de valeurs numériques.

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

Using Types

The "any" Type

Le type any en TypeScript est un type joker qui peut être attribué à une variable dont le type peut changer ou est inconnu lors de l'écriture du script. L'utilisation de any annule essentiellement le contrôle de type strict de TypeScript, ce qui peut être utile dans certains cas, mais il est généralement conseillé de l'éviter pour maintenir les bénéfices de la typographie forte.

```
let notSure: any = 4;  
notSure = "maybe a string instead";  
notSure = false; // okay, definitely a boolean
```

Using Types

Union Types

Un type union en TypeScript permet de déclarer une variable qui peut tenir plus d'un type. Les types sont séparés par des barres verticales (|). Cela est utile lorsqu'une variable peut être de plusieurs types, et TypeScript s'assurera que toutes les opérations sur cette variable sont valides pour chaque type possible.

```
let multiType: number | string;  
multiType = 20; // Okay  
multiType = "twenty"; // Okay
```

Using Types

Literal Types

Les types littéraux permettent de restreindre une variable à des valeurs spécifiques. En utilisant des types littéraux, vous pouvez indiquer qu'une variable ne peut avoir que certaines valeurs définies. Cela renforce la précision du code et la documentation par le type.

```
let myTrue: true = true;  
let specificString: "fixed" = "fixed";
```

Using Types

Type Aliases / Custom Types

Les aliases de types (ou types personnalisés) permettent de définir un type et de le réutiliser sous un nouveau nom. Cela est utile pour simplifier des types complexes ou pour donner des noms descriptifs aux types utilisés dans plusieurs endroits du code.

```
type StringOrNumber = string | number;
```

```
type User = { name: string; id: StringOrNumber };
```

```
let user: User = { name: "John", id: "1234" };
```

Using Types

function Return Types & "void"

En TypeScript, vous pouvez spécifier le type de la valeur renvoyée par une fonction. Si une fonction ne retourne rien, vous pouvez utiliser le type void. Cela indique clairement qu'il n'y a rien à retourner.

functions as Types

TypeScript permet d'utiliser des fonctions comme types. Cela signifie que vous pouvez définir des variables qui sont du type fonction avec une signature spécifique.

```
let callFunction: (name: string) => void;
```

```
callFunction = name => {  
    console.log(`Hello, ${name}`);};
```

Using Types

Function Types & Callbacks

Les types de fonctions sont particulièrement utiles pour définir des callbacks. Vous pouvez spécifier le type de fonction attendue en tant que paramètre d'une autre fonction, incluant le type des arguments et le type de retour.

```
function processUserInput(callback: (input: string) => void) {  
  let input = "Hello from callback!";  
  callback(input);  
}  
processUserInput(input => {  
  console.log(input);  
});
```

Using Types

The "unknown" Type

Le type `unknown` en TypeScript est similaire à `any`, mais plus sécurisé. Une variable de type `unknown` ne peut pas être utilisée directement sans une assertion de type ou une vérification de type, ce qui renforce la sécurité du code.

```
let userInput: unknown;
userInput = 5;
userInput = "Hello";

if (typeof userInput === "string") {
  console.log(userInput.length); // OK car la vérification de type est faite
}
```

Using Types

The "never" Type

Le type never indique que quelque chose ne doit jamais arriver. Par exemple, il est utilisé pour le type de retour des fonctions qui ne retournent jamais normalement (par exemple, une fonction qui lance toujours une exception ou une fonction qui entre dans une boucle infinie).

```
function error(message: string): never {  
    throw new Error(message);  
}
```

```
function infiniteLoop(): never {  
    while (true) {  
    }  
}
```

Using Types

"let" and "const"

`let` : Permet de déclarer des variables qui peuvent être réaffectées plus tard. Elle est généralement utilisée pour les variables dont les valeurs sont susceptibles de changer.

`const` : Permet de déclarer des variables qui ne peuvent pas être réaffectées après leur affectation initiale. C'est utilisé pour les variables qui doivent rester constantes tout au long de l'exécution.

Using Types

Arrow Functions

Les fonctions fléchées fournissent une syntaxe concise pour écrire des fonctions et sont particulièrement utiles pour les fonctions en ligne et les callbacks. Elles gèrent aussi différemment le this par rapport aux fonctions traditionnelles, le liant lexicalement à partir du contexte environnant.

```
const additionner = (a, b) => a + b;  
console.log(additionner(2, 3)); // Affiche : 5
```

Using Types

Default Function Parameters

Les paramètres de fonction par défaut permettent d'initialiser des paramètres nommés avec des valeurs par défaut si aucune valeur ou undefined est passé.

```
function saluer(nom, salutation = "Bonjour") {  
  console.log(`${salutation}, ${nom} !`);  
}  
  
saluer("Alice");    // Affiche : Bonjour, Alice!  
saluer("Bob", "Bienvenue"); // Affiche : Bienvenue, Bob!
```

The Spread Operator (...)

Using Types

The Spread Operator (...)

L'opérateur de décomposition permet à un itérable comme un tableau ou une chaîne d'être étendu dans des endroits où zéro ou plusieurs arguments (pour les appels de fonction) ou éléments (pour les littéraux de tableau) sont attendus. Il peut également être utilisé pour décomposer des objets dans un nouvel objet, en copiant leurs propriétés.

```
let nombres = [1, 2, 3];
console.log(Math.max(...nombres)); // Équivalent à Math.max(1, 2, 3)
```

```
let nums1 = [1, 2, 3];
let nums2 = [...nums1, 4, 5]; // [1, 2, 3, 4, 5]
```

Using Types

The Spread Operator (...)

L'opérateur de décomposition permet à un itérable comme un tableau ou une chaîne d'être étendu dans des endroits où zéro ou plusieurs arguments (pour les appels de fonction) ou éléments (pour les littéraux de tableau) sont attendus. Il peut également être utilisé pour décomposer des objets dans un nouvel objet, en copiant leurs propriétés.

Dans les littéraux d'objets (ES2018 et plus tard) :

```
let obj1 = { a: 1, b: 2 };
let obj2 = { ...obj1, c: 3 }; // { a: 1, b: 2, c: 3 }
```

Passage des éléments d'un tableau comme arguments à une fonction

```
function somme(x, y, z)
{
  return x + y + z;
}
```

```
const nombres = [1, 2, 3];
console.log(somme(...nombres)); // Affiche 6, équivalent à appeler somme(1, 2, 3)
```

Collecte d'arguments dans un tableau

Inversement, l'opérateur de décomposition peut être utilisé dans la définition de la fonction pour rassembler les arguments passés en un tableau, ce qui est utile pour les fonctions qui acceptent un nombre variable d'arguments.

```
function multiplication(...args) {
  return args.reduce((acc, curr) => acc * curr, 1);
}
```

```
console.log(multiplication(2, 3, 4)); // Affiche 24
console.log(multiplication(4, 5)); // Affiche 20
```

Combinaison avec d'autres paramètres

L'opérateur de décomposition peut être utilisé en combinaison avec d'autres paramètres réguliers. Il est important de noter que l'opérateur de décomposition doit être utilisé en dernier dans la liste des paramètres lorsqu'il est utilisé pour collecter le reste des arguments.

```
function affiche(nom, age, ...hobbies) {  
  console.log(`Nom: ${nom}, Age: ${age}, Hobbies: ${hobbies.join(", ")}`);  
}  
  
affiche("Alice", 30, "Escalade", "Lecture", "Voyage");  
// Affiche: Nom: Alice, Age: 30, Hobbies: Escalade, Lecture, Voyage
```

Classes Interfaces

Constructor Functions & The "this" Keyword

"private" and "public" Access Modifiers

Shorthand Initialization

"readonly" Properties

Inheritance

Overriding Properties & The "protected" Modifier

Getters & Setters

Static Methods & Properties

Abstract Classes

Singletons & Private Constructors

Classes - A Summary

A First Interface

Using Interfaces with Classes

Readonly Interface Properties

Extending Interfaces

Interfaces as Function Types

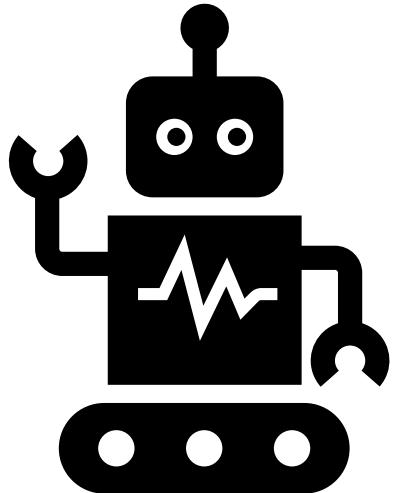
Optional Parameters & Properties

Compiling Interfaces to JavaScript

Advanced Types

Intersection Types

Premiers pas : Création d'une application simple



Composants et Templates (8 heures)

Professeur Titulaire : Millimono Sory (PhD, AI
Seacher)

Plan Du Cours



○ Approfondissement des composants : cycle de vie, @Input et @Output.

○ Travailler avec les templates : data binding, directives.

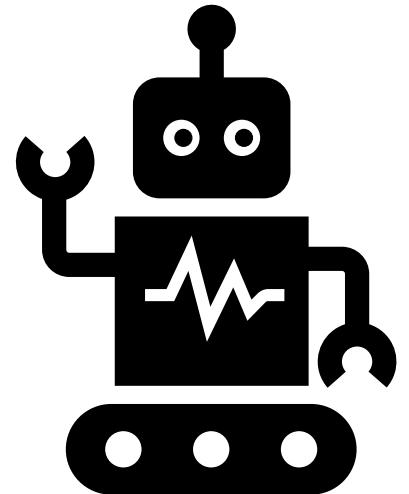
○ Création et gestion des événements.

○ Exercices pratiques.



Dans Angular, chaque composant a un cycle de vie bien défini géré par le framework, et la communication entre les composants est facilitée par des mécanismes spécifiques comme les décorateurs `@Input` et `@Output`.

Approfondissement des composants :
cycle de vie, @Input et @Output.



Modifier Premier Composant

The screenshot shows the Visual Studio Code interface with the following details:

- EXPLORER** sidebar: Shows the project structure under "MY-PROJECT".
- Code Editor**: The file "app.component.ts" is open, showing the following code:

```
1 import { Component } from '@angular/core';
2
3 @Component({
4   selector: 'app-root',
5   // templateUrl: './app.component.html',
6   template: ' <h1> Welcome Students {{etudiants }}</h1>',
7   // styleUrls: ['./app.component.css'
8 })
9 export class AppComponent {
10   title = 'my-project';
11   data= {
12     title:'saisir une data'
13   };
14   etudiants =['sory','aicha','aminata'];
15 }
16
17 }
```

The code defines an Angular component named AppComponent. It has a template with an h1 element displaying "Welcome Students" followed by a variable {{etudiants}} which is set to an array of student names: ['sory', 'aicha', 'aminata']. The component also has a title of 'my-project'.

Cycle de vie des composants Angular

Chaque composant Angular passe par une série d'étapes ou de "hooks" de cycle de vie qui permettent d'intervenir à différents moments de la création, de la mise à jour et de la destruction du composant. Les principaux hooks sont :

- **ngOnInit:** Se déclenche une fois que le composant est initialisé, après que les données liées ont été assignées. Idéal pour initialiser les données du composant.
- **ngOnChanges:** Appelé avant ngOnInit et chaque fois qu'une propriété liée via @Input change de valeur. Utile pour effectuer une logique en réponse à des changements de données spécifiques.
- **ngDoCheck:** Offre une vérification personnalisée des changements qui ne sont pas détectés par les mécanismes par défaut d'Angular. Exécuté après ngOnChanges et ngOnInit.
- **ngAfterContentInit:** Appelé après que le contenu projeté dans le composant a été initialisé.
- **ngAfterContentChecked:** Appelé après la vérification du contenu projeté dans le composant, et à chaque exécution de la détection de changement.
- **ngAfterViewInit:** Appelé après l'initialisation de la vue du composant et des vues enfant.
- **ngAfterViewChecked:** Appelé après la vérification de la vue du composant et des vues enfant.
- **ngOnDestroy:** Nettoie juste avant que Angular détruise le composant. Utilisé pour désabonner des observables et détacher des gestionnaires d'événements pour éviter les fuites de mémoire.

```
// import { Component } from '@angular/core';
import { Component, OnInit, OnChanges, DoCheck, AfterContentInit,
AfterContentChecked, AfterViewInit, AfterViewChecked, OnDestroy, SimpleChanges } from
'@angular/core';

@Component({
  selector: 'app-root',
  // templateUrl: './app.component.html',
  template: ' <h1> Welcome Students {{etudiants }}</h1>'
  // styleUrls: ['./app.component.css'
})
export class AppComponent implements
  OnInit, OnChanges, DoCheck, AfterContentInit,
  AfterContentChecked, AfterViewInit, AfterViewChecked, OnDestroy {

  title = 'my-project';
  data= {
    title:'saisir une data'
  };
  etudiants =['sory','aicha','aminata'];

  constructor() {
    console.log('AppComponent: Constructor');
  }

  ngOnChanges(changes: SimpleChanges) {
    // Called before ngOnInit and whenever one or more data-bound input properties
    // change.
    console.log('AppComponent: OnChanges', changes);
  }
}

ngOnInit() {
  // Called once the component is initialized.
  console.log('AppComponent: OnInit');
}

ngDoCheck() {
  // Called during every change detection run.
  console.log('AppComponent: DoCheck');
}

ngAfterContentInit() {
  // Called after content (ng-content) has been projected into view.
  console.log('AppComponent: AfterContentInit');
}

ngAfterContentChecked() {
  // Called after every check of the component's or directive's content.
  console.log('AppComponent: AfterContentChecked');
}

ngAfterViewInit() {
  // Called after the component's view (and child views) has been initialized.
  console.log('AppComponent: AfterViewInit');
}

ngAfterViewChecked() {
  // Called after every check of the component's view (and child views).
  console.log('AppComponent: AfterViewChecked');
}

ngOnDestroy() {
  // Called once the component is about to be destroyed.
  console.log('AppComponent: OnDestroy');
}
```

```
SelectStdent(StudentsName: String) {  
  console.log(`Vous avez Mr ${StudentsName}`);  
}  
}
```

```
ngOnInit() {  
  // Called once the component is initialized.  
  //console.log('AppComponent: OnInit');  
  console.table(this.etudiants);  
  this.SelectStdent(this.etudiants[0]);  
}  
}
```

Exercices :

- Modele Personne
- Tableau de Personnes typé de type Personne
- Charger le Liste des Personnes
- Afficher Une personne de la liste

Développer Votre Première Application avec Angular (2024)

The screenshot shows a dark-themed VS Code interface. On the left is the Explorer sidebar with the project structure:

- NG-POKEMON-APP
- .angular
- .vscode
- node_modules
- src
 - app
 - app-routing.module.ts
 - app.component.ts
 - app.module.ts
 - pokemon.ts
 - assets
 - environments
 - favicon.ico
 - index.html
 - main.ts
 - polyfills.ts
 - styles.css
 - .browserslistrc
 - .gitignore
 - angular.json
 - package-lock.json

On the right is the Editor pane showing the file `pokemon.ts` with the following code:

```
export class Pokemon {  
  id: number;  
  hp: number;  
  cp: number;  
  name: string;  
  picture: string;  
  types: Array<string>;  
  created: Date;  
}
```

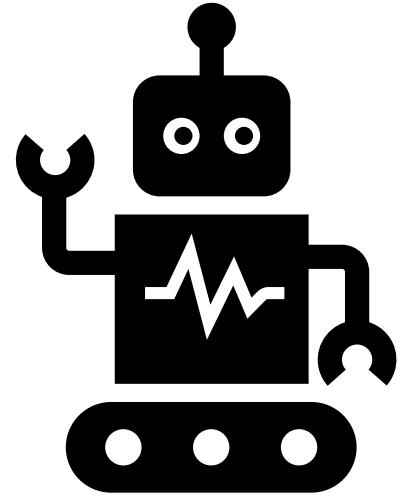
A tooltip at the bottom center of the editor pane says: "Et le deuxième fichier dont on va avoir besoin."

At the bottom of the interface, there are status bar items: L 9, col 2 Espaces : 4 UTF-8 LF {} TypeScript Atom

```
export class Personne {  
    id: number | undefined;  
    firstName: string | undefined;  
    lastName: string | undefined;  
    age: number | undefined;  
    picture: string | undefined;  
    email: string | undefined;  
    phone: string | undefined;  
    address: string | undefined;  
    created: Date | undefined;  
}
```

```
export const personnes : Personne[] =  
[  
    {  
        id: 447,  
        firstName: 'Alice',  
        lastName: 'Jones',  
        age: 35,  
        picture:  
            'https://th.bing.com/th/id/R.53bb1512421aa9534427a4b0e62f  
5aa1?rik=H2BmPf2jcnfhPg&riu=http%3a%2f%2fimagesolutions  
.ca%2fwp-content%2fuploads%2f2016%2f01%2fDSC02624-  
Edit.jpg&ehk=NTIVsJxrfayk9LEHeISQ8%2f0ooZ6dFVw8PMFJH%  
2fOcv1Q%3d&risl=&pid=ImgRaw&r=0',  
        email: 'alice.jones@test.com',  
        phone: '555-7890',  
        address: '456 Elm St',  
        created: new Date()  
    }, ]
```

Building our templates :



The screenshot shows the Visual Studio Code interface with the following details:

- File Bar:** File, Edit, Selection, View, Go, Run, ...
- Search Bar:** my-project
- Icons:** Explorer, Search, Problems (7), Find, Split, Terminal, Help.
- Left Sidebar (Explorer):** MY-PROJECT folder containing .angular, .vscode, node_modules, src (with app, app-routing.module.ts, app.component.css, app.component.html, app.component.spec.ts, app.component.ts, app.module.ts, list_Personne.ts, personne.ts), assets, favicon.ico, index.html, main.ts, styles.css, .editorconfig, .gitignore, angular.json, package-lock.json, package.json, README.md, OUTLINE, and TIMELINE.
- Code Editor:** The active file is app.component.ts (M). The code defines the AppComponent class, which implements OnInit. It includes imports for Component, OnInit, OnChanges, DoCheck, AfterContentInit, AfterContentChecked, AfterViewInit, AfterViewChecked, and OnDestroy from '@angular/core' and Personne from './personne'. The AppComponent selector is 'app-root', and its template URL is './app.component.html'. The component has a title property set to 'my-project' and a data property with a title of 'saisir une data'. It also has an etudiants array of Personne type initialized to personnes. The constructor logs 'AppComponent: Constructor'. The SelectStudent method logs a message with the student's last name.
- Bottom Status Bar:** master* (green), 0 △ 0, 0, Ln 9, Col 1, Spaces: 2, UTF-8, CRLF, TypeScript, Prettier, 22:22.

```
// import { Component } from '@angular/core';
import { Component, OnInit, OnChanges, DoCheck, AfterContentInit, AfterContentChecked, AfterViewInit, AfterViewChecked, OnDestroy } from '@angular/core';
import { personnes } from './list_Personne';
import { Personne } from './personne';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  // styleUrls: ['./app.component.css'
})

export class AppComponent implements OnInit {
  //, OnChanges, DoCheck, AfterContentInit, AfterContentChecked, AfterViewInit, AfterViewChecked, OnDestroy
  {

    title = 'my-project';
    data= {
      title:'saisir une data'
    };
    etudiants : Personne[] = personnes;

  constructor() {
    console.log('AppComponent: Constructor');
  }

  SelectStudent(Student: Personne) {
    // Called before ngOnInit and whenever one or more data-bound input properties change.
    console.log(`Vous avez Mr ${Student.lastName}`);
  }
}
```

The screenshot shows the Visual Studio Code interface with the following details:

- EXPLORER**: Shows the project structure under "MY-PROJECT".
 - src folder contains:
 - app folder contains:
 - app-routing.module.ts
 - app.component.css (marked M)
 - app.component.html (marked M)
 - app.component.spec.ts
 - app.component.ts (marked M)
 - app.module.ts
 - list_Personne.ts (marked U)
 - personne.ts (marked U)
 - assets
 - favicon.ico
 - index.html
 - main.ts
 - styles.css
 - .editorconfig
 - .gitignore
 - angular.json
 - package-lock.json
 - package.json
 - README.md
 - OUTLINE and TIMELINE sections are also visible.

EDITOR: The active file is app.component.html, showing the following code:

```
<h1>Welcome Students</h1>
<p> {{etudiants[0].lastName }}</p>
<p> {{etudiants[1].lastName }}</p>
<p> {{etudiants[2].lastName }}</p>
<p> {{etudiants[3].lastName }}</p>
<p> {{etudiants[4].lastName }}</p>
<p> {{etudiants[5].lastName }}</p>
<p> {{etudiants[6].lastName }}</p>
<p> {{etudiants[7].lastName }}</p>
<p> {{etudiants[8].lastName }}</p>
<p> {{etudiants[9].lastName }}</p>
<p> {{etudiants[10].lastName }}</p>
<p> {{etudiants[11].lastName }}</p>
```

STATUS BAR: Shows tabs for app.component.html, app.component.ts, personne.ts, list_Personne.ts, and tsconfig.json, along with icons for search, file operations, and other settings.

A screenshot of the Visual Studio Code (VS Code) interface. The top bar includes the standard menu (File, Edit, Selection, View, Go, Run, ...), a back/forward navigation bar, a search bar containing "my-project", and window control buttons. The left sidebar features icons for Explorer, Search, Problems (with 7 notifications), and Outline. The Explorer pane shows a project structure under "MY-PROJECT": .angular, .vscode, node_modules, src (containing app, app-routing.module.ts, app.component.css, app.component.html, app.component.spec.ts, app.component.ts, app.module.ts, list_Personne.ts, personne.ts), assets, favicon.ico, index.html, main.ts, styles.css, .editorconfig, .gitignore, angular.json, package-lock.json, package.json, README.md, and OUTLINE. The main editor pane displays the content of "app.component.html". The code shown is:

```
<h1>Welcome Students</h1>
<ul>
  <li *ngFor="let student of etudiants">
    {{ student.lastName }}, {{ student.firstName }} - Age: {{ student.age }}
  </li>
</ul>
```

The code editor has syntax highlighting for HTML and TypeScript. The status bar at the bottom right shows "1339 / 1339" and "100%".

La syntaxe de liaison des données

Propriétés	Code	Explications
Propriété d'élément	<code></code>	On utilise les crochets pour lier directement la source de l'image à la propriété du composant <i>someImageUrl</i> .
Propriété d'attribut	<code><label [attr.for]= »someLabelId «>...</label></code>	On lie l'attribut <i>for</i> de l'élément <i>label</i> avec la propriété de notre composant <i>someLabelId</i> .
Propriété de la classe	<code><div [class.special]= »isSpecial «>Special</div></code>	Fonctionnement similaire, pour attribuer ou non la classe <i>special</i> à l'élément <i>div</i> .
Propriété de style	<code><button [style.color]= »isSpecial?'red':'green' «>Special</button></code>	On peut également définir un style pour nos éléments de manière dynamique : ici on définit la couleur de notre bouton en fonction de la propriété <i>isSpecial</i> , soit rouge, soit vert. (C'est un opérateur ternaire que l'on utilise comme expression).

Interpolation {{...}}

Utilisée principalement pour insérer des valeurs de texte :

```
<p>{{ title }}</p>
```

Property Binding [...]

Lie une propriété d'un élément DOM à une propriété du composant :

```
<img [src]="imageUrl">
```

Event Binding (...)

Exécute une expression ou appelle une méthode du composant en réponse à un événement du DOM :

```
<button (click)="handleClick()">Click me</button>
```

Two-way Binding (...)

Combine le property et event binding pour une synchronisation bidirectionnelle :

```
<input [(ngModel)]="username">
```

Attribute Binding [attr...]

Lie une propriété d'un élément DOM à la valeur d'un attribut HTML :

```
<button [attr.aria-label]="help">Help</button>
```

Class Binding [class...]

Ajoute ou supprime une classe CSS de l'élément en fonction d'une expression booléenne :

```
<div [class.style_css]="isSpecial">Special content</div>
```

Style Binding [style...]

Lie une propriété de style d'un élément DOM à une expression :

```
<div [style.font-weight]="isBold ? 'bold' : 'normal'">Bold text</div>
```

Liaison d'attributs Angular spéciaux comme ngClass et ngStyle

ngClass et ngStyle permettent de lier dynamiquement plusieurs classes ou styles :

```
<div [ngClass]="{{'class1': condition1, 'class2': condition2}}></div>
```

```
<div [ngStyle]="{{'color': color, 'font-size': fontSize + 'px'}}></div>
```

Gérer les interactions de l'utilisateur

```
<button (click)="handleClick(var)">Click me</button>
```

Intercepter tous les événements du DOM

```
-> app > app.component.html > input
Go to component
1  <!--
2   <h1>Welcome Students</h1>
3   <p> {{etudiants[0].lastName }}</p>
4   <p> {{etudiants[1].lastName }}</p>
5   <p> {{etudiants[2].lastName }}</p>
6   <p> {{etudiants[3].lastName }}</p>
7   <p> {{etudiants[4].lastName }}</p>
8   <p> {{etudiants[5].lastName }}</p>
9   <p> {{etudiants[6].lastName }}</p>
10  <p> {{etudiants[7].lastName }}</p>
11  <p> {{etudiants[8].lastName }}</p>
12  <p> {{etudiants[9].lastName }}</p>
13  <p> {{etudiants[10].lastName }}</p>
14  <p> {{etudiants[11].lastName }}</p> -->
15
16 <input type="number" (click)="OnpersnneClicked($event)"/>
17
```

```
OnpersnneClicked(event : MouseEvent)
{
  const index : number = +(event.target as HTMLInputElement).value;
  console.log(this.etudiants[index].lastName)
}
```

Les variables référencées dans le template

```
<p> {{etudiants[2].lastName}}</p>
<p> {{etudiants[3].lastName}}</p>
<p> {{etudiants[4].lastName}}</p>
<p> {{etudiants[5].lastName}}</p>
<p> {{etudiants[6].lastName}}</p>
<p> {{etudiants[7].lastName}}</p>
<p> {{etudiants[8].lastName}}</p>
<p> {{etudiants[9].lastName}}</p>
<p> {{etudiants[10].lastName}}</p>
<p> {{etudiants[11].lastName}}</p> -->

<input #input type="text" (keyup)="0"/>

<p>
|   {{input.value}}
</p>
```

Rôle de (keyup)

L'événement keyup est utilisé pour effectuer des actions immédiatement après que l'utilisateur a tapé dans un champ de saisie, par exemple pour :

- Valider la saisie de l'utilisateur dès qu'il a fini de taper un caractère.
- Fournir des suggestions automatiques ou compléter automatiquement le texte.
- Mettre à jour d'autres parties de l'interface utilisateur en réaction à la saisie de l'utilisateur.

- #input est une variable de template qui fait référence à l'élément <input>.

Les variables de template dans Angular vous permettent de référencer des éléments DOM ou des instances de directives et de composants directement dans votre template HTML. En utilisant le préfixe # suivi d'un nom, vous créez une variable de template qui peut ensuite être utilisée n'importe où dans le template de ce composant pour accéder à cet élément ou à cette instance.

Créer un flux de données birectionnel

```
<p> {{etudiants[6].lastName }}</p>
<p> {{etudiants[7].lastName }}</p>
<p> {{etudiants[8].lastName }}</p>
<p> {{etudiants[9].lastName }}</p>
<p> {{etudiants[10].lastName }}</p>
<p> {{etudiants[11].lastName }}</p> -->

<input #input type="number" (keyup)="OnpersonneClicked(input.value)"/>

<p>
|   {{personeselected?.lastName}}
</p>
```

```
OnpersonneClicked(personneid : string)
{
    const id = +personneid;
    console.log(this.etudiants[id].lastName, " ", this.etudiants[id].firstName)
}
```

Créer un flux de données birectionnel

```
<p> {{etudiants[6].lastName }}</p>
<p> {{etudiants[7].lastName }}</p>
<p> {{etudiants[8].lastName }}</p>
<p> {{etudiants[9].lastName }}</p>
<p> {{etudiants[10].lastName }}</p>
<p> {{etudiants[11].lastName }}</p> -->

<input #input type="number" (keyup)="OnpersnneClicked(input.value)"/>

<p>
  {{personeselected?.lastName}}
</p>
```

```
OnpersnneClicked(personneid : string)
{
  const personne  : Personne | undefined =  this.etudiants.find(personne => personne.id == +personneid )

  if(personne)
  {
    console.log(personne.lastName, " ",personne.firstName);
    this.personeselected = personne;
  }
  else{
    this.personeselected = personne;
  }
}
```

DéTECTER L'APPUI SUR LA TOUCHE Entrée

```
15 <input #input type="number" (keyup.enter)="OnpersonClicked(input.value)"/>
16
17
18 <p>
19   {{personeselected?.lastName}}  {{personeselected?.firstName}}
20 </p>
21
22 <app-personnes (PersonneSelected)="OnPersonneSelected($event)" [personne] = "etudiants[0]"></app-personnes>
23 <app-personnes [personne] = "etudiants[1]"></app-personnes>
24 <app-personnes [personne] = "etudiants[2]"></app-personnes>
25
26
```

Pseudo évènement d'angular

Conditionner un affichage avec NgIf

```
<p> {{etudiants[8].lastName }}</p>
<p> {{etudiants[9].lastName }}</p>
<p> {{etudiants[10].lastName }}</p>
<p> {{etudiants[11].lastName }}</p> -->

<input #input type="number" (keyup.enter)="OnpersonneClicked(input.value)"/>

<p *ngIf="personeselected">
  Vous avez selectionné {{personeselected?.lastName}} {{personeselected?.firstName}}
</p>
```

```
<p> {{etudiants[8].lastName }}</p>
<p> {{etudiants[9].lastName }}</p>
<p> {{etudiants[10].lastName }}</p>
<p> {{etudiants[11].lastName }}</p> -->

<input #input type="number" (keyup.enter)="OnpersonneClicked(input.value)"/>

<p *ngIf="personeselected">
  Vous avez selectionné {{personeselected?.lastName}} {{personeselected?.firstName}}
</p>

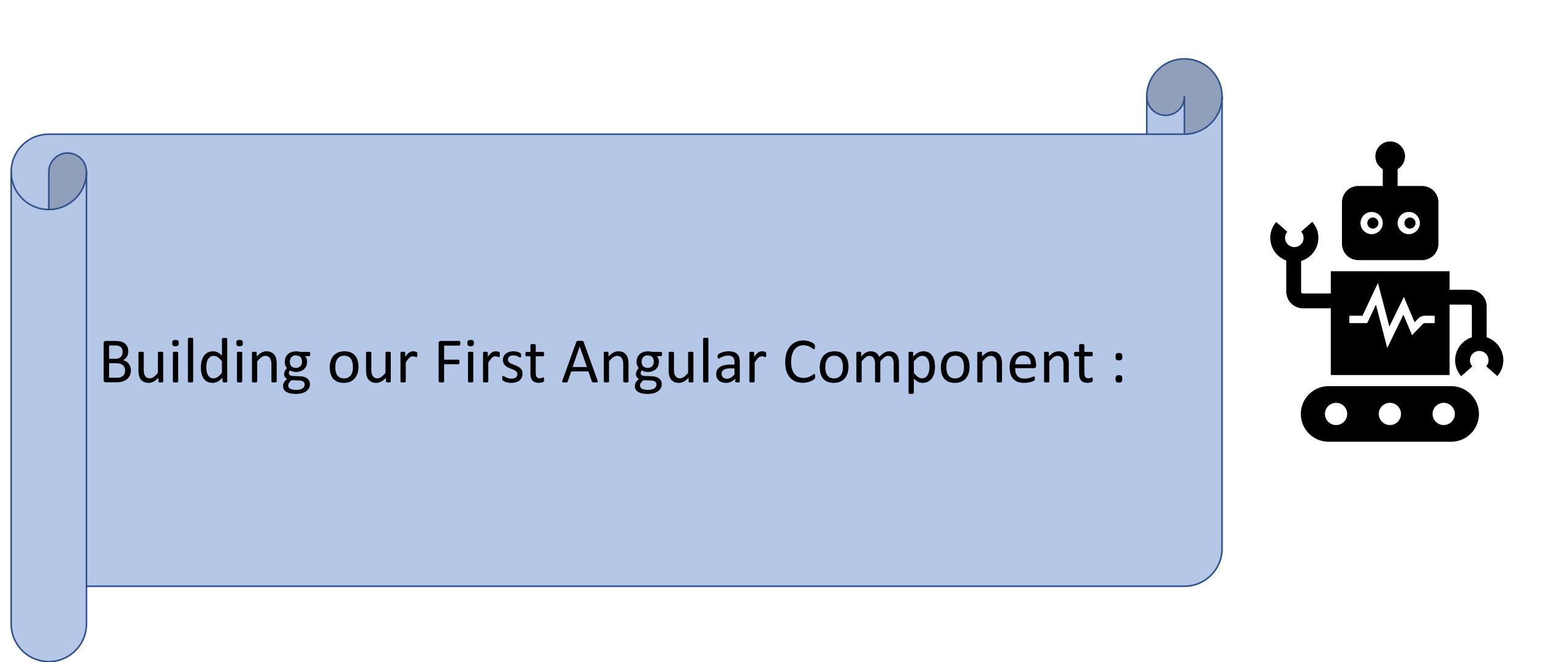
<p *ngIf="!personeselected">
  Aucune Personne Trouvée
</p>
```

Afficher une liste avec NgFor

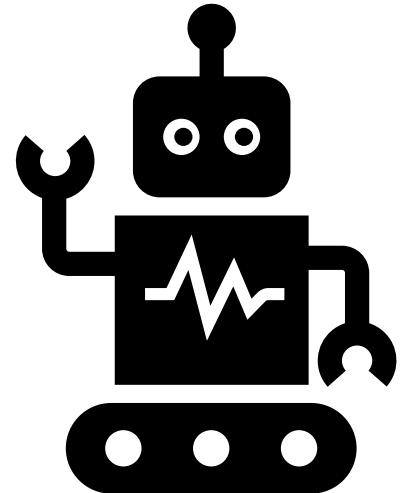
```
1  <h1>Welcome Students</h1>
2  <p *ngFor="let personne of etudiants">
3      {{personne.lastName }}
4
5  </p>
6
7
8
9  <!-- <p> {{etudiants[0].lastName }}</p>
10 <p> {{etudiants[1].lastName }}</p>
11 <p> {{etudiants[2].lastName }}</p>
12 <p> {{etudiants[3].lastName }}</p>
13 <p> {{etudiants[4].lastName }}</p>
14 <p> {{etudiants[5].lastName }}</p>
15 <p> {{etudiants[6].lastName }}</p>
16 <p> {{etudiants[7].lastName }}</p>
17 <p> {{etudiants[8].lastName }}</p>
18 <p> {{etudiants[9].lastName }}</p>
19 <p> {{etudiants[10].lastName }}</p>
20 <p> {{etudiants[11].lastName }}</p> -->
21
```

Materialize

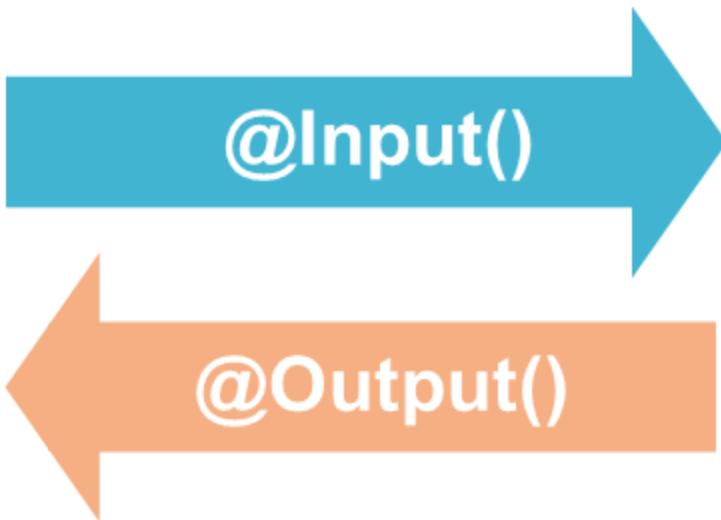
w3.css



Building our First Angular Component :



Parent
component



Child
component

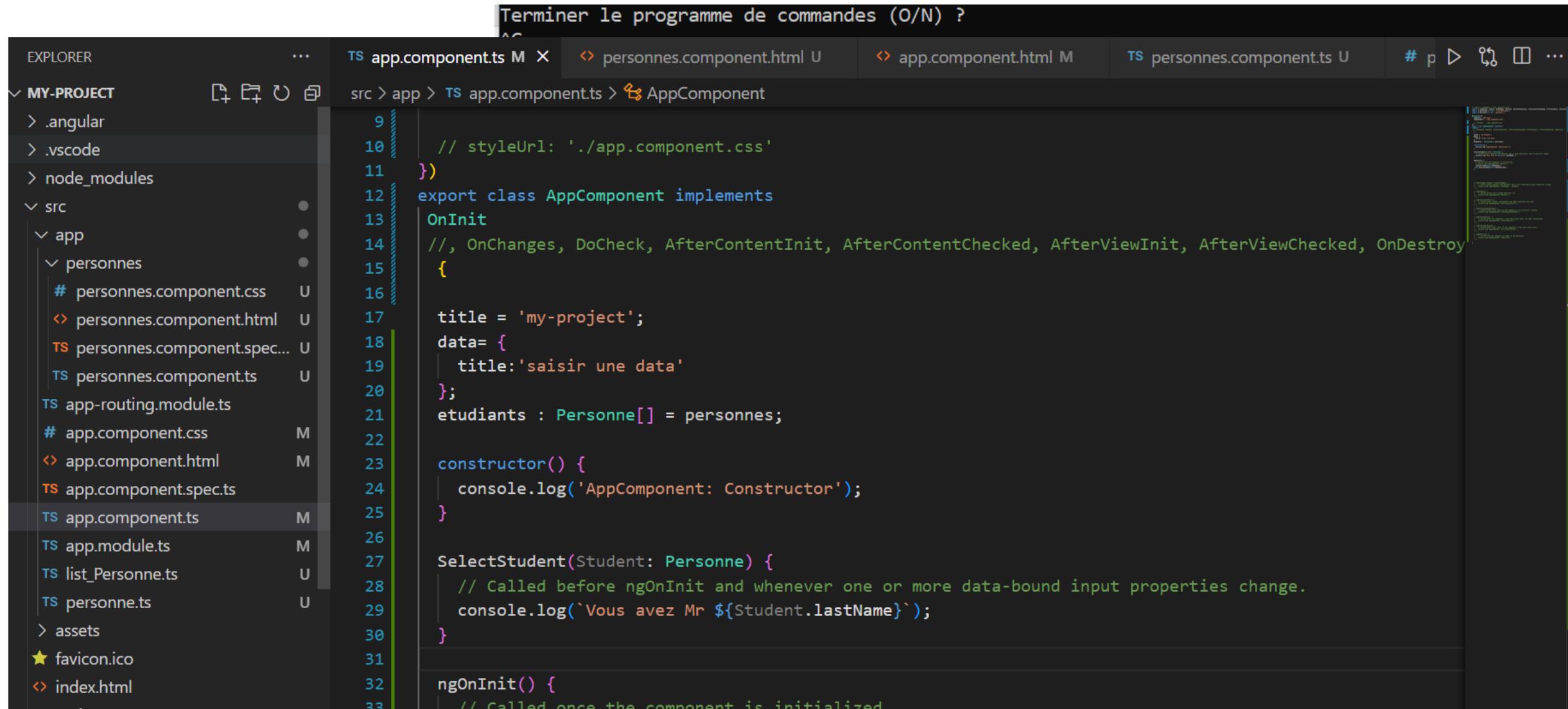
Building our First Angular Component :

ng generate component personnes

```
Terminer le programme de commandes (O/N) ?  
^C  
C:\FILES\PHD\COURS_UNIVERSITE\Angular\my-project>ng generate component personnes  
CREATE src/app/personnes/personnes.component.html (25 bytes)  
CREATE src/app/personnes/personnes.component.spec.ts (645 bytes)  
CREATE src/app/personnes/personnes.component.ts (221 bytes)  
CREATE src/app/personnes/personnes.component.css (0 bytes)  
UPDATE src/app/app.module.ts (507 bytes)  
  
C:\FILES\PHD\COURS_UNIVERSITE\Angular\my-project>
```

Building our First Angular Component :

ng generate component personnes



The screenshot shows the Visual Studio Code interface with the following details:

- EXPLORER** sidebar: Shows the project structure under "MY-PROJECT".
- COMMANDS** bar: Displays the command "ng generate component personnes".
- TERMINAL** tab: Shows the message "Terminer le programme de commandes (O/N) ?" (Finish the command program (Y/N) ?).
- CODE EDITOR**: The active file is `app.component.ts`. The code is as follows:

```
Terminer le programme de commandes (O/N) ?
...
EXPLORER ...
TS app.component.ts M X  ⟳ personnes.component.html U  ⟳ app.component.html M  TS personnes.component.ts U  # p ▶ ⌂ ...
MY-PROJECT ...
src > app > TS app.component.ts > AppComponen...
  9
  10   // styleUrls: ['./app.component.css'
  11 }
  12 export class AppComponent implements
  13 OnInit
  14 //, OnChanges, DoCheck, AfterContentInit, AfterContentChecked, AfterViewInit, AfterViewChecked, OnDestroy
  15 {
  16
  17   title = 'my-project';
  18   data= {
  19     title:'saisir une data'
  20   };
  21   etudiants : Personne[] = personnes;
  22
  23   constructor() {
  24     console.log('AppComponent: Constructor');
  25   }
  26
  27   SelectStudent(Student: Personne) {
  28     // Called before ngOnInit and whenever one or more data-bound input properties change.
  29     console.log(`Vous avez Mr ${Student.lastName}`);
  30   }
  31
  32   ngOnInit() {
  33     // Called once the component is initialized
  34   }
  35 }
```

Building our First Angular Component :

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

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

  ngOnInit(): void {
  }

}
```

```
<div class="Personnes">
  <div class="personne-card">
    <div class="personne-Name">
      Alice Jones
    </div>
    
    <div class="personne-description">
      A detailed walk-through of the most important part of
      Angular - the Core and Common modules.
    </div>
  </div>
</div>
```

Building our First Angular Component :

```
.Personnes {  
  display: flex;  
  flex-wrap: wrap;  
  justify-content: center;  
}  
  
.personne-card {  
  border: 1px solid #ddd;  
  border-radius: 4px;  
  padding: 16px;  
  margin: 16px;  
  width: 300px;  
}
```

```
.personne-Name {  
  font-size: 1.2em;  
  margin-bottom: 8px;  
}  
  
.personne-description {  
  font-size: 0.9em;  
}  
  
img {  
  max-width: 100%;  
  height: auto;  
  display: block;  
  margin-bottom: 8px;  
}
```

Building our First Angular Component :

The screenshot shows the Visual Studio Code interface. The left sidebar displays the file structure of a project named "MY-PROJECT". The "src" folder contains "app", "assets", "main.ts", "styles.css", and ".editorconfig". The "app" folder contains "personnes", "app-routing.module.ts", "app.component.css", "app.component.html", "app.component.spec.ts", "app.component.ts", "app.module.ts", "list_Personne.ts", and "personne.ts". The "personnes" folder contains "personnes.component.css", "personnes.component.html", "personnes.component.spec.ts", and "personnes.component.ts". The "app.component.html" file is open in the editor, showing the following code:

```
<h1>Welcome Students</h1>
<p> {{etudiants[0].lastName }}</p>
<p> {{etudiants[1].lastName }}</p>
<p> {{etudiants[2].lastName }}</p>
<p> {{etudiants[3].lastName }}</p>
<p> {{etudiants[4].lastName }}</p>
<p> {{etudiants[5].lastName }}</p>
<p> {{etudiants[6].lastName }}</p>
<p> {{etudiants[7].lastName }}</p>
<p> {{etudiants[8].lastName }}</p>
<p> {{etudiants[9].lastName }}</p>
<p> {{etudiants[10].lastName }}</p>
<p> {{etudiants[11].lastName }}</p>
<app-personnes></app-personnes>
```

```
C:\FILES\PHD\COURS_UNIVERSITE\Angular\my-project>ng serve
? Port 4200 is already in use.
Would you like to use a different port? Yes
Initial chunk files | Names | Raw size
polyfills.js | polyfills | 83.60 kB |
main.js | main | 10.73 kB |
styles.css | styles | 95 bytes |
| Initial total | 94.42 kB

Application bundle generation complete. [3.427 seconds]

Watch mode enabled. Watching for file changes...
Re-optimizing dependencies because vite config has changed
  Local: http://localhost:61984/
  press h + enter to show help

No output file changes.

Application bundle generation complete. [0.256 seconds]
```

@Input et @Output

Pour utiliser les décorateurs **@Input** et **@Output** dans Angular pour la communication entre composants, nous devons les déclarer dans la classe de notre composant enfant et les lier ensuite dans le template de notre composant parent.

```
<app-personnes [nom] = "etudiants[0].lastName"></app-personnes>
<app-personnes></app-personnes>
<app-personnes></app-personnes>
```

```
export class PersonnesComponent implements OnInit {
  @Input()
  nom : String | undefined;
  ngOnInit(): void {
  }
}
```

```
<div class="personne-Name">
  {{ nom }}
</div>
```

```
app > personnes > TS personnes.component.ts > PersonnesComponent
  import { Component, Input, OnInit } from '@angular/core';
  import { Personne } from '../personne';

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

    @Input()
    personne : Personne | undefined;

    ngOnInit(): void {
    }
  }
```

```
src > app > personnes > personnes.component.html > div.Personnes > div.personne-card
  Go to component
  1  <div class="Personnes">
  2    <div class="personne-card">
  3      <div class="personne-Name">
  4        {{ personne?.lastName }} {{ personne?.firstName }}
  5      </div>
  6      <img width="300" alt="Angular Logo"
  7        [src]="personne?.picture">
  8      <div class="personne-description">
  9        {{personne?.address}}
 10      </div>
 11    </div>
 12  </div>
 13
```

```
<app-personnes [personne] = "etudiants[0]"></app-personnes>
<app-personnes [personne] = "etudiants[1]"></app-personnes>
<app-personnes [personne] = "etudiants[2]"></app-personnes>
```

OUTPUT

Go to component

```
<div class="Personnes">
  <div class="personne-card">
    <div class="personne-Name">
      {{ personne?.lastName }} {{ personne?.firstName }}
    </div>
    <img width="300" alt="Angular Logo"
      [src]="personne?.picture">
    <div class="personne-description">
      {{personne?.address}}
    </div>
    <button (click)="OnpersonneViewed()">Details sur la personne</button>
  </div>
</div>
```

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

@Component({
  selector: 'app-personnes',
  templateUrl: './personnes.component.html',
  styleUrls: ['./personnes.component.css'
})
export class PersonnesComponent implements OnInit {

  @Input()
  personne : Personne | undefined;

  ngOnInit(): void {
  }

  OnpersonneViewed()
  {
    console.log("btn clické");
  }
}
```

```
<app-personnes (click)="OncardClicked()" [personne] = "etudiants[0]"></app-personnes>
<app-personnes [personne] = "etudiants[1]"></app-personnes>
<app-personnes [personne] = "etudiants[2]"></app-personnes>
```

```
export class AppComponent implements
SelectStudent(Student: Personne) {
  // Called before ngOnInit and whenever one or more data-bound input properties change.
  console.log(`Vous avez Mr ${Student.lastName}`);
}

ngOnInit() {
  // Called once the component is initialized.
  //console.log('AppComponent: OnInit');
  console.table(this.etudiants);
  this.SelectStudent(this.etudiants[0]);
}

OncardClicked()
{
  console.log('click de card');

}
```

```
<app-personnes (PersonneSelected)="OnPersonneSelected()" [personne] = "etudiants[0]"></app-personnes>
<app-personnes [personne] = "etudiants[1]"></app-personnes>
<app-personnes [personne] = "etudiants[2]"></app-personnes>
```

```
export class AppComponent implements
  SelectStudent(Student: Personne) {
  // Called before ngOnInit and whenever one or more
  // inputs change
  console.log(`Vous avez Mr ${Student.lastName}`);
}

ngOnInit() {
  // Called once the component is initialized.
  //console.log('AppComponent: OnInit');
  console.table(this.etudiants);
  this.SelectStudent(this.etudiants[0]);
}

OnPersonneSelected()
{
  console.log('click de card');
}
```

```
export class PersonnesComponent implements OnInit {
  @Input()
  personne : Personne | undefined;

  PersonneSelected = new EventEmitter<Personne>();

  ngOnInit(): void {
  }
}
```

```
)  
export class PersonnesComponent implements OnInit {  
  
    @Input()  
    personne : Personne | undefined;  
  
    @Output()  
    PersonneSelected = new EventEmitter<Personne>();  
  
    ngOnInit(): void {  
    }  
  
    OnpersonneViewed()  
    {  
        console.log("btn clické");  
        this.PersonneSelected.emit(this.personne);  
    }  
}
```

```
<app-personnes (PersonneSelected)="OnPersonneSelected($event)" [personne] = "etudiants[0]"></app-personnes>  
<app-personnes [personne] = "etudiants[1]"></app-personnes>  
<app-personnes [personne] = "etudiants[2]"></app-personnes>
```

```
export class AppComponent implements  
  OnInit {  
  data= `  
  `;  
  etudiants : Personne[] = personnes;  
  
  constructor() {  
    console.log('AppComponent: Constructor');  
  }  
  
  SelectStudent(Student: Personne) {  
    // Called before ngOnInit and whenever one or more  
    // students are selected.  
    console.log(`Vous avez Mr ${Student.lastName}`);  
  }  
  
  ngOnInit() {  
    // Called once the component is initialized.  
    //console.log('AppComponent: OnInit');  
    console.table(this.etudiants);  
    this.SelectStudent(this.etudiants[0]);  
  }  
  
  OnPersonneSelected(personne:Personne)  
  {  
    console.log('click de card' , personne);  
  }  
}
```

Angular 17 Control Flow Syntax:

The Angular 17 @for syntax

```
31
32 <app-personnes (PersonneSelected)="OnPersonneSelected($event)" [personne] = "etudiants[0]"></app-personnes>
33 <app-personnes [personne] = "etudiants[1]"></app-personnes>
34 <app-personnes [personne] = "etudiants[2]"></app-personnes>
35
```

```
</p> -->
@for (per of etudiants; track $per?.id)
{
<app-personnes (PersonneSelected)="OnPersonneSelected($event)" [personne] = "per"></app-personnes>
}

```

Angular 17 Control Flow Syntax:

Angular @for @empty, \$index and other extra options

```
@for (per of etudiants; track $per?.id)
{
<app-personnes (PersonneSelected)="OnPersonneSelected($event)" [personne] = "per"></app-personnes>
}
@empty {
| <h1>No Student found</h1>
}
```

Angular 17 Control Flow Syntax:

Angular @for @empty, \$index and other extra options

```
</p>
  >
  @for (per of etudiants; track $per?.id;let index = $index)
  {
    <app-personnes (PersonneSelected)="OnPersonneSelected($event)" [personne] = "per"
      | [index]="index"></app-personnes>
    }
  @empty {
    <h1>No Student found</h1>
  }
  >
  >
```

```
Go to component
<div class="Personnes">
  <div class="personne-card">
    <div class="personne-Name">
      | {{index}} {{ personne?.lastName }} {{ personne?.firstName }}
    </div>
    <img width="300" alt="Angular Logo"
      [src]="personne?.picture">
    <div class="personne-description">
      | {{personne?.address}}
    </div>
    <button (click)="OnpersonneViewed()">Details sur la personne</button>
  </div>
</div>
```

```
)>
  export class PersonnesComponent implements OnInit {
    @Input()
    personne : Personne | undefined;

    @Input()
    index : Number | undefined;

    @Output()
    PersonneSelected = new EventEmitter<Personne>();

    ngOnInit(): void {
    }

    OnpersonneViewed(){
      console.log("btn clické");
      this.PersonneSelected.emit(this.personne);
    }
  }
```

Angular 17 Control Flow Syntax:

Angular @for @empty, \$index and other extra options

```
31 @for (per of etudiants; track $per?.id; let index = $index; let count = $count)
32 {
33     <h1> {{count }} </h1>
34     <app-personnes (PersonneSelected)="OnPersonneSelected($event)" [personne] = "per"
35         [index]="index"></app-personnes>
36 }
37 }
38 @empty {
39     <h1>No Student found</h1>
40 }
41
42
```

Angular 17 Control Flow Syntax:

Angular @for @empty, \$index and other extra options

```
1  /* You can add global styles to this file, and also import other style files */
2  app-personnes.is-first
3  {
4      font-weight: bold;
5      color: blue;
6  }
7
8
9  app-personnes.is-last
10 {
11     font-weight: bold;
12     color: red;
13 }
```

```
@for (per of etudiants; track $per?.id; let index = $index; let count = $count;
let first = $first; let last = $last)
{
    <h1> {{count }} </h1>
    <app-personnes
        (PersonneSelected)="OnPersonneSelected($event)"
        [personne] = "per"
        [index]="index"
        [class.is-first]="first"
        [class.is-last]="last"

    />
}
@empty {
    <h1>No Student found</h1>
}
```

Angular 17 Control Flow Syntax:

Angular @for @empty, \$index and other extra options

```
1  @for (per of etudiants; track $per?.id; let index = $index; let count = $count;
2  let first = $first; let last = $last; let even = $even; let odd = $odd)
3  {
4      <app-personnes
5          (PersonneSelected)="OnPersonneSelected($event)"
6          [personne] = "per"
7          [index]="index"
8          [class.is-first]="first"
9          [class.is-last]="last"
10         [class.is-even]="even"
11         [class.is-odd]="odd"
12     />
13     <!-- Inside your PersonnesComponent template -->
14     <!-- Inside your PersonnesComponent template -->
15
16
17 }
18 @empty {
19     <h1>No Student found</h1>
20 }
```

```
app-personnes.is-even
{
    .personne-card {background-color : red;}
}

app-personnes.is-odd
{
    .personne-card {background-color : black;}
}
```

```
Go to component
<div class="Personnes">
    <div class="personne-card">
        <div class="personne-Name">
            {{index}} {{ personne?.lastName }} {{ personne?.firstName }}
        </div>
        <img width="300" alt="Angular Logo"
            [src]="personne?.picture">
        <div class="personne-description">
            {{personne?.address}}
        </div>
        <button (click)="OnpersonneViewed()">Details sur la personne</button>
    </div>
</div>
```

Angular 17 Control Flow Syntax:

Angular @for tracking functions

```
    @for (per of etudiants; track trackPersonne; let index = $index; let count = $count;
let first = $first; let last = $last; let even = $even; let odd = $odd)
{
  <app-personnes
    (PersonneSelected)="OnPersonneSelected($event)"
    [personne] = "per"
    [index]="index"
    [class.is-first]="first"
    [class.is-last]="last"
    [class.is-even]="even"
    [class.is-odd]="odd"
  />
  <!-- Inside your PersonnesComponent template -->
<!-- Inside your PersonnesComponent template -->

}

@empty {
  <h1>No Student found</h1>
}
```

```
trackPersonne(index :Number,per: Personne)
{
  return per.id;
}
```

Angular 17 Control Flow Syntax:

Angular @for tracking functions

```
| </p> -->
@for (per of etudiants; track $index ;let count = $count;
let first = $first; let last = $last;let even = $even; let odd = $odd)
{
  <app-personnes
    (PersonneSelected)="OnPersonneSelected($event)"
    [personne] = "per"
    [index]="$index"
    [class.is-first]="first"
    [class.is-last]="last"
    [class.is-even]="even"
    [class.is-odd]="odd"
  />
  <!-- Inside your PersonnesComponent template -->
<!-- Inside your PersonnesComponent template -->
```

Angular Control Flow Syntax:

The Angular ngFor Core Directive

```
<app-personnes *ngFor="let per of etudiants; index as i; let first = first; let last = last;  
even as ev ; odd as odd;"  
(PersonneSelected)="OnPersonneSelected($event)" [personne] = "per"  
[index] = "i +1" [class.is-first]="first" [class.is-last]="last"  
[class.is-even]="ev" [class.is-odd]="odd"
```

Run this command to upgrade your project:

```
ng generate angular/core:control-flow
```

Angular Control Flow Syntax:

The Angular 17 @if else syntax

```
<div class="Personnes">
  <div class="personne-card">
    <div class="personne-Name">
      {{index}} {{ personne?.lastName }} {{ personne?.firstName }}
    </div>
    @if (personne?.picture)
    {
      <img width="300" alt="Angular Logo"
           [src]="personne?.picture">
    }
    @else {
      <h2>No image available</h2>
    }

    <div class="personne-description">
      {{personne?.address}}
    </div>
    <button (click)="OnpersonneViewed()">Details sur la personne</button>
  </div>
</div>
```

```
Go to component
<div class="Personnes">
  <div class="personne-card">
    <div class="personne-Name">
      {{index}} {{ personne?.lastName }} {{ personne?.firstName }}
    </div>
    @if (personne?.id==1) {
      <h1> Personne Id 1 detected</h1>
    }
    @if (personne?.picture)
    {
      <img width="300" alt="Angular Logo"
           [src]="personne?.picture">
    }
    @else {
      <h2>No image available</h2>
    }

    <div class="personne-description">
      {{personne?.address}}
    </div>
    <button (click)="OnpersonneViewed()">Details sur la personne</button>
  </div>
</div>
```

Angular Control Flow Syntax:

Angular ngIf Directive and the Elvis Operator

```
<div class="Personnes">
  <div class="personne-card">
    <div class="personne-Name">
      {{index}} {{ personne?.lastName }} {{ personne?.firstName }}
    </div>

    <img width="300" alt="Angular Logo" *ngIf="personne?.picture"
         [src]=> "personne?.picture">

    <div class="personne-description">
      {{personne?.address}}
    </div>
    <button (click)="OnpersonneViewed()">Details sur la personne</button>
  </div>
</div>
```

app > personnes > personnes.component.html > div.Personnes > div.personne-card > img

Go to component

```
<div class="Personnes">
  <div class="personne-card">
    <div class="personne-Name">
      {{index}} {{ personne?.lastName }} {{ personne?.firstName }}
    </div>

    <img width="300" alt="Angular Logo" *ngIf="isImg"
         [src]=> "personne?.picture">

    <div class="personne-description">
      {{personne?.address}}
    </div>
    <button (click)="OnpersonneViewed()">Details sur la personne</button>
  </div>
</div>
```

```
isImg(){
  return this.personne && this.personne.picture;
}
```

Angular Control Flow Syntax:

Angular ngIf Directive and the Elvis Operator

```
export const personnes : any[] =  
[  
    undefined,  
  
    {  
        id: 0,  
        firstName: 'Alice',  
        lastName: 'Jones',  
        age: 35,  
        picture: 'https://th.bing.com/th/id/R.53bb1512...',  
        email: 'alice.jones@test.com',  
        phone: '555-7890',  
        address: '456 Elm St',  
        created: new Date()  
    },  
    {
```

```
<div class="Personnes">  
    <div class="personne-card">  
        <div class="personne-Name">  
            {{index}} {{ personne?.lastName }} {{ personne?.firstName }}  
        </div>  
  
        <img width="300" alt="Angular Logo" *ngIf="isImg(); else noImage"  
              [src]="personne?.picture">  
  
        <ng-template #noImage>  
            <p> No Image Available</p>  
        </ng-template>  
  
        <div class="personne-description">  
            {{personne?.address}}  
        </div>  
        <button (click)="onPersonneViewed()">Details sur la personne</button>  
    </div>  
</div>
```

Angular Control Flow Syntax:

Angular ngClass Core Directive - Learn All Features

La directive `ngClass` dans Angular est un outil puissant pour dynamiquement ajouter ou retirer des classes CSS à des éléments HTML en fonction de l'état ou des données de votre application. Cela peut être particulièrement utile pour changer l'apparence d'un élément en réponse à des interactions de l'utilisateur ou des changements dans les données de l'application.

Angular Control Flow Syntax:

Angular ngClass Core Directive - Learn All Features

Utilisation de ngClass avec une méthode

La directive ngClass dans Angular est un outil puissant pour

Go to component

```
<div class="Personnes" >
  <div class="personne-card" [ngClass]="getAgeClass()">
    <div class="personne-Name">
      {{index}} {{ personne?.lastName }} {{ personne?.firstName }}
    </div>

    <img width="300" alt="Angular Logo" *ngIf="isImg(); else noImage"
         [src]="personne?.picture">

    <ng-template #noImage>
      <p> No Image Available</p>
    </ng-template>

    <div class="personne-description">
      {{personne?.address}}
    </div>
    <button (click)="OnpersonneViewed()">Details sur la personne</button>
  </div>
</div>
```

```
src > app > personnes > # personnes.component.css > jeune
31   button {
32     transition-duration: 0.4s; /* Durée de l'effet de transition */
33     cursor: pointer; /* Change le curseur en une main pour indiquer la possibilité de cliquer */
34     border-radius: 5px; /* Bordures arrondies */
35     box-shadow: 0 2px 5px rgba(0,0,0,0.2); /* Ombre légère pour donner de la profondeur */
36   }
37
38   button:hover {
39     background-color: #0056b3; /* Couleur de fond plus foncée lors du survol */
40     box-shadow: 0 5px 10px rgba(0,0,0,0.3); /* Ombre plus prononcée lors du survol */
41   }
42
43   .jeune {
44     border: 2px solid blue;
45   }
46
47   .adulte {
48     border: 2px solid green;
49   }
50
51   .senior {
52     border: 2px solid gray;
53   }
54
55
56
57
58
59
60
61
62
63
64
```

Angular Control Flow Syntax:

Angular ngClass Core Directive - Learn All Features

Utilisation de ngClass avec une méthode

La directive ngClass dans Angular est une autre manière de donner des classes CSS à des éléments HTML particulièrement utile pour ou des changements dans

```
src > app > personnes > TS personnes.component.ts > PersonnesComponent > getAgeClass
  9  export class PersonnesComponent implements OnInit {
27    OnpersonneViewed()
32  }
33
34  getAgeClass() {
35    const age = this.personne?.age;
36    if (age !== undefined) {
37      if (age < 25) {
38        return 'jeune';
39      } else if (age >= 25 && age < 65) {
40        return 'adulte';
41      } else {
42        return 'senior';
43      }
44    }
45    return ''; // retourne une chaîne vide ou une classe par défaut si age
46  }
47
48
49
50}
51
```

ou retirer des classes
Cela peut être
ctions de l'utilisateur

Angular Control Flow Syntax:

Angular ngClass Core Directive - Learn All Features

Utilisation de ngClass avec un objet

La directive ngClass dans Angular est un outil puissant pour dynamiquement ajouter ou retirer des classes CSS à des éléments HTML en fonction de l'état ou des données de votre application. Cela peut être particulièrement utile pour changer l'apparence d'un élément en réponse à des interactions de l'utilisateur ou des changements dans les données de l'application.

```
get ageClass() {  
  return {  
    jeune: this.personne.age < 25,  
    adulte: this.personne.age >= 25  
    && this.personne.age < 65,  
    senior: this.personne.age >= 65  
  };  
}
```

```
<div class="personne-card" [ngClass]="ageClass">
```

Angular Control Flow Syntax:

Angular ngClass Core Directive - Learn All Features

Utilisation de ngClass avec une chaîne conditionnelle

La directive ngClass dans Angular est un outil puissant pour dynamiquement ajouter ou retirer des classes CSS à des éléments HTML en fonction de l'état ou des données de votre application. Cela peut être particulièrement utile pour changer l'apparence d'un élément en réponse à des interactions de l'utilisateur ou des changements dans les données de l'application.

```
<div class="Personnes" >
  <div class="personne-card"
    [ngClass]="(personne?.age ?? 30) < 25 ?
      'jeune' : (personne?.age ?? 30) < 65 ? 'adulte' : 'senior'">
```

Ici, ?? est l'opérateur de coalescence nulle qui utilise la valeur par défaut 30 si personne?.age est undefined.

Angular Control Flow Syntax:

Angular ngStyle Core Directive - Learn All Features

Utilisation de ngStyle avec une méthode du composant

La directive ngStyle dans Angular vous permet de modifier les styles en ligne d'un élément HTML de manière dynamique, basée sur l'état ou les données de votre application. C'est particulièrement utile pour appliquer des styles qui ne peuvent pas être prédéfinis dans des classes CSS.

```
> app > personnes > personnes.component.html > div.Personnes > div.personne-card
  Go to component
1  <div class="Personnes" >
2    <div class="personne-card" [ngStyle]="getPersonStyle()">
3
4      <div class="personne-Name">
5        {{index}} {{ personne?.lastName }} {{ personne?.firstName }}
6      </div>
7
8      <img width="300" alt="Angular Logo" *ngIf="isImg(); else noImage"
9        [src]="personne?.picture">
```

```
src > app > personnes > personnes.component.ts > PersonnesComponent
  9  export class PersonnesComponent implements OnInit {
34    getAgeClass() {
46    }
47
48    getPersonStyle()
49    {
50      const age = this.personne?.age;
51      if (age !== undefined) {
52        if (age < 25) {
53          return { 'background-color': 'lightblue' };
54        } else if (age >= 25 && age < 65) {
55          return { 'background-color': 'lightgreen' };
56        } else {
57          return { 'background-color': 'gray' };
58        }
59      }
60      return { 'background-color': 'gray' }; // retourne
61    }
62  }
63
64
65
66  }
67
```

Angular Control Flow Syntax:

Angular ngStyle Core Directive - Learn All Features

Utilisation de ngStyle directement dans le template avec des expressions ternaires

```
Go to component
1 <div class="Personnes" >
2   <div class="personne-card"
3
4   [ngStyle]="{
5     'background-color': (personne?.age ?? 30) < 25 ? 'lightblue' :
6     (personne?.age ?? 30) < 65 ? 'lightgreen' : 'gray',
7     'border': (personne?.age ?? 30) < 25 ? '1px solid blue' :
8     (personne?.age ?? 30) < 65 ? '1px solid green' : '1px solid darkgray'
9   }"
10  >
11  |
```

Angular Control Flow Syntax:

@switch syntax in action

```
<div class="personne-description">
| {{personne?.address}}
</div>

<div class="personne-categorie">
  @switch (personne?.categorie)
  {
    @case ('dev')
    {
      <div class="categorie"> developpeur</div>
    }
    @case ('commercial') {
      <div class="categorie"> commercial</div>
    }
    @case ('ingenieur') {
      <div class="categorie"> ingenieur</div>
    }
    @default [
      <div class="categorie"> unknown</div>
    ]
  }
}
```

Angular Control Flow Syntax:

The Angular 17 @switch syntax in action

```
<div class="personne-description">
  {{personne?.address}}
</div>

<div class="personne-categorie">
  @switch (personne?.categorie)
  {
    @case ('dev')
    {
      <div class="categorie"> developpeur</div>
    }
    @case ('commercial') {
      <div class="categorie"> commercial</div>
    }
    @case ('ingenieur') {
      <div class="categorie"> ingenieur</div>
    }
    @default {
      <div class="categorie"> unknown</div>
    }
  }
</div>
```

Angular Control Flow Syntax:

Angular ngSwitch Core Directive In Detail

```
</div>

<div class="personne-categorie" [ngSwitch]="personne?.categorie">

    <div class="categorie" *ngSwitchCase=" 'dev' " > developpeur</div>

    <div class="categorie" *ngSwitchCase=" 'commercial' "> commercial</div>

    <div class="categorie" *ngSwitchCase=" 'ingenieur' "> ingenieur</div>

    <div class="categorie" *ngSwitchDefault> unknown</div>

</div>
```

Angular Control Flow Syntax:

Angular ng-container Core Directive - When to use it?

```
<ng-container [ngSwitch]="personne?.categorie" >

  <div class="personne-categorie" >

    <ng-container *ngSwitchCase=" 'dev' " >
      <div class="categorie" > developpeur</div>
    </ng-container>

    <div class="categorie" *ngSwitchCase=" 'commercial' " > commercial</div>

    <div class="categorie" *ngSwitchCase=" 'ingenieur' " > ingenieur</div>

    <div class="categorie" *ngSwitchDefault> unknown</div>

  </div>

</ng-container>
```

Angular Control Flow Syntax:

Angular ng-container Core Directive - When to use it?

```
<ng-container [ngSwitch]="personne?.categorie" >

  <div class="personne-categorie" >

    <ng-container *ngSwitchCase=" 'dev' " >
      <div class="categorie" > developpeur</div>
    </ng-container>

    <div class="categorie" *ngSwitchCase=" 'commercial' " > commercial</div>

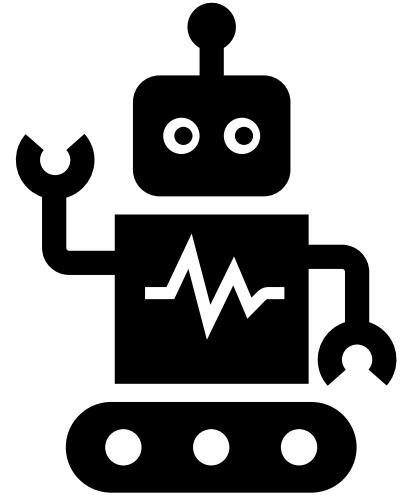
    <div class="categorie" *ngSwitchCase=" 'ingenieur' " > ingenieur</div>

    <div class="categorie" *ngSwitchDefault> unknown</div>

  </div>

</ng-container>
```

Angular Templates In Depth



Angular Templates Introduction with ng-template:

ng-template est une directive Angular qui vous permet de définir un template HTML pouvant être rendu dynamiquement dans votre application. Il sert de bloc de construction pour les structures conditionnelles ou répétitives au sein de vos vues, permettant une plus grande flexibilité dans la manipulation du contenu affiché

ng-template est souvent utilisé en combinaison avec des directives structurelles comme *ngIf, *ngFor, ou avec des directives telles que ngTemplateOutlet et ngContainer.

```
<div class="personne-Name">
  {{ 2 | exponential:3 }} {{index}} {{ personne?.lastName }} {{ personne?.firstName }}
</div>

<img width="300" alt="Angular Logo" *ngIf="isImg(); else noImage"
  [src]="personne?.picture">

<ng-template #noImage>
  <p> {{personne?.lastName}} has no Image Available yet</p>
  

</ng-template>
```

`ngTemplateOutlet` est une directive Angular utilisée pour injecter des templates à des endroits spécifiques dans le DOM. Elle est particulièrement utile pour réutiliser des portions de HTML dynamique à travers différents composants ou à divers endroits au sein d'un même composant. Cette directive permet une grande flexibilité dans la gestion des affichages conditionnels et la réutilisation de code HTML.

Fonctionnement de `ngTemplateOutlet`

Définition du template : Vous définissez d'abord un `<ng-template>` qui contient le markup HTML que vous souhaitez réutiliser. Ce template peut inclure des bindings Angular, comme l'interpolation, des directives, etc.

Utilisation de `ngTemplateOutlet` : Ensuite, vous utilisez `ngTemplateOutlet` pour spécifier où vous souhaitez que le contenu du `<ng-template>` soit rendu.

```
<div class="personne-Name">  
  {{ 2 | exponential:3 }} {{index}} {{ personne?.lastName }} {{ personne?.firstName }}  
</div>  
  
<img width="300" alt="Angular Logo" *ngIf="isImg(); else noImage"  
      [src]="personne?.picture">  
  
<ng-template #noImage>  
  <p> {{personne?.lastName}} has no Image Available yet</p>  
    
</ng-template>
```

ngTemplateOutlet est une directive Angular utilisée pour injecter des templates à des endroits spécifiques dans le DOM. Elle est particulièrement utile pour réutiliser des portions de HTML dynamique à travers différents composants ou à divers endroits au sein d'un même composant. Cette directive permet une grande flexibilité dans la gestion des affichages conditionnels et la réutilisation de code HTML.

<**ng-template**> qui contient le markup HTML que vous souhaitez réutiliser. Ce template peut inclure des bindings Angular, comme l'interpolation, des directives, etc.

Ensuite, vous utilisez **ngTemplateOutlet** pour spécifier où vous souhaitez que le contenu du <**ng-template**> soit rendu.

Angular Template Instantiation with ngTemplateOutlet:

```
<ng-template #noImage let-personneName>

  <p> {{personneName}} has no Image Available yet</p>
  

</ng-template>

<ng-container *ngTemplateOutlet="noImage"></ng-container>

<div> start date : {{ date |date:'MMM/dd/yyyy' }} </div>
<app-personnes *ngFor="let per of etudiants ; index as i; let first = first; let last = last;
even as ev ; odd as odd;" (PersonneSelected)="OnPersonneSelected($event)" [personne] = "per"
[index] = "i +1" [class.is-first]="first" [class.is-last]="last"
[class.is-even]="ev" [class.is-odd]="odd"/>
```

Angular Template Instantiation with ngTemplateOutlet:

```
ng-template #noImage let-personneName="name">

<p> {{personneName}} has no Image Available yet</p>


</ng-template>

<ng-container *ngTemplateOutlet="noImage ; context : { name : etudiants[0]?.firstName}" >
</ng-container>

<hr>

<div> start date : {{ date | date:'MMM/dd/yyyy' }} </div>
<app-personnes *ngFor="let per of etudiants ; index as i; let first = first; let last = last;
even as ev ; odd as odd;" (PersonneSelected)="OnPersonneSelected($event)" [personne] = "per"
[index] = "i +1" [class.is-first]="first" [class.is-last]="last"
[class.is-even]="ev" [class.is-odd]="odd"/>
```

Angular Templates as Component Inputs:

```
<ng-template #noImage let-personneName="name">  
  <p> {{personneName}} has no Image Available yet</p>  
    
</ng-template>  
  
<div> start date : {{ date |date:'MMM/dd/yyyy' }} </div>  
  
<app-personnes *ngFor="let per of etudiants ; index as i; let first = first; let last =  
last;  
even as ev ; odd as odd;"  
(PersonneSelected)="OnPersonneSelected($event)" [personne] = "per"  
[index] = "i +1" [class.is-first]="first" [class.is-last]="last"  
[class.is-even]="ev" [class.is-odd]="odd"  
[noImageTpl]="noImage"/>
```

personnes.component.ts

```
@Input()  
noImageTpl : TemplateRef<any> | null = null;
```

Angular Templates as Component Inputs:

```
<img width="300" alt="Angular Logo" *ngIf="isImg(); else noImg"
    [src]="personne?.picture">

<ng-template #noImg >

    <ng-container *ngTemplateOutlet="noImageTmpl; context : {name : personne?.lastName}">
        </ng-container>

</ng-template>
```


Exercice 2: Composant d'Affichage de Produit

Objectif : Construire un composant pour afficher les détails d'un produit et permettre à l'utilisateur de signaler son intérêt pour ce produit à travers un événement.

Fonctionnalités :

Affichage des détails du produit : Utiliser `@Input` pour recevoir les informations du produit (nom, prix, description).
Signaler un intérêt : Émettre un événement avec `@Output` lorsque l'utilisateur clique sur un bouton pour indiquer son intérêt.

Étapes :

Créer un composant `ProductComponent`.

Ajoutez des propriétés avec `@Input()` pour le nom, le prix et la description du produit.

Ajoutez un `@Output()` pour émettre un événement lorsque l'utilisateur exprime son intérêt.

Template du composant :

Utilisez le data binding pour afficher les informations du produit.

Ajoutez un bouton qui utilise `(click)` pour émettre l'événement d'intérêt.

Exercice 3: Composant pour Afficher une Liste de Messages

Objectif : Développer un composant qui affiche une liste de messages et permet à l'utilisateur de sélectionner un message pour voir plus de détails.

Fonctionnalités :

Affichage des messages : Utiliser un composant pour afficher chaque message dans une liste.

Sélection de message : Émettre un événement lorsque l'utilisateur sélectionne un message.

Étapes :

Créer un composant MessageComponent.

Utilisez @Input() pour recevoir les données de chaque message.

Le template doit afficher brièvement le message.

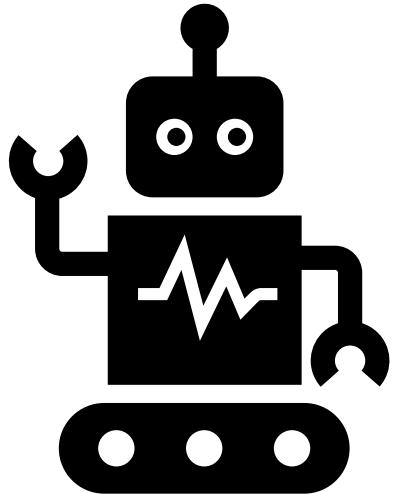
Créer un composant MessageListComponent.

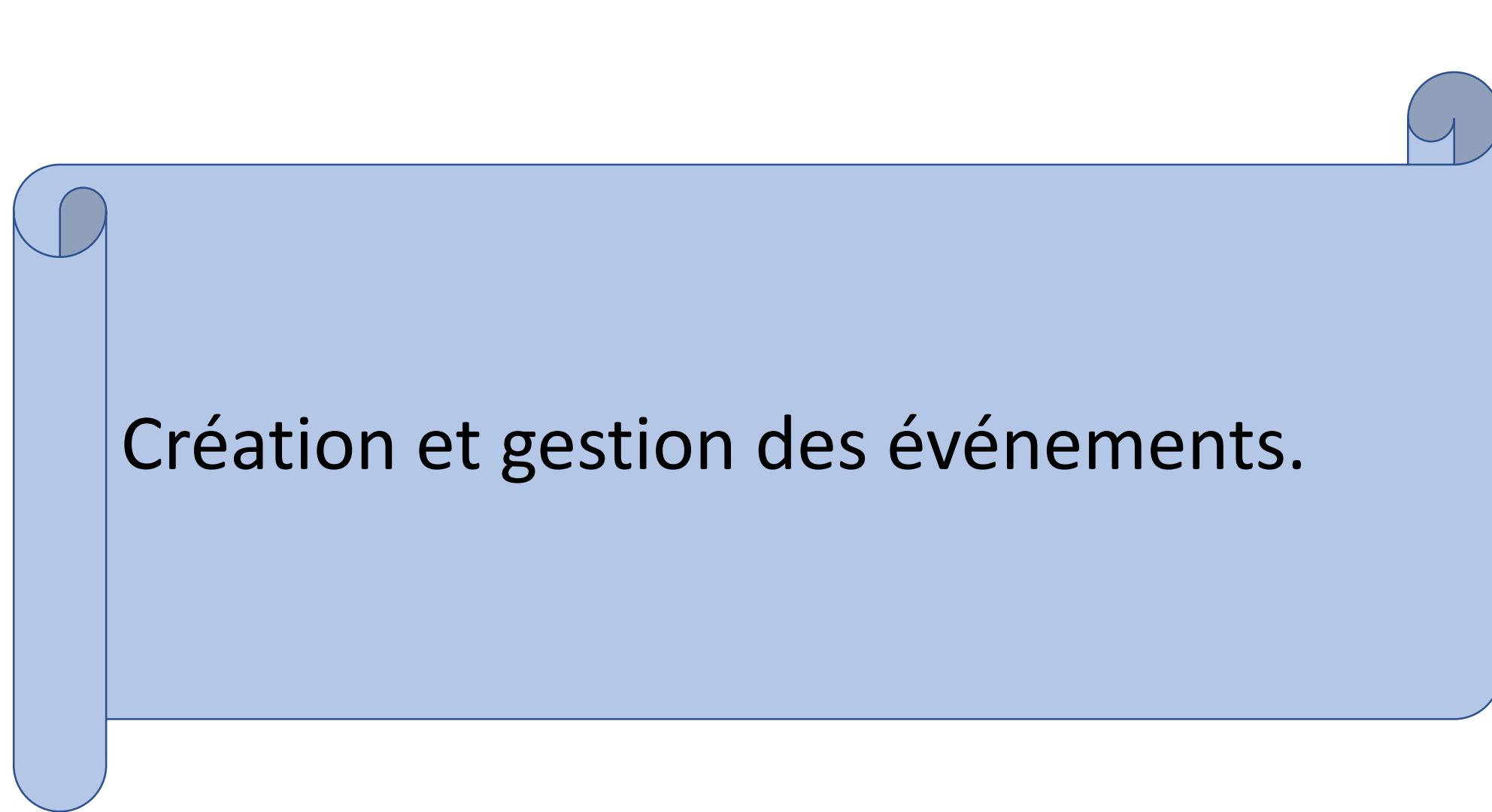
Ce composant contient une liste de messages.

Utilisez *ngFor pour lister tous les messages.

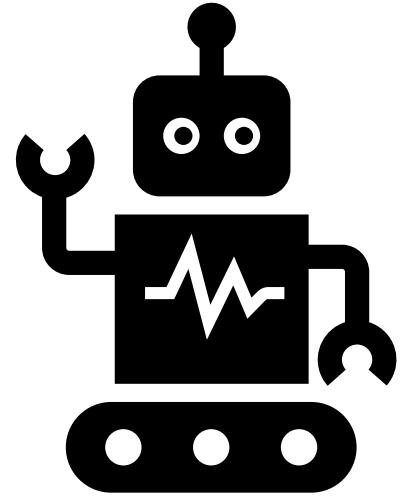
Utilisez (click) sur chaque message pour émettre un événement indiquant le message sélectionné.

Travailler avec les templates : data binding, directives.





Création et gestion des événements.



La gestion des événements joue un rôle crucial dans le développement d'applications avec Angular, permettant une interaction directe avec les utilisateurs et facilitant l'application de modifications dynamiques. Ce chapitre détaillera les méthodes par lesquelles Angular traite ces événements et vous montrera comment les exploiter pour élaborer des applications hautement interactives.

Dans le contexte du développement front-end, interagir avec les éléments d'une application—que ce soit par des clics de souris, des mouvements de curseur, des frappes sur le clavier, ou autres formes d'interaction—est fondamental. Angular offre un ensemble complet d'outils robustes destinés à la gestion de ces interactions utilisateur, vous permettant de répondre efficacement à chaque action.

Syntaxe de Base

Angular utilise la syntaxe `(event)="handler()"` pour écouter les événements DOM. Dans cette syntaxe, event est le nom de l'événement DOM et handler() est la méthode du composant qui sera exécutée lors du déclenchement de cet événement.

Exemple: Supposons que vous ayez un composant appelé `ButtonComponent`.

Dans `button.component.html` :

```
<button (click)="handleClick()">Cliquez-moi!</button>
```

Dans button.component.ts :

```
@Component({  
  // ...  
})  
export class ButtonComponent {  
  handleClick() {  
    console.log("Bouton cliqué!");  
  }  
}
```

Événements Angular Couramment Utilisés

Angular prend en charge une grande variété d'événements DOM. Voici une liste des plus couramment utilisés :

- **click:** Événement de clic sur un élément
- **dblclick:** Événement de double-clic sur un élément
- **submit:** Événement de soumission de formulaire
- **focus:** Événement de mise au point sur un élément
- **blur:** Événement de perte de focus sur un élément
- **keyup et keydown:** Événements de pression et de relâchement de touche
- **change:** Modification de la valeur d'un élément de formulaire
- **input:** Chaque changement de valeur dans un élément de saisie
- **mouseenter et mouseleave:** Événements de survol d'élément

Événements Customisés avec EventEmitter

Vous pouvez créer vos propres événements personnalisés en utilisant la classe EventEmitter.

Exemple Dans un composant enfant (child.component.ts):

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

```
@Component({
  // ...
})
export class ChildComponent {
  @Output() customEvent = new EventEmitter<string>();

  triggerCustomEvent() {
    this.customEvent.emit("Data from child");
  }
}
```

Dans un composant parent (parent.component.html):

```
<app-child (customEvent)="handleCustomEvent($event)"></app-child>
```

Et dans le parent.component.ts :

```
handleCustomEvent(data: string) {  
  console.log(`Received data: ${data}`);  
}
```

Directives et Pipes (8 heures)

Professeur Titulaire : Millimono Sory (PhD, AI)

Plan Du Cours



○ Directives structurelles et attribut.

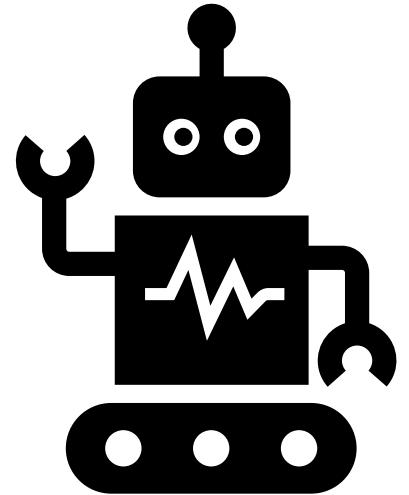
○ Utilisation et création de pipes pour le formatage des données.

○ Projet pratique intégrant directives et pipes.

○ Exercices pratiques.



les directives dans Angular.



Les directives sont l'un des concepts les plus importants d'Angular, dans cette section, nous verrons ce qu'est une directive et ses types et comment créer nos propres directives

Qu'entend-on par directives dans Angular ?

Les directives sont des classes qui ajoutent un nouveau comportement ou modifient le comportement existant aux éléments du modèle.

Fondamentalement, les directives sont utilisées pour manipuler le DOM, par exemple en ajoutant/supprimant l'élément du DOM ou en changeant l'apparence des éléments du DOM.

Types de directives

- Directive sur les composants
- Directive structurelle
- Directive d'attribut

Directive sur les composants

Les composants sont des directives spéciales dans Angular. Il s'agit de la directive avec un modèle (template ou templateUrls).

Directive structurelle

Les directives structurelles modifient la structure du DOM en ajoutant, remplaçant ou supprimant des éléments. Ce type de directive peut conditionner l'affichage d'un élément dans le DOM ou répéter un élément plusieurs fois.

exemples communs :

***ngIf** : Affiche ou masque un élément en fonction de la valeur d'une expression booléenne.

***ngFor** : Répète un élément de template pour chaque élément d'une liste.

***ngSwitch, *ngSwitchCase, *ngSwitchDefault** : Affiche des éléments en fonction d'une expression.

Directives d'Attribut

Les directives d'attribut modifient le comportement ou l'apparence des éléments sur lesquels elles sont appliquées, sans altérer leur disposition dans le DOM.

Exemples communs :

[ngClass] : Ajoute ou supprime des classes CSS en fonction d'une expression.

[ngStyle] : Applique des styles dynamiques à un élément.

[ngModel] : Crée une liaison bidirectionnelle sur les formulaires.

Utilisation des Directives pour Manipuler le DOM

Les directives dans Angular permettent de manipuler le DOM de manière déclarative et réactive, ce qui aide à séparer la logique de la vue et à garder le code propre et maintenable.

Réactivité : Les modifications de l'état de l'application peuvent immédiatement se refléter dans le DOM grâce aux directives structurelles et d'attribut.

Performance : Angular optimise les modifications du DOM, en particulier avec *ngFor en utilisant des algorithmes intelligents pour recharger uniquement les parties nécessaires du DOM.

Conseils pour l'Utilisation des Directives

Éviter les Complexités Excessives : Garder les expressions dans les directives simples pour éviter les performances dégradantes.

Utilisez des Stratégies de DéTECTeur de Changement : Utiliser ChangeDetectionStrategy.OnPush dans les composants peut améliorer les performances avec des listes grandes ou complexes.

Création de Directives Personnalisées dans Angular

Dans Angular, une directive personnalisée est une classe décorée avec le décorateur **@Directive**, qui vous permet de manipuler le comportement des éléments du DOM de manière programmée. Elle peut modifier l'apparence, le comportement, ou même la structure des éléments sans nécessiter un template associé, contrairement aux composants.

Avantages des Directives Personnalisées

Modularité et Réutilisabilité : Les directives permettent de réutiliser des comportements liés au DOM à travers différentes parties de votre application, réduisant la duplication de code et améliorant la maintenance.

Simplicité et Clarté : En extrayant des comportements spécifiques à des directives, vous simplifiez vos composants, rendant votre code plus clair et plus facile à comprendre.

Contrôle Fin sur le DOM : Les directives offrent un contrôle précis sur le DOM et les interactions avec l'utilisateur, permettant des manipulations complexes qui seraient difficiles à gérer autrement.

Extension des Capacités HTML : Les directives permettent d'étendre les éléments HTML standard avec des fonctionnalités supplémentaires, adaptant le HTML standard à vos besoins spécifiques.

Exemple de Mise en Place de Directives Personnalisées

Pour illustrer la création d'une directive personnalisée, prenons l'exemple d'une directive qui modifie la couleur de fond d'un élément au survol de la souris.

```
ng generate directive nom-de-la-directive
```

```
ng g d nom-de-la-directive
```

```
ng g d highlight
```

```
import { Directive, ElementRef, HostListener, Renderer2 } from '@angular/core';

@Directive({
  selector: '[appHighlight]'
})
export class HighlightDirective {
  constructor(private el: ElementRef, private renderer: Renderer2) {}

  @HostListener('mouseenter') onMouseEnter() {
    this.changeBackgroundColor('yellow');
  }

  @HostListener('mouseleave') onMouseLeave() {
    this.changeBackgroundColor(null);
  }

  private changeBackgroundColor(color: string | null) {
    this.renderer.setStyle(this.el.nativeElement, 'background-color', color);
  }
}

<div class="Personnes" >
  <div class="personne-card" appHighlight
    >
```

Objectif : Créer une directive qui affiche un tooltip personnalisé lors du survol d'un élément.

```
.tooltip {  
  position: absolute;  
  background-color: black;  
  color: white;  
  padding: 4px 8px;  
  border-radius: 4px;  
  z-index: 1000;  
}
```

```
<button appTooltip="Ceci est un tooltip!"  
(click)="OnpersonneViewed()">Details sur la personne</button>
```

```
import { Directive, ElementRef, Input, HostListener,
Renderer2, ComponentFactoryResolver,
ViewContainerRef } from '@angular/core';

@Directive({
  selector: '[appTooltip]'
})
export class TooltipDirective {
  @Input('appTooltip') tooltipText: string | undefined;
  private tooltipDiv: HTMLElement | null = null;

  constructor(private el: ElementRef, private renderer:
  Renderer2, private viewContainerRef: ViewContainerRef,
  private componentFactoryResolver:
  ComponentFactoryResolver) {}

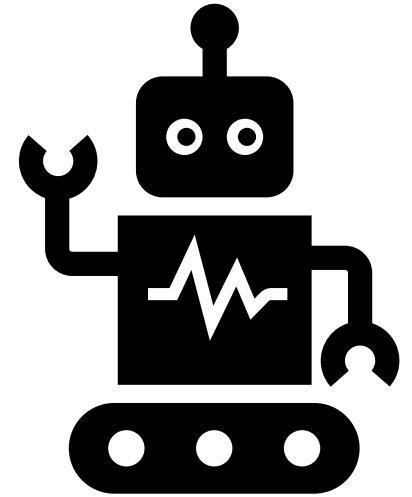
  @HostListener('mouseenter') onMouseEnter() {
    if (!this.tooltipDiv) {
      this.showTooltip();
    }
  }

  @HostListener('mouseleave') onMouseLeave() {
    this.hideTooltip();
  }
}
```

```
private showTooltip(): void {
  this.tooltipDiv = this.renderer.createElement('div');
  this.renderer.addClass(this.tooltipDiv, 'tooltip');
  this.renderer.setProperty(this.tooltipDiv, 'textContent',
  this.tooltipText);
  this.renderer.setStyle(this.tooltipDiv, 'position', 'absolute');
  this.renderer.setStyle(this.tooltipDiv, 'top',
  `${this.el.nativeElement.offsetTop +
  this.el.nativeElement.offsetHeight}px`);
  this.renderer.setStyle(this.tooltipDiv, 'left',
  `${this.el.nativeElement.offsetLeft}px`);
  this.renderer.appendChild(document.body, this.tooltipDiv);
}

private hideTooltip(): void {
  if (this.tooltipDiv) {
    this.renderer.removeChild(document.body,
    this.tooltipDiv);
    this.tooltipDiv = null;
  }
} }
```

Angular Directives In Depth.



Angular Host Binding in Detail - DOM Properties vs Attributes

@HostBinding Decorator:

- **@HostBinding** est un décorateur en Angular qui permet à une directive de se lier à une propriété de l'élément hôte. Ce décorateur peut être utilisé pour lier des propriétés telles que les classes, les styles, les attributs, etc., de l'élément DOM auquel la directive est appliquée.

```
@HostBinding('className')
get cssClass()
{
  return 'highlight';
}
```

```
@HostBinding('class.highlight')
get cssClass() {
  return true;
}
```

```
/* CSS pour la classe highlight */
.highlight {
  background-color: red;
  color: black;
  padding: 5px; /* Un peu de padding pour mieux distinguer
l'élément */

  border-radius: 5px; /* Bords arrondis pour un look plus
doux */
}
```

Angular Host Binding in Detail - DOM Properties vs Attributes

@HostBinding Decorator:

- **@HostBinding** est un décorateur en Angular qui permet à une directive de se lier à une propriété de l'élément hôte. Ce décorateur peut être utilisé pour lier des propriétés telles que les classes, les styles, les attributs, etc., de l'élément DOM auquel la directive est appliquée.

```
@HostBinding('style.border')
get cssClass()
{
  return "20px solid black";
}
```

angular Host Listener - Handling Events in Directives

@HostListener

permet d'associer des méthodes dans un composant ou une directive à des événements comme le clic de souris, les frappes au clavier, les mouvements de souris, etc. Le décorateur écoute spécifiquement les événements sur l'élément auquel la directive ou le composant est appliquée, appelé "élément hôte".

```
@HostListener('mouseenter') onMouseEnter() {  
  this.changeBackgroundColor('yellow');  
}
```

```
@HostListener('mouseleave') onMouseLeave() {  
  this.changeBackgroundColor(null);  
}
```

```
private changeBackgroundColor(color: string | null) {  
  this.renderer.setStyle(this.el.nativeElement, 'background-color', color);  
}
```

Angular Directive Export As syntax - When to Use it and Why

```
@Directive({
  selector: '[appHighlight]',
  exportAs : 'hl'
})
export class HighlightDirective {
  constructor(private el: ElementRef, private renderer: Renderer2) {}

  @HostListener('mouseenter') onMouseEnter() {
    this.changeBackgroundColor('yellow');
  }

  @HostListener('mouseleave') onMouseLeave() {
    this.changeBackgroundColor(null);
  }

  private changeBackgroundColor(color: string | null) {
    this.renderer.setStyle(this.el.nativeElement, 'background-color', color);
  }

  private toogle()
{
  this.changeBackgroundColor('black');
}
}
```

```
<div class="Personnes" >
  <div class="personne-card" appHighlight #hiliter = 'hl'>

    <div class="personne-Name" *ngIf="isImg()">
      {{ 2 | exponential:3 }} {{index}} {{ personne?.lastName }} {{ personne?.firstName }} </div>

      <img width="300" alt="Angular Logo" *ngIf="isImg();else noImg"
            [src]="personne?.picture">

      <ng-template #noImg >
        <ng-container *ngTemplateOutlet="noImageTpl; context: {name : personne?.lastName}">
          </ng-container>
      </ng-template>

      <button appTooltip="{{ personne?.address }}"
        (click)="hiliter.toogle()">Details sur la personne</button>
    </div>
  </div>
</div>
```

Angular Structural Directives - Understanding the Star Syntax

```
Microsoft Windows [version 10.0.19045.4291]
(c) Microsoft Corporation. Tous droits réservés.

C:\FILES\PHD\COURS_UNIVERSITE\Angular\my-project>ng g d directives/ngx-unless
CREATE src/app/directives/ngx-unless.directive.spec.ts (245 bytes)
CREATE src/app/directives/ngx-unless.directive.ts (157 bytes)
UPDATE src/app/app.module.ts (848 bytes)

C:\FILES\PHD\COURS_UNIVERSITE\Angular\my-project>
```

```
<app-personnes *ngIf="etudiants[0] as per"
(PersonneSelected)="OnPersonneSelected($event)" [personne] = "per"
[noImageTpl]="noImage"
/>
```

Angular Structural Directives - Understanding the Star Syntax

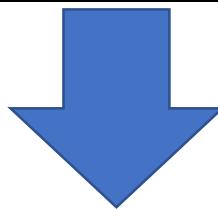
```
Microsoft Windows [version 10.0.19045.4291]
(c) Microsoft Corporation. Tous droits réservés.

C:\FILES\PHD\COURS_UNIVERSITE\Angular\my-project>ng g d directives/ngx-unless
CREATE src/app/directives/ngx-unless.directive.spec.ts (245 bytes)
CREATE src/app/directives/ngx-unless.directive.ts (157 bytes)
UPDATE src/app/app.module.ts (848 bytes)

C:\FILES\PHD\COURS_UNIVERSITE\Angular\my-project>
```

ng g d directives/ngx-unless

```
<app-personnes *ngIf="etudiants[0] as per"
(PersonneSelected)="OnPersonneSelected($event)" [personne] = "per"
[noImageTpl]="noImage"
/>
```



```
<ng-template [ngIf]="etudiants[0]" let-per>
  <app-personnes
    (PersonneSelected)="OnPersonneSelected($event)" [personne] = "per"
    [noImageTpl]="noImage"
  />
</ng-template>
```

Angular Structural Directives - Step-by-Step Implementation

```
constructor(private templateRef : TemplateRef<any> ,private viewContainer: ViewContainerRef)
{  
}  
}
```

```
@Input()  
set ngxUnless(condition : boolean)  
{  
}  
}
```

ng g d directives/ngx-unless

```
ab :boolean = false;
```

```
<div *ngxUnless="ab"> start date : {{ date |date :'MMM/dd/yyyy' }} </div>
```

Angular Structural Directives - Step-by-Step Implementation

```
@Input()  
set ngxUnless(condition : boolean)  
{  
  if(!condition)  
  {  
    this.viewContainer.createEmbeddedView(this.templateRef);  
  
  }  
  else if (condition)  
  {  
    this.viewContainer.clear();  
  }  
}  
}
```

Angular Structural Directives - Step-by-Step Implementation

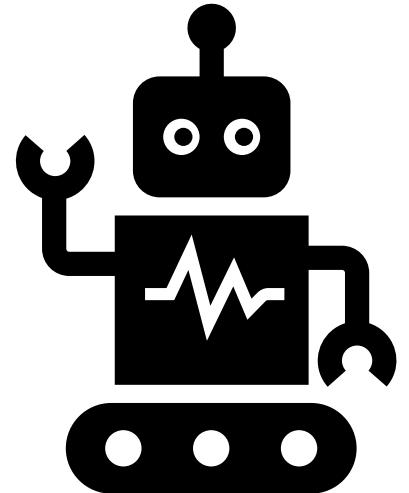
```
private hasView = false;

@Input()
set ngxUnless(condition : boolean)
{
  if(!condition && !this.hasView)
  {
    this.viewContainer.createEmbeddedView(this.templateRef);
    this.hasView = true;
  }
  else if (condition && this.hasView)
  {
    this.viewContainer.clear();
    this.hasView = false;
  }
}
```

Angular Structural Directives - Step-by-Step Implementation

```
<app-personnes  
  (PersonneSelected)="OnPersonneSelected($event)" [personne] = "per"  
  [noImageTpl]="noImage" *ngxUnless="!per?.picture">  
  
</app-personnes>
```


Utilisation et création de pipes pour le formatage des données.



Pipes dans Angular : Présentation et Utilité

Dans Angular, les pipes sont des outils puissants utilisés pour transformer, formater ou manipuler des données directement dans les templates de manière déclarative. Similaires aux filtres dans AngularJS ou aux helpers dans d'autres frameworks de template, les pipes permettent de prendre des données d'entrée et de les transformer en un format plus adapté à l'affichage.

Qu'est-ce qu'un Pipe ?

Un pipe est une classe décorée avec `@Pipe`, où vous définissez une méthode de transformation qui prend une valeur d'entrée, effectue une opération, et retourne la valeur transformée. Angular propose plusieurs pipes prédéfinis et permet également aux développeurs de créer des pipes personnalisés pour répondre à des besoins spécifiques.

Utilité des Pipes

- 1. Formatage des Données :** Les pipes sont souvent utilisés pour formater des données brutes en formats plus lisibles. Par exemple, afficher des dates, des montants monétaires, des décimales, ou appliquer des formattages textuels comme la mise en majuscule, la minuscule, etc.
- 2. Internationalisation :** Les pipes peuvent aider à adapter une application aux différentes locales, par exemple en formatant les dates et les monnaies selon les normes locales.
- 3. Filtrage et Tri :** Dans des cas plus avancés, les pipes peuvent être utilisés pour filtrer des listes ou trier des données directement dans le template.

Syntaxe

La syntaxe des pipes Angular est inspirée de celle du shell Unix

```
<div>{{ user.lastName | uppercase }}</div>
```

Des paramètres peuvent être passés aux pipes. Ils sont placés après le nom du pipe et séparés par des deux-points :

```
<div>{{ user.registrationDate | date:'dd/MM/yyyy' }}</div>
```

```
<div>{{ user.registrationDate | date:'dd/MM/yyyy hh:mm':'UTC' }}</div>
```

```
<div>{{ user.registrationDate | date:'dd/MM/yyyy hh:mm':'+0200':'fr' }}</div>
```

Les pipes peuvent être enchaînés :

```
<div>{{ user.birthDate | date | uppercase }}</div>
```

Exemples de Pipes Intégrés dans Angular

DatePipe : Formate une date selon un format localisé.

UpperCasePipe et LowerCasePipe : Transforme le texte en majuscules ou en minuscules.

DecimalPipe : Transforme un nombre en une chaîne avec des décimales.

CurrencyPipe : Formate un nombre en format monétaire, avec des symboles de devises.

PercentPipe : Convertit un nombre en pourcentage.

Exemple d'Utilisation d'un Pipe

Supposons que vous ayez une date ISO (par exemple "2020-03-14T00:00:00Z") que vous souhaitez formater de manière conviviale :

```
<p>Date: {{ today | date:'fullDate' }}</p>
```

Dans cet exemple, **today** est une variable dans votre composant Angular contenant la date à afficher, et **date:'fullDate'** est le pipe qui transforme cette date ISO en un format plus lisible, comme "samedi 14 mars 2020".

- **CurrencyPipe** transforme un nombre en une chaîne de caractère formatée avec la devise selon les règles de locale
- **DatePipe** formate une valeur de date selon les règles de locale
- **DecimalPipe** formate une valeur en fonction d'option de formatage des décimales et des règles de locale
- **I18nPluralPipe** fait correspondre une valeur à une chaîne de caractère qui pluralise la valeur selon les règles de locale
- **I18nSelectPipe** sélecteur générique qui affiche la chaîne de caractère qui correspond à la valeur actuelle
- **JsonPipe** convertit une valeur en sa représentation au format JSON, utile pour le débogage
- **KeyValuePipe** transforme l'objet ou la Map en un tableau de paires clé-valeur
- **LowerCasePipe** transforme le texte en minuscules
- **PercentPipe** transforme un nombre en une chaîne de caractères formatée avec en pourcentage, selon les règles de locale
- **SlicePipe** crée un nouveau tableau ou chaîne de caractère contenant un sous-ensemble (tranche) des éléments
- **TitleCasePipe** transforme le texte en casse de titre
- **UpperCasePipe** transforme le texte en majuscules

Création d'un Pipe Personnalisé

ng generate pipe nom-du-pipe

ng g p nom-du-pipe

ng g p exponential

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
  name: 'exponential'
})
export class ExponentialPipe implements PipeTransform {

  transform(value: number, exponent: number = 1): number {
    return Math.pow(value, exponent);
  }
}
```

`{{ 2 | exponential:3 }}`

Bonnes Pratiques pour la Conception de Pipes Réutilisables

1. Responsabilité Unique: Chaque pipe doit avoir une seule responsabilité. Cela facilite sa compréhension, son test, et sa réutilisation.

2. Pureté: Les pipes doivent être purs, c'est-à-dire qu'ils ne doivent pas modifier les objets passés en entrée ni maintenir un état interne qui affecte leur sortie. Un pipe pur a un nombre déterministe de sorties pour des entrées données, ce qui améliore les performances en permettant à Angular d'optimiser le processus de détection de changements.

3. Performance: Pensez à la performance lors de la conception de votre pipe. Évitez des opérations coûteuses telles que les appels HTTP ou les opérations complexes dans un pipe, car elles peuvent être exécutées fréquemment.

4. Paramétrisation: Rendez vos pipes flexibles en permettant des paramètres qui adaptent leur comportement. Par exemple, un pipe de formatage de date pourrait accepter des formats de date différents.

5. Testabilité: Assurez-vous que vos pipes sont testables. Concevez-les de manière à ce qu'ils soient faciles à tester en isolation sans dépendances extérieures.

6. Documentation: Documentez clairement ce que fait le pipe et comment l'utiliser. Cela inclut la description des paramètres qu'il accepte et des exemples de sortie.

7. Sécurité: Soyez conscient des problèmes de sécurité, surtout si votre pipe manipule du HTML ou des données susceptibles de provenir de sources non fiables. Utilisez des fonctions de nettoyage pour éviter les attaques comme le Cross-Site Scripting (XSS).

Exercices Pratiques

Exercice 1: Pipe de Filtrage et de Tri

Objectif: Créer une application qui affiche une liste de tâches filtrées par statut et triées par priorité.

Modèle de Données:

Chaque tâche a un title, status, et priority.

Création des Pipes:

Un pipe pour filtrer les tâches par statut.

Un pipe pour trier les tâches par priorité.

Utilisation dans un Composant:

Utiliser les pipes dans le template pour afficher uniquement les tâches actives, triées par leur priorité.

Exercices Pratiques

Exercice 2: Pipe de Transformation Conditionnelle

Objectif: Transformer les données d'entrée en fonction de plusieurs critères fournis sous forme de paramètres.

Scénario:

Vous avez des descriptions de produits qui doivent être affichées de manière concise dans un catalogue mais de manière détaillée dans la page du produit.

Création du Pipe:

Un pipe qui accepte la longueur maximale du texte, si les mots doivent être complets, et quel texte ajouter à la fin si le texte est coupé.

Utilisation:

Appliquer ce pipe dans différents composants avec différentes configurations pour montrer son adaptabilité.

Description du Projet : Tableau de Bord de Suivi de Projets

L'application permettra aux utilisateurs de visualiser rapidement l'état des projets, avec des informations telles que le budget, la progression, les dates de début et de fin, et plus encore. Les pipes seront utilisés pour formater les dates, les montants monétaires, la progression en pourcentage, et pour filtrer ou trier les listes de projets.

Étapes de Développement

Configuration de l'Environnement:

Créez un nouveau projet Angular avec Angular CLI :

- ng new projectDashboard

Naviguez dans votre nouveau projet :

- cd projectDashboard

1. Création de Modèles de Données:

- Définissez un modèle de projet dans src/app/models/project.model.ts:

```
export class Project {  
  constructor(  
    public id: number,  
    public title: string,  
    public startDate: Date,  
    public endDate: Date,  
    public budget: number,  
    public progress: number  
  ) {}  
}
```

Création de Composants:

Générez un composant pour afficher la liste des projets :

- ng generate component project-list

Implémentation des Pipes:

Pipe de Date: Formate les dates de début et de fin.

```
ng generate pipe pipes/dateFormat
```

Implémentez le pipe pour utiliser DatePipe d'Angular avec un format personnalisé.

Pipe Monétaire: Formate le budget des projets.

```
ng generate pipe pipes/currencyFormat
```

Utilisez CurrencyPipe pour afficher les montants en euros avec des séparateurs de milliers.

Pipe de Progression: Convertit les valeurs numériques de progression en pourcentages formatés.

```
ng generate pipe pipes/progressFormat
```

implémentez une transformation qui ajoute le symbole % à la progression.

Intégration et Test:

Dans project-list.component.html, utilisez les pipes pour formater l'affichage :

```
<ul>
  <li *ngFor="let project of projects">
    <h2>{{ project.title | uppercase }}</h2>
    <p>Date de début: {{ project.startDate | dateFormat:'fullDate' }}</p>
    <p>Date de fin: {{ project.endDate | dateFormat:'mediumDate' }}</p>
    <p>Budget: {{ project.budget | currencyFormat:'EUR' }}</p>
    <p>Progression: {{ project.progress | progressFormat }}</p>
  </li>
</ul>
```

Déploiement et Révision:

- Testez l'application localement avec ng serve.
- Revoyez et ajustez si nécessaire pour assurer l'exactitude des formats.

Utiliser un pipe en dehors d'un template

Il est également possible d'utiliser des pipes dans une classe de composant en l'injectant dans son constructeur et en appelant sa méthode transform. Le pipe doit être importé dans le module auquel appartient le composant et ajouté aux providers du composant ou du module (à privilégier).

```
import { ExponentialPipe } from './exponential.pipe';

private exponentialPipe = new ExponentialPipe();

calculateExponential()
{
    let value = 5;
    let exponent = 3;
    let result = this.exponentialPipe.transform(value, exponent);
    console.log('Result:', result); // Affiche : Result: 125
}
```

Angular Control Flow Syntax:

Angular Built-In Pipes - Complete Catalog

```
<div> start date : {{ date |date }} </div>
<app-personnes *ngFor="let per of etudiants;
index as i; let first = first; let last = last;
even as ev ; odd as odd;">
  (PersonneSelected)="OnPersonneSelected($event)"
  [personne] = "per"
  [index] = "i +1"    [class.is-
first]="first"      [class.is-last]="last"
  [class.is-even]="ev"    [class.is-odd]="odd"/>
```

```
<div> start date : {{ date |date :'MMM/dd/yyyy' }} </div>
```

Angular Control Flow Syntax:

Angular Built-In Pipes - Complete Catalog

```
<div> start date : {{ date |date :'MMM/dd/yyyy' }} </div>
<app-personnes *ngFor="let per of etudiants | slice:0:2; index as i; let first = first; let
last = last;
even as ev ; odd as odd;" 
(PersonneSelected)="OnPersonneSelected($event)" [personne] = "per"
[index] = "i +1" [class.is-first]="first" [class.is-last]="last"
[class.is-even]="ev" [class.is-odd]="odd"/>
```

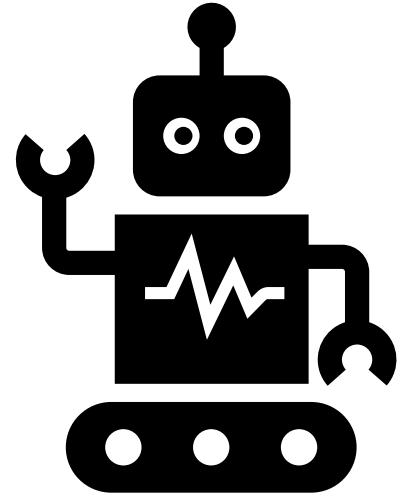
```
{{etudiants | json}}
<div> start date : {{ date |date :'MMM/dd/yyyy' }} </div>
<app-personnes *ngFor="let per of etudiants | slice:0:2; index as i; let first = first; let
last = last;
even as ev ; odd as odd;" 
(PersonneSelected)="OnPersonneSelected($event)" [personne] = "per"
[index] = "i +1" [class.is-first]="first" [class.is-last]="last"
[class.is-even]="ev" [class.is-odd]="odd"/>
```

Angular Control Flow Syntax:

Angular Built-In Pipes - Complete Catalog

```
<div *ngFor="let pair of etudiants[0] |keyvalue">  
  {{pair.key + ' ' + pair.value}}  
  
</div>  
  
<div> start date : {{ date |date:'MMM/dd/yyyy' }} </div>  
 <app-personnes *ngFor="let per of etudiants | slice:0:2; index as i; let first = first; let  
 last = last;  
   even as ev ; odd as odd;"  
   (PersonneSelected)="OnPersonneSelected($event)" [personne] = "per"  
   [index] = "i +1" [class.is-first]="first" [class.is-last]="last"  
   [class.is-even]="ev" [class.is-odd]="odd"/>
```


Angular Pipes In Depth.



ng g p FitredBycategorie

```
import { Pipe, PipeTransform } from '@angular/core';
import { Personne } from './personne';

@Pipe({
  name: 'fitredBycategorie'
})
export class FitredBycategoriePipe implements PipeTransform {

  transform(personne: Personne [],categorie: string)
  {
    return    personne.filter(per => per.categorie === categorie);
  }
}

<app-personnes *ngFor="let per of etudiants |fitredBycategorie : 'ingenieur' ; index as i; let first =
first; let last = last;
even as ev ; odd as odd;"  

(PersonneSelected)="OnPersonneSelected($event)" [personne] = "per"
[index] = "i +1" [class.is-first]="first" [class.is-last]="last"
[class.is-even]="ev" [class.is-odd]="odd"
[noImageTpl]="noImage"
/>
```

Angular Impure Pipes In Detail

Angular Local Template Querying In Depth:

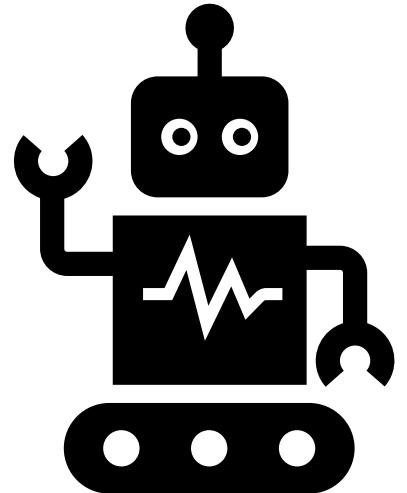
Angular View Child Decorator - How Does it Work?

@ViewChild est un décorateur dans Angular qui joue un rôle clé dans la gestion des interactions entre les composants.

@ViewChild permet à un composant parent d'accéder aux propriétés et méthodes d'un composant enfant ou d'une directive. Cela se fait en marquant une propriété de classe avec le décorateur `@ViewChild`, qui peut ensuite être utilisé pour accéder à des instances de composants enfants, des éléments DOM ou des directives dans le template du composant parent.

To Be Continued....

Projet pratique intégrant directives et pipes.



Services et Injection de Dépendance (8 heures)

Professeur Titulaire : Millimono Sory (PhD, AI)

Plan Du Cours



○ Concepts de services et injection de dépendance (DI).

○ Création de services pour la logique métier.

○ Intégration de services dans une application.

○ Exercices pratiques.

Dans Angular, les services et l'injection de dépendance sont des concepts fondamentaux qui jouent un rôle crucial dans le développement d'applications structurées et maintenables. Ces concepts aident à séparer la logique métier de la logique de présentation et à organiser le code de manière modulaire, ce qui facilite la gestion de projets complexes et améliore la testabilité.

Un **service** dans Angular est généralement une classe avec un objectif bien défini. Il est conçu pour effectuer une fonction spécifique en dehors d'un composant Angular pour favoriser la réutilisabilité, la modularité, et la séparation des préoccupations. Les services peuvent inclure des fonctions pour interagir avec une base de données, effectuer des calculs, ou gérer des opérations de données complexes.

La seule préoccupation d'un composant devrait être d'afficher des données et non de les gérer. Les services sont là où l'équipe Angular préconise de placer la logique métier et la gestion des données de l'application. Avoir une séparation claire entre la couche de présentation et les autres traitements de l'application augmente la réutilisabilité et la modularité.

Utilisations typiques des services :

Gestion des Données : Les services peuvent être utilisés pour gérer les opérations liées aux données, telles que la récupération des données, la persistance des informations ou l'interaction avec des API backend.

Logique Métier : Ils permettent de centraliser la logique métier de l'application, garantissant que les règles et les calculs sont localisés dans un seul endroit, plutôt que dispersés à travers plusieurs composants.

Partage de Données : Les services peuvent partager des informations entre des composants non liés, en agissant comme un magasin global ou un bus de données.

Fonctionnalités Utilitaires : Ils peuvent offrir des fonctions utilitaires, comme la journalisation, la validation des données, ou des tâches d'authentification.

Injection de Dépendance (DI)

L'injection de dépendance est un motif de conception logicielle utilisé pour rendre les classes moins dépendantes des implémentations de leurs dépendances.

Angular possède un système d'injection de dépendance puissant et efficace qui simplifie le développement en gérant la création d'instances de classes et en fournissant ces instances aux composants ou services qui en ont besoin.

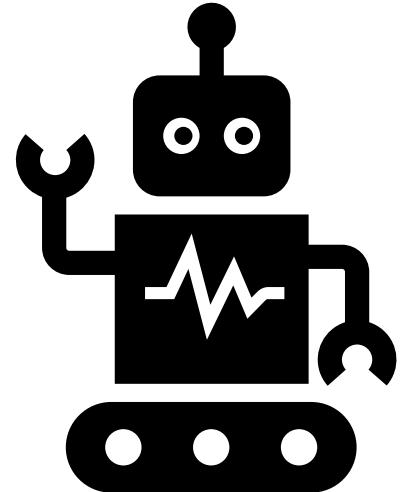
Comment fonctionne l'injection de dépendance dans Angular :

Injecteur : Angular utilise un système d'injecteurs qui fournit des dépendances à un composant ou à un service. Un injecteur crée les dépendances et les maintient en mémoire, les réutilisant au besoin.

Fournisseur : Les dépendances sont définies par des fournisseurs qui indiquent à l'injecteur comment obtenir ou créer une dépendance.

Décorateur @Injectable : Ce décorateur est utilisé pour décorer une classe de service, indiquant qu'elle peut avoir ses dépendances injectées par Angular.

Création de services pour la logique métier.



Créer un API qui sert les données des personnes (Personne[]) et les consommer à l'aide d'Angular avec HTTP :

- Un serveur backend pour héberger notre API.
- Un service Angular pour interagir avec cette API.
- Modification de nos données pour utiliser une API plutôt qu'une importation statique.

Étape 1 : Créer un serveur backend simple

On peut utiliser Node.js avec Express pour créer un serveur API simple qui retourne nos données.

Créez un nouveau dossier pour notre projet backend et initialisez un nouveau projet Node.js :

```
mkdir my-person-api  
cd my-person-api  
npm init -y  
npm install express
```

b. Ajoutez un fichier server.js qui sera notre serveur Express :

Étape 1 : Créer un serveur backend simple

b. Ajoutez un fichier `server.js` qui sera notre serveur Express :

```
const express = require('express');
const app = express();
const port = 3000;

const personnes = [
    // nos données ici comme dans votre question
];

app.get('/api/personnes', (req, res) => {
    res.json(personnes);
});

app.listen(port, () => {
    console.log(`Server running on http://localhost:${port}`);
});
```

`node server.js`

localhost:3000/api/personnes

The Angular HTTP Client - GET calls with Request Parameters

```
constructor( private http : HttpClient)
{
  console.log('AppComponent: Constructor');
}

ngOnInit() {
  // Called once the component is initialized.
  //console.log('AppComponent: OnInit');
  //  console.table(this.etudiants);
  //  this.SelectStudent(this.etudiants[0]);
  this.http.get('/api/personnes').
    subscribe(
      val => console.table(val)
    );
}
```

src\app\app.module.ts

```
imports: [
  BrowserModule,
  AppRoutingModule,
  HttpClientModule
],
```

Configurer un Proxy pour Angular

Créer un fichier **proxy.conf.json** à la racine de notre projet Angular avec le contenu suivant :

```
{  
  "/api/*": {  
    "target": "http://localhost:3000",  
    "secure": false,  
    "changeOrigin": true,  
    "logLevel": "debug"  
  }  
}
```

Modifiez ensuite notre fichier angular.json pour utiliser ce fichier de configuration de proxy lors du lancement du serveur de développement :

```
...  
"serve": {  
  "builder": "@angular-devkit/build-angular:dev-server",  
  "options": {  
    "browserTarget": "your-project-name:build",  
    "proxyConfig": "proxy.conf.json"  
  },  
...  
}
```

ng serve

```
etudiants: Personne[] = []; //  
Initialisation à un tableau vide
```

```
ngOnInit()  
{  
  this.http.get<any[]>('/api/personnes').subscribe(  
    data => {  
      this.etudiants = data;  
      console.log('Étudiants chargés:', data);  
    },  
    error => {  
      console.error('Erreur lors du chargement des étudiants:', error);  
    }  
  );  
}
```

```
ngOnInit() {  
  
  const params = new HttpParams()  
    .set("page", "1")  
    .set("pagesize", "10");  
  
  this.http.get<any[]>('/api/personnes',{params}).subscribe(  
    data => {  
      this.etudiants = data;  
      console.log('Étudiants chargés:', data);  
    },  
    error => {  
      console.error('Erreur lors du chargement des étudiants:', error);  
    }  
  );  
}
```

The Async Pipe - a Better way of passing Observable data to the View

```
etudiants: Personne[] = []; // Initialisation à un tableau vide  
etudiants$: Observable<Personne[]> | undefined;  
  
ngOnInit() {  
  
  const params = new HttpParams()  
    .set("page", "1")  
    .set("pagesize", "10");  
  
  this.etudiants$ = this.http.get<Personne[]>('/api/personnes',{params});  
}
```

```
<ng-container *ngIf="(etudiants$ | async) as etudiants">

  <app-personnes *ngFor="let per of (etudiants | fitredBycategorie : 'ingenieur') ; index as i;
  let first = first; let last = last;
  even as ev ; odd as odd;">
    (PersonneSelected)="OnPersonneSelected($event)" [personne] = "per"
    [index] = "i +1" [class.is-first]="first" [class.is-last]="last"
    [class.is-even]="ev" [class.is-odd]="odd"
    [noImageTpl]="noImage"
  />

</ng-container>
```

Angular Custom Services - The Injectable Decorator

ng generate service sservice/personnes

```
constructor( private http : HttpClient,  
    private personneService : PersonnesService  
)  
{  
    console.log('AppComponent: Constructor');  
}
```

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

L'injection de dépendance (DI) est un modèle de conception puissant utilisé en programmation pour augmenter la modularité et la testabilité des applications. Bien que nous puissions souvent atteindre les mêmes objectifs fonctionnels en important directement des modules ou des classes, l'injection de dépendance offre plusieurs avantages significatifs qui peuvent améliorer la structure et la qualité de notre code.

Avantages de l'Injection de Dépendance par rapport à l'Importation Directe

1. Découplage des composants :

- 1. Importation Directe :** Lorsque vous importez directement des dépendances, chaque composant est étroitement lié aux implémentations spécifiques des modules ou des services qu'il utilise. Cela rend le composant dépendant non seulement de l'interface, mais aussi de l'implémentation.
- 2. Injection de Dépendance :** DI permet à vos composants de rester complètement indépendants des implémentations spécifiques des dépendances qu'ils utilisent. Les composants dépendent seulement d'une interface ou d'un contrat, ce qui signifie que n'importe quelle implémentation respectant ce contrat peut être utilisée sans modifier le composant.

2. Facilitation des tests :

- 1. Importation Directe :** Tester des composants qui importent directement leurs dépendances peut être difficile car il est souvent nécessaire de mettre en place tout l'environnement nécessaire pour ces dépendances.
- 2. Injection de Dépendance :** DI facilite le mocking et le stubbing des dépendances dans les tests unitaires. Vous pouvez fournir des implémentations factices ou simulées de dépendances qui ne contiennent que la logique nécessaire pour les tests, rendant les tests plus simples et plus rapides.

1.Gestion de la configuration et des instances :

- 1. Importation Directe** : La gestion des instances et configurations spécifiques peut devenir complexe et dispersée à travers l'application.
- 2. Injection de Dépendance** : DI permet de configurer et de gérer les dépendances de manière centralisée. Par exemple, vous pouvez configurer une instance de service comme singleton ou avec une portée plus étroite (comme par composant dans Angular), ce qui est difficile à gérer avec des importations directes.

2.Évolution et maintenance du code :

- 1. Importation Directe** : Les changements dans les dépendances peuvent nécessiter des modifications importantes dans tous les composants qui les utilisent.
- 2. Injection de Dépendance** : DI permet de changer les implémentations des dépendances sans modifier les composants qui les consomment. Cela simplifie les mises à jour et les améliorations des logiciels.

3.Réutilisation du code :

- 1. Importation Directe** : La réutilisation de composants dans différents contextes ou applications peut être limitée par des dépendances spécifiques codées en dur.
- 2. Injection de Dépendance** : Comme les composants ne créent pas ou ne recherchent pas activement leurs dépendances, ils sont plus faciles à réutiliser dans différents contextes.

L'injection de dépendance améliore la modularité, la flexibilité et la testabilité des applications en réduisant le couplage entre les composants et leurs dépendances. Cela facilite la maintenance, la mise à jour et la réutilisation des composants à travers les grandes bases de code et les équipes de développement. En fin de compte, DI contribue à une architecture logicielle plus propre et plus robuste.

Angular Custom Service - Fetching Data

```
import { HttpClient, HttpParams } from '@angular/common/http';
import { Injectable } from '@angular/core';
import { Personne } from '../personne';
import { Observable } from 'rxjs';

@Injectable({
  providedIn: 'root'
})
export class PersonnesService {

  constructor( private http : HttpClient) { }

  loadCourse():Observable<Personne[]>
  {
    const params = new HttpParams()
      .set("page", "1")
      .set("pagesize", "10");

    return this.http.get<Personne[]>('/api/personnes',{params});
  }
}
```

```
constructor(
  private personneService : PersonnesService
)
{
  console.log('AppComponent: Constructor');
}

ngOnInit() {
  this.etudiants$ =
  this.personneService.loadCourse();
}
```

Angular Custom Service - Data Modification with an HTTP PUT

ETAT ACTUEL :

```
<button appTooltip="{{  
personne?.address}}"  
(click)="OnpersonneViewed()">D  
etails sur la  
personne</button>
```

```
@Output()  
PersonneSelected = new EventEmitter<Personne>();  
  
OnpersonneViewed()  
{  
    console.log("btn clické");  
    this.PersonneSelected.emit(this.personne)  
};  
}
```

```
<app-personnes *ngFor="let per of (etudiants  
|fitredBycategorie : 'ingenieur') ; index as i;  
let first = first; let last = last;  
even as ev ; odd as odd;"  
(PersonneSelected)="OnPersonneSelected($event)"  
[personne] = "per"  
[index] = "i +1" [class.is-  
first]="first" [class.is-last]="last"  
[class.is-even]="ev" [class.is-odd]="odd"  
[noImageTpl]="noImage"  
/>
```

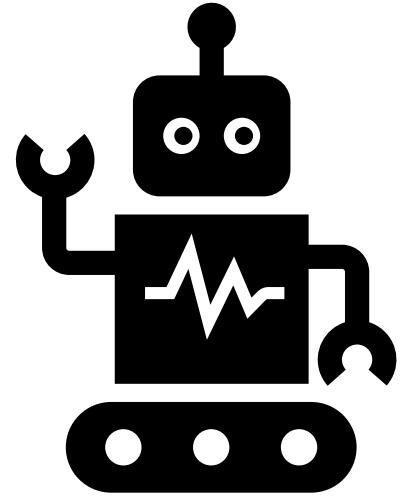
```
OnPersonneSelected(personne:Personne)  
{  
    console.log('click de card' , personne);  
}
```

```
OnPersonneSelected(<app-personnes *ngFor="let per of etudiants | fitredBycategorie : 'ingenieur' ; index as i > [class.is-first]= "first" [class.is-last]= "last"; @Output() PersonneSelected = new EventEmitter<Personne>();) [personne] = "per" [index] = "i +1" [class.is-even]= "ev" [class.is-odd]= "odd" [noImageTpl]= "noImage" />
```

```
OnPersonneSelected(personne:Personne)
{
    //console.log('click de card' , personne);
    this.personneService.savePersonne(personne);

}
```

Angular Dependency Injection In Depth.



Introduction to the Angular Dependency Injection System

```
@Injectable({  
})  
export class PersonnesService {  
  
    constructor( private http : HttpClient) { }  
  
    loadCourse():Observable<Personne[]>  
{  
        const params = new HttpParams()  
        .set("page", "1")  
        .set("pagesize", "10");  
  
        return this.http.get<Personne[]>('/api/personnes',{params});  
    }  
}
```

Angular DI - Understanding Providers and Injection Tokens

```
function PersonneProvider():PersonnesService
{
  return new PersonnesService();

}

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  // styleUrls: ['./app.component.css'
})
```

```
function PersonneProvider(http:HttpClient):PersonnesService
{
  return new PersonnesService( http);
}
```

Angular DI - Understanding Providers and Injection Tokens

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  providers: [
    {
      provide:
    }
  ]
// styleUrls: ['./app.component.css']
})
```

Angular DI - Understanding Providers and Injection Tokens

```
const PERSONNE_SERVICE = new InjectionToken<PersonnesService>('PERSONNE_SERVICE');

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  providers: [
    {provide:PERSONNE_SERVICE}
    // styleUrls: ['./app.component.css']
})
```

Angular DI - Understanding Providers and Injection Tokens

```
const PERSONNE_SERVICE = new InjectionToken<PersonnesService>('PERSONNE_SERVICE');

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  providers: [
    {provide:PERSONNE_SERVICE,useFactory : PersonneProvider,
     deps:[HttpClient]
    }
  ]
  // styleUrls: ['./app.component.css'
})
```

Angular DI - Understanding Providers and Injection Tokens

```
constructor(@Inject(PERSONNE_SERVICE) private personneService : PersonnesService )  
{  
  console.log('AppComponent: Constructor');  
}
```

Angular DI - Understanding Simplified Provider Configuration

```
function PersonneProvider(http:HttpClient):PersonnesService
{
    return new PersonnesService( http );
}

const PERSONNE_SERVICE = new InjectionToken<PersonnesService>('PERSONNE_SERVICE');

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  providers: [
    {
      provide:PERSONNE_SERVICE,
      useClass : PersonnesService,
    }
  ]
  // styleUrls: ['./app.component.css'
})
```

Angular DI - Understanding Simplified Provider Configuration

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  providers: [PersonnesService

]
// styleUrls: ['./app.component.css'
})
```

Understanding Hierarchical Dependency Injection

L'injection de dépendance (ID) elle-même est une technique de design qui permet à un composant de recevoir ses dépendances plutôt que de les créer lui-même. Cela facilite le découplage des composants, rendant ainsi l'application plus modulaire et facile à tester.

Fonctionnement de l'Injection de Dépendance Hiérarchique

Dans une structure hiérarchique, Angular utilise un système d'injecteurs qui peut être imaginé comme un arbre où chaque injecteur peut avoir un parent et peut-être des enfants. Chaque niveau de l'arbre peut définir ses propres providers (fournisseurs de dépendance) ou hériter de ceux de son parent.

Understanding Hierarchical Dependency Injection

Niveaux de l'Arbre :

Module Root : Au sommet de la hiérarchie, vous avez l'injecteur racine qui est configuré par le module racine de l'application (typiquement AppModule).

Modules de Fonctionnalités : Chaque module de fonctionnalité peut avoir son propre injecteur qui fournit des instances spécifiques aux composants dans ce module.

Composants : Chaque composant peut également définir ses propres providers, permettant ainsi de limiter la portée des services à ceux spécifiquement nécessaires pour le composant.

Propagation :

Lorsqu'un composant demande une dépendance, Angular commence la recherche à partir de l'injecteur du composant, puis remonte dans la hiérarchie jusqu'à ce qu'il trouve un provider correspondant pour la dépendance demandée. Si aucun provider n'est trouvé, une erreur est générée (sauf configuration spéciale).

Understanding Hierarchical Dependency Injection

Avantages de l'Injection de Dépendance Hiérarchique

Modularité : Chaque module ou composant peut gérer ses propres dépendances sans affecter le reste de l'application.

Réutilisation : Les services communs peuvent être fournis à un niveau supérieur et être réutilisés par tous les enfants de ce niveau.

Encapsulation : Les services peuvent être encapsulés dans des modules ou des composants, évitant les conflits et les dépendances non désirées.

Flexibilité dans les tests : Il est plus facile de remplacer ou de mocker des services à différents niveaux pour les tests.

Supposons que vous ayez un service AuthService pour gérer l'authentification, que vous souhaitez disponible dans toute l'application, mais avec des comportements spécifiques à certains composants. Vous pourriez fournir AuthService globalement dans un module, et le redéfinir dans des composants spécifiques qui nécessitent un comportement personnalisé.

Understanding Hierarchical Dependency Injection

```
@Injectable({
  providedIn: 'root'
})
export class AuthService {

  authenticate() {
    console.log("Global
authentication logic");
  }
}
```

```
@Component({
  selector: 'app-special-component',
  templateUrl: './special-component.component.html',
  providers: [AuthService] // Redéfinit le service ici
})
export class SpecialComponent implements OnInit {
  constructor(private authService: AuthService) {}

  ngOnInit() {
    this.authService.authenticate = this.customAuthenticate;
  }

  private customAuthenticate() {
    console.log("Custom authentication logic for special
component");
  }
}
```

Understanding Angular Tree-Shakeable Providers

Le concept de "**Tree-Shakeable Providers**" dans Angular fait partie de l'optimisation du framework pour améliorer la performance des applications en permettant un meilleur "tree shaking". "Tree shaking" est un terme utilisé en développement web pour décrire le processus d'élimination du code non utilisé dans les fichiers finaux d'une application lors du processus de build, ce qui réduit la taille du bundle final et améliore le temps de chargement de l'application.

```
@Injectable({
  providedIn: 'root',
  useFactory: (http: HttpClient) => new
PersonnesService(http),
  deps: [HttpClient],
})
export class PersonnesService {
  constructor(private http: HttpClient) {}
```

```
@Injectable({
  providedIn: 'root',
  useClass: PersonnesService,
})
export class PersonnesService {
  constructor(private http:
HttpClient) {}
```

Understanding Angular Tree-Shakeable Providers

Qu'est-ce qu'un Provider "Tree-Shakeable" ?

Un provider "tree-shakeable" dans Angular est un service qui peut être inclus dans le bundle final seulement s'il est effectivement utilisé quelque part dans l'application. Cela contraste avec les providers non tree-shakeable, qui sont inclus dans le bundle final qu'ils soient utilisés ou non, car leur inclusion est décidée lors de la compilation.

Comment Créer un Provider Tree-Shakeable ?

Pour qu'un service soit tree-shakeable, il doit être fourni dans le décorateur `@Injectable()` du service lui-même en utilisant `providedIn`. Angular offre plusieurs options pour `providedIn`:

providedIn: 'root' : Le service est disponible globalement dans l'application, et Angular l'inclura dans le bundle final seulement si le service est utilisé quelque part dans l'application.

providedIn: SomeModule : Le service est disponible uniquement dans le module spécifié et ses dépendants. De même, le service ne sera inclus dans le bundle que s'il est utilisé.

providedIn: 'any' : Nouveauté depuis Angular 9, cette option permet au service d'être un singleton au niveau de chaque module qui l'importe, tout en étant tree-shakeable.

Understanding Angular Tree-Shakeable Providers

Avantages des Providers Tree-Shakeable

Réduction de la Taille du Bundle : En éliminant les services non utilisés du bundle final, les applications deviennent plus légères et chargent plus rapidement.

Modularité Améliorée : Permet une meilleure séparation et encapsulation du code en fournissant des services à des scopes plus ciblés.

Gestion des Dépendances Optimisée : Simplifie la gestion des dépendances en évitant la nécessité de gérer explicitement les providers dans les modules.

```
@Injectable({
  providedIn: 'root',
  useFactory: (http: HttpClient) => new
PersonnesService(http),
  deps: [HttpClient],
})
export class PersonnesService {
  constructor(private http: HttpClient) {}
```

```
@Injectable({
  providedIn: 'root',
  useClass: PersonnesService,
})
export class PersonnesService
{
  constructor(private http:
HttpClient) {}
```

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

Angular DI - Injection Tokens In Detail

Les Injection Tokens dans Angular sont un concept avancé de l'Injection de Dépendance (DI) qui permet de fournir et d'injecter des valeurs ou des dépendances sans dépendre directement d'une classe concrète. Ce mécanisme est particulièrement utile pour rendre votre application plus flexible et pour faciliter les tests en isolant les dépendances.

Un Injection Token est un objet qui sert de clé pour le système d'Injection de Dépendance d'Angular. Avec les Injection Tokens, vous pouvez injecter des valeurs comme des chaînes, des nombres, des configurations, des interfaces, et d'autres objets qui ne peuvent pas être injectés classiquement.

Angular DI - Injection Tokens In Detail

Pourquoi Utiliser des Injection Tokens?

Injection de types primitifs : Angular ne permet pas l'injection directe de types primitifs (comme string, number, etc.) parce qu'ils ne fournissent pas les métadonnées nécessaires à l'injecteur. Les Injection Tokens résolvent ce problème.

Dépendances d'interface : TypeScript transpile les interfaces en JavaScript de sorte qu'elles ne sont pas disponibles au runtime. Les Injection Tokens permettent d'injecter des implémentations basées sur des interfaces.

Configuration flexible : Vous pouvez utiliser des Injection Tokens pour fournir des configurations qui peuvent être modifiées sans avoir besoin de modifier les composants qui les utilisent.

Meilleure testabilité : Les Injection Tokens facilitent le mocking des dépendances dans les tests unitaires.

Angular DI - Injection Tokens In Detail

Créons dans app par exemple : config.ts

```
import { Inject, InjectionToken } from
'@angular/core';

export interface AppConfig {
  apiUrl: string;
  PersonneCacheSize: number;
}

export const CONFIG_TOKEN = new
InjectionToken<AppConfig>('CONFIG_TOKEN');

export const APP_CONFIG: AppConfig = {
  apiUrl: 'http://localhost:9000',
  PersonneCacheSize: 10,
};
```

Dans app.component.ts

```
@Component({
  selector: 'app-root',
  templateUrl:
  './app.component.html',
  providers: [{ provide:
  CONFIG_TOKEN, useFactory: () =>
  APP_CONFIG }],
})
export class AppComponent implements
OnInit {
```

```
constructor(private personneService:
PersonnesService, @Inject(CONFIG_TOKEN) private
config: AppConfig
) {
  console.log('AppComponent: Constructor
config');
  console.log(config);
}
```

Angular DI - Injection Tokens In Detail

Créons dans app par exemple : config.ts

```
import { Inject, InjectionToken } from
'@angular/core';

export interface AppConfig {
  apiUrl: string;
  PersonneCacheSize: number;
}

export const CONFIG_TOKEN = new
InjectionToken<AppConfig>('CONFIG_TOKEN');

export const APP_CONFIG: AppConfig = {
  apiUrl: 'http://localhost:9000',
  PersonneCacheSize: 10,
};
```

Dans app.component.ts

```
@Component({
  selector: 'app-root',
  templateUrl:
  './app.component.html',
  providers: [{ provide:
  CONFIG_TOKEN, useValue: APP_CONFIG}],
})
export class AppComponent implements
OnInit {
```

```
constructor(private personneService:
PersonnesService, @Inject(CONFIG_TOKEN) private
config: AppConfig
) {
  console.log('AppComponent: Constructor
config');
  console.log(config);
}
```

Angular DI - Injection Tokens Tree-Shakeable

Créons dans app par exemple : config.ts

```
import { Inject, InjectionToken } from
'@angular/core';

export interface AppConfig {
  apiUrl: string;
  PersonneCacheSize: number;
}

export const CONFIG_TOKEN = new
InjectionToken<AppConfig>('CONFIG_TOKEN', {
  providedIn: 'root',
  factory: () => APP_CONFIG,
});

export const APP_CONFIG: AppConfig = {
  apiUrl: 'http://localhost:9000',
  PersonneCacheSize: 10,
};
```

Dans app.component.ts

```
@Component({
  selector: 'app-root',
  templateUrl:
  './app.component.html',
})

export class AppComponent implements
OnInit {
```

```
constructor(private personneService:
PersonnesService, @Inject(CONFIG_TOKEN) private
config: AppConfig
) {
  console.log('AppComponent: Constructor
config');
  console.log(config);
}
```

Angular DI Decorators - Optional, Self, SkipSelf

Routage (8 heures)

Professeur Titulaire : Millimono Sory (PhD, AI)

Plan Du Cours



- Concepts de services et injection de dépendance (DI).
- Création de services pour la logique métier.
- Intégration de services dans une application.
- Exercices pratiques.
-

Dans Angular, le routage est un aspect crucial qui permet de naviguer entre différentes vues ou composants dans une application web. Angular Router est un module puissant qui gère la navigation de façon intuitive et efficace, en rendant l'application riche et interactive.

```
ng g c PersonnesDetails/details.personne
```

Configuration du Routage

Le système de routage dans Angular repose sur une configuration déclarative. Vous définissez les routes dans un module (généralement nommé **AppRoutingModule**), en associant des chemins d'URL à des composants.

```
ng g c ListesPersonnes / listes.personnes
```

Le routage dans Angular est un aspect crucial pour construire des applications web à page unique (Single Page Applications, SPAs). Il permet de naviguer entre différentes vues ou composants de l'application sans recharger la page.

1. Configuration de Base

Pour utiliser le routage dans Angular, vous devez d'abord configurer le RouterModule:

Importez le RouterModule :

Ajoutez RouterModule dans les imports de votre module principal (habituellement AppModule).

Créer des routes

src\app\app-routing.module.ts

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { PersonnesComponent } from './personnes/personnes.component';
import { DetailsPersonneComponent } from
'./PersonnesDetails/details.personne/details.personne.component';

const routes: Routes = [];

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

Créer des routes

src\app\app-routing.module.ts

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { PersonnesComponent } from './personnes/personnes.component';
import { DetailsPersonneComponent } from
'./PersonnesDetails/details.personne/details.personne.component';

const routes: Routes = [
  {path: 'personnes', component:PersonnesComponent},
  {path: 'detailspersonnes/:id', component:DetailsPersonneComponent},
  {path: '', redirectTo: 'personnes', pathMatch : 'full'},
];

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

La balise <router-outlet>

Navigation

La navigation entre les pages se fait soit programmablement à l'aide du service Router, soit par des liens définis dans les templates avec la directive routerLink. Exemple d'utilisation de routerLink dans un template HTML :

```
<nav>
  <div class="nav-wrapper">
    <a href="#" class="brand-logo">Logo</a>
    <ul id="nav-mobile" class="right hide-on-med-and-down">
      <li><a routerLink="/accueil">Accueil</a></li>
      <li><a routerLink="/personnes">Personnes</a></li>
      <li><a routerLink="/contact">Contact</a></li>
    </ul>
  </div>
</nav>

<router-outlet></router-outlet>
```

Le `<router-outlet>` est un élément fondamental du système de routage dans Angular. C'est une directive qui fonctionne comme un placeholder (emplacement réservé) où Angular peut charger dynamiquement les composants de la route active. L'utilisation de `<router-outlet>` est essentielle dans toute application Angular qui utilise le routage pour naviguer entre différentes vues ou pages.

Fonctionnement de `<router-outlet>`

Lorsque vous configurez les routes de votre application Angular et que vous démarrez la navigation entre ces routes, Angular utilise `<router-outlet>` pour afficher les composants associés à chaque route. Chaque fois qu'une nouvelle route est activée, Angular détruit le composant de la route précédente qui était chargé dans le `<router-outlet>` et crée et insère le composant de la nouvelle route à la même place.

Placement de `<router-outlet>`

Vous placez `<router-outlet>` dans le template de votre application là où vous souhaitez que le contenu des composants de vos routes soit affiché. Typiquement, il est placé dans le template du composant principal (souvent `AppComponent`) de l'application.

La balise <router-outlet>

Navigation Programmée

Vous pouvez également naviguer programmablement en utilisant le service Router d'Angular.

```
this.router.navigate(['/home']);
```

Route Guards

Les Route Guards sont utilisés pour contrôler l'accès à certaines routes en fonction de conditions spécifiques comme l'authentification ou les permissions. Angular offre plusieurs types de guards, tels que CanActivate, CanDeactivate, et Resolve. Par exemple, CanActivate peut être utilisé pour vérifier si un utilisateur est connecté avant de lui permettre d'accéder à une route spécifique .

La balise <router-outlet>

Route Parameters

Les routes peuvent également inclure des paramètres, utiles pour passer des valeurs spécifiques comme des identifiants d'entité.



```
{ path: 'profile/:id', component: ProfileComponent }
```



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

```
constructor(private route: ActivatedRoute) {
  this.route.params.subscribe(params => {
    console.log(params['id']); // Affiche l'ID passé à la route
  });
}
```



listes.personnes.component.html

```
<ng-container *ngIf="(etudiants$ | async) as etudiants">

<div class="container">
  <div class="row">

    <app-personnes *ngFor="let per of (etudiants) ; index as i; let first = first; let last = last;
even as ev ; odd as odd;" (PersonneSelected)="OnPersonneSelected($event)" [personne] = "per"
[index] = "i +1" [class.is-first]="first" [class.is-last]="last"
[class.is-even]="ev" [class.is-odd]="odd"
[noImageTmp1]="noImage"
class="col s12 m6 l4"
/>
  </div>
</div>

</ng-container>

<ng-template #noImage let-personneName="name">

  <p> {{personneName}} has no Image Available yet</p>
  

</ng-template>
```

listes.personnes.component.ts

```
export class ListesPersonnesComponent {  
  
  etudiants$: Observable<Personne[]> | undefined; // Initialisation à un tableau vide  
  
  constructor(private personneService : PersonnesService )  
{  
  console.log('AppComponent: Constructor');  
}  
  
  SelectStudent(Student: Personne) {  
    console.log(`Vous avez Mr ${Student.lastName}`);  
}  
  
  ngOnInit() {  
  
    this.etudiants$ = this.personneService.loadCourse();  
}  
  
  OnPersonneSelected(personne:Personne)  
{  
  //console.log('click de card' , personne);  
  this.personneService.savePersonne(personne);  
}  
}
```

details.personne.component.html

```
<div *ngIf="etudiant$ | async as etudiant" class="row">
  <div class="col s12 m8 offset-m2">
    <h2 class="header center">{{ etudiant }} {{ etudiant.lastName }}</h2>
    <div class="card horizontal hoverable">
      <div class="card-image">
        <img [src]="etudiant.picture" [alt]="etudiant.firstName">
      </div>
      <div class="card-stacked">
        <div class="card-content">
          <table class="bordered striped">
            <tbody>
              <tr>
                <td>Prénom</td>
                <td><strong>{{ etudiant.firstName }}</strong></td>
              </tr>
              <tr>
                <td>Nom</td>
                <td><strong>{{ etudiant.lastName }}</strong></td>
              </tr>
              <tr>
                <td>Âge</td>
                <td><strong>{{ etudiant.age }}</strong></td>
              </tr>
              <tr>
                <td>Email</td>
                <td>{{ etudiant.email }}</td>
              </tr>
              <tr>
                <td>Adresse</td>
                <td>{{ etudiant.address }}</td>
              </tr>
            </tbody>
          </table>
        </div>
      </div>
    </div>
  </div>
</div>
```

details.personne.component.ts

```
export class DetailsPersonneComponent
{
  etudiants$: Observable<Personne[]> | undefined; // Initialisation à un tableau vide
  etudiant$: Observable<Personne | undefined> | undefined; // Observable pour une seule personne

  constructor(private router: ActivatedRoute, private personneService : PersonnesService,
    private route: Router)
  {}

  ngOnInit() {
    console.log('Initialisation de la page des détails');
    // Récupération de l'ID depuis les paramètres de route
    const id = this.router.snapshot.paramMap.get('id');
    if (id) {
      this.etudiant$ = this.personneService.loadCourse().pipe(
        map(personnes => personnes.find(p => p.id === +id)) // Utiliser '+' pour convertir 'id' en nombre si nécessaire
      );
    }
  }

  goBack()
  {
    this.route.navigate(['listespersonnes']);
  }
}
```

personne.component.html

```
<button (click)="OnpersonneViewed(personne?.id)">Details sur la personne</button>
```

personne.component.ts

```
OnpersonneViewed(id: number | undefined) {
  if (id !== undefined) {
    console.log('btn clické');
    this.PersonneSelected.emit(this.personne);
    this.router.navigate(['detaislpersonnes', id]);
  } else {
    console.log('ID est indéfini');
    // Gérer le cas où l'ID est indéfini, par exemple, afficher un message d'erreur
  }
}
```


Utilisations Avancées <router-outlet>

Routage imbriqué

Angular permet le routage imbriqué, ce qui signifie que vous pouvez avoir des <router-outlet> dans les composants chargés eux-mêmes. Cela est utile pour des applications avec des sous-sections complexes. Chaque sous-route peut être configurée pour charger des composants dans le <router-outlet> du composant parent, créant ainsi une hiérarchie de vues. disons que nous voulons ajouter des sous-routes au **DetailsPersonneComponent**

```
const routes: Routes = [
  { path: 'listespersonnes', component: ListesPersonnesComponent },
  {
    path: 'detailspersonnes/:id',
    component: DetailsPersonneComponent,
    children: [
      {
        path: 'info',
        component: PersonneInfoComponentComponent, // Composant pour afficher des informations supplémentaires
      },
      {
        path: 'contacts',
        component: PersonneContactsComponentComponent, // Composant pour afficher les contacts
      },
    ],
  },
  { path: '', redirectTo: 'listespersonnes', pathMatch: 'full' },
  // { path: '**', component: NotFoundComponent } // Wildcard route for a 404 page
];
```

ng g c PersonnelInfoComponent/PersonnelInfoComponent

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

@Component({
  selector: 'app-personne-info-component',
  template: `
    <div *ngIf="etudiant as etudiant">
      <h4>Informations Supplémentaires</h4>
      <p>Date de naissance: {{ etudiant.created | date }}</p>
      <p>Matricule: {{ etudiant.id }}</p>
    </div>
  `,
  styleUrls: ['./personne-info-component.component.css'],
})
export class PersonnelInfoComponentComponent {
  etudiant: Personne | undefined;
}
```

Ce composant attend que l'objet Personne soit passé comme Input. Il affiche la date de naissance et le matricule de l'étudiant.

ng g c PersonneContactsComponent/PersonneContactsComponent

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

@Component({
  selector: 'app-personne-contacts-component',
  template: `
    <div *ngIf="etudiant as etudiant">
      <h4>Contacts</h4>
      <p>Téléphone: {{ etudiant.phone }}</p>
      <p>Email: {{ etudiant.email }}</p>
    </div>
  `,
  styleUrls: ['./personne-contacts-component.component.css']
})
export class PersonneContactsComponentComponent {
  etudiant: Personne | undefined;
}
```

Ce composant, similaire au précédent, affiche le téléphone et l'email de l'étudiant, attendus en tant que Input.

```
constructor(  
    private route: ActivatedRoute,  
    private personnesService: PersonnesService  
) {}  
  
ngOnInit() {  
    this.route.parent!.params.pipe(  
        switchMap((params) => {  
            const id = +params['id'];  
            return this.personnesService.loadCourse().pipe(  
                map((personnes) => {  
                    const foundPersonne = personnes.find((p) => p.id === id);  
                    return foundPersonne;  
                })  
            );  
        })  
.subscribe(  
    (etudiant) => {  
        this.etudiant = etudiant;  
    },  
    (error) => {  
        console.error(  
            "Erreur lors de la récupération des données de l'étudiant",  
            error  
        );  
    }  
);  
}
```

```
<div class="card-action">
  <a (click)="goBack()">Retour</a>

  <a [routerLink]="['/detailspersonnes', etudiant.id, 'contacts']">Voir les
  Contacts</a>
  <a [routerLink]="['/detailspersonnes', etudiant.id, 'info']">Voir les Infos</a>

</div>
<router-outlet></router-outlet>  <!-- Ici Angular chargera les composants enfants -->

</div>

</div>
</div>

</div>
<h4 *ngIf="!(etudiant$ | async)" class="center">Aucune personne à afficher !</h4>
```

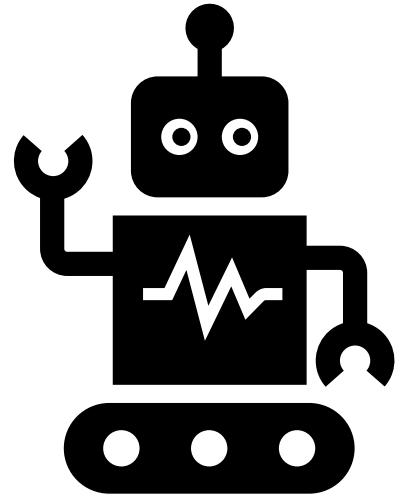
```
<div *ngIf="etudiant$ | async as etudiant; else noData" class="row">
  <div class="col s12 m10 offset-m1 l8 offset-l2">
    <div class="card horizontal hoverable">
      <div class="card-image">
        
      </div>
      <div class="card-stacked">
        <div class="card-content">
          <h4 class="header center-align">{{ etudiant.firstName }} {{ etudiant.lastName }}</h4>
          <table class="striped highlight responsive-table">
            <tbody>
              <tr><td>Prénom:</td><td>{{ etudiant.firstName }}</td></tr>
              <tr><td>Nom:</td><td>{{ etudiant.lastName }}</td></tr>
              <tr><td>Âge:</td><td>{{ etudiant.age }}</td></tr>
              <tr><td>Email:</td><td>{{ etudiant.email }}</td></tr>
              <tr><td>Adresse:</td><td>{{ etudiant.address }}</td></tr>
            </tbody>
          </table>
        </div>
        <div class="card-action">
          <a class="waves-effect waves-teal btn-flat" (click)="goBack()">Retour</a>
          <a class="waves-effect waves-teal btn-flat" [routerLink]="['/detailspersonnes', etudiant.id, 'contacts']">Voir les Contacts</a>
          <a class="waves-effect waves-teal btn-flat" [routerLink]="['/detailspersonnes', etudiant.id, 'info']">Voir les Infos</a>
        </div>
      </div>
    </div>
  </div>
<ng-template #noData><h4 class="center-align">Aucune personne à afficher !</h4></ng-template>
<router-outlet></router-outlet> <!-- Angular chargera les composants enfants ici -->
```

Utilisations Avancées <router-outlet>

Routage auxiliaire

Vous pouvez également utiliser plusieurs <router-outlet> nommés dans une même application pour réaliser ce qu'on appelle le routage auxiliaire. Cela permet d'afficher plusieurs composants en même temps à des endroits différents de votre application. Par exemple, vous pourriez vouloir afficher un panneau latéral de dialogue en plus de la vue principale, chacun dans son propre <router-outlet>.

Angular Modules.



Les modules Angular sont un concept central dans le développement avec Angular. Ils permettent de grouper des composants, des directives, des pipes, et des services qui sont liés à des fonctionnalités spécifiques, facilitant ainsi l'organisation du code, la réutilisabilité des composants et l'optimisation de l'application. Angular utilise un système modulaire qui s'appuie sur ES6 modules, mais il introduit aussi ses propres décorateurs pour fournir des fonctionnalités supplémentaires spécifiques à l'écosystème Angular.

Définition d'un Module Angular

Un module Angular est défini par une classe décorée avec `@NgModule`. Ce décorateur accepte un objet de configuration qui décrit comment compiler et lancer l'application. Voici les principaux champs de cet objet :

declarations : Liste les composants, directives et pipes qui appartiennent à ce module. Les éléments déclarés ici peuvent être utilisés dans le template de n'importe quel composant du même module.

imports : Permet d'inclure d'autres modules dans le module actuel, rendant leurs déclarations disponibles pour les composants du module actuel.

providers : Liste les services que ce module contribue à l'injecteur global (ou scope de ce module si le service est configuré avec `providedIn` comme 'root' ou le module lui-même).

bootstrap : Déclare le composant racine qui est utilisé lors du démarrage du module; utilisé principalement dans le module racine.

exports : Liste les composants, directives et pipes que vous voulez rendre accessibles aux composants d'autres modules.

Chargement Paresseux (Lazy Loading)

Le chargement paresseux est une technique qui permet de charger des modules de l'application sur demande, au lieu de les charger tous au démarrage. Cela peut considérablement améliorer le temps de démarrage initial de l'application. Angular rend cela facile avec le routeur, qui peut être configuré pour charger des modules uniquement lorsque l'utilisateur navigue vers leurs routes associées.

```
ng generate module Personnes --routing
```

L'option --routing crée également un fichier de routage pour le module, ce qui est utile pour le chargement paresseux.

src\app\app.module.ts

```
imports: [
  BrowserModule,
  AppRoutingModule,
  HttpClientModule,
  PersonnesModulesModule,
],
```

Formulaires (8 heures)

Professeur Titulaire : Millimono Sory (PhD, AI)

Plan Du Cours



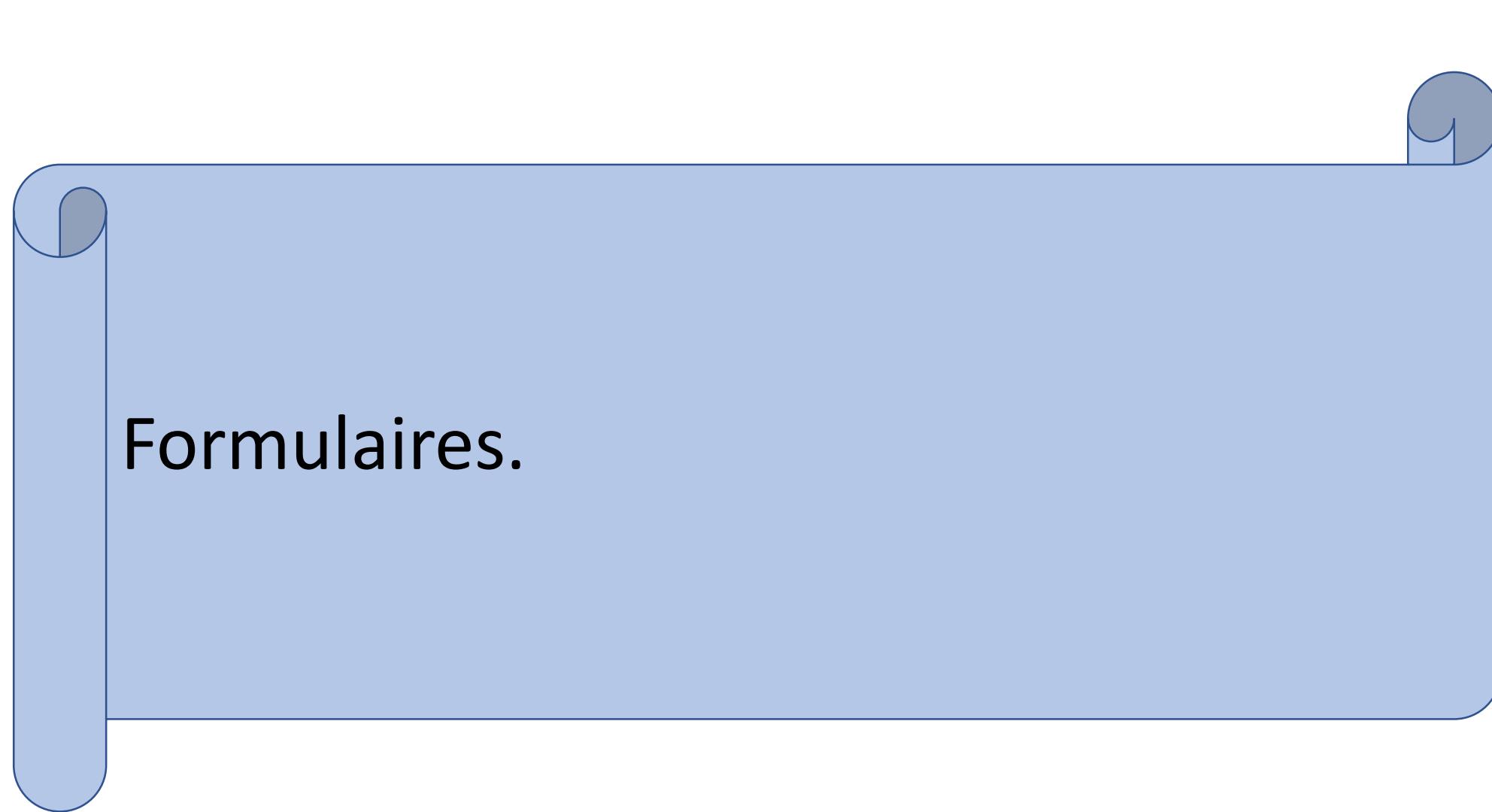
○ **Formulaires réactifs : création et validation.**

○ **Formulaires pilotés par le template.**

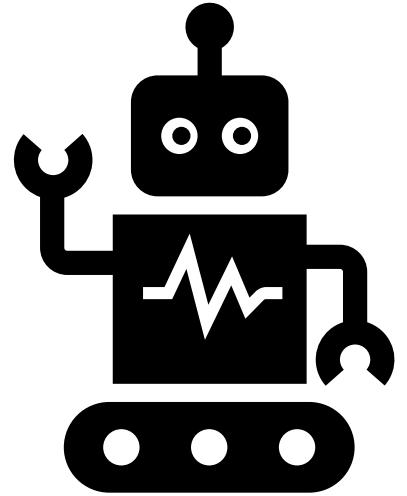
○ **Projet pratique sur les formulaires.**

○ **Projet pratique sur les formulaires.**



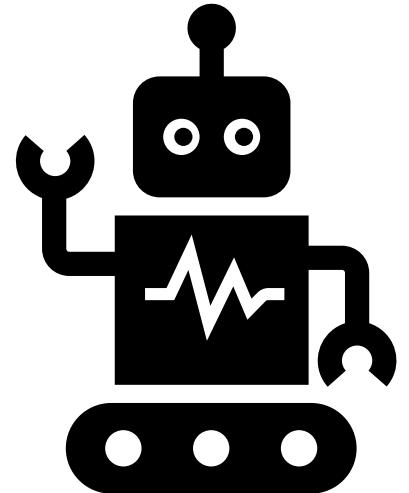


Formulaires.



Dans Angular, la gestion des formulaires est une partie fondamentale qui permet aux développeurs de construire des interfaces utilisateur interactives et efficaces pour la collecte de données. Angular offre deux approches principales pour gérer les formulaires : **les formulaires réactifs** et **les formulaires basés sur des templates**.

1. Formulaires Basés sur des Templates (Template-driven Forms).



Les formulaires basés sur des templates dans Angular représentent une approche simplifiée et déclarative pour la création et la gestion de formulaires. Cette méthode utilise principalement le template pour définir la logique et la structure du formulaire, en tirant parti de directives spécifiques d'Angular telles que **ngModel** pour lier les données **bidirectionnellement** entre le modèle **TypeScript** du composant et **la vue HTML**.

Avantages :

Facilité d'Utilisation : Moins de code à écrire car beaucoup de la logique est automatisée par Angular dans le template.

Rapidité de Développement : Plus rapide à mettre en place pour les formulaires simples.

Support Direct dans le Template : Intégration directe avec les éléments HTML sans nécessiter de logique supplémentaire dans le composant TypeScript.

Comment ça marche ? :

Vous utilisez des directives comme **ngModel** et utilisez le nom du formulaire pour lier des données de champs spécifiques. Les validations sont définies via des attributs dans le template.

Introduction to Template Driven Forms - the ngForm Directive

Use angular/material

ng add @angular/material

```
import { MatSlideToggleModule } from  
'@angular/material/slide-toggle';  
  
@NgModule ({  
  imports: [  
    MatCardModule, MatInputModule,  
    FormsModule,  
  ]  
})  
class AppModule {}
```

```
imports: [
  BrowserModule,
  AppRoutingModule,
  HttpClientModule,
  FormsModule, // Ajoutez FormsModule ici
],
```

Introduction to Template Driven Forms - the ngForm Directive

ngForm Directive

La directive **ngForm** est une partie essentielle de la gestion des formulaires dans Angular. Elle est automatiquement appliquée à toute balise **<form>** dans les templates Angular, sauf si le formulaire est désactivé avec un attribut **ngNoForm**. aspects clés de ngForm que à connaître :

Automatisation et rôle

Form Control :

ngForm crée et attache une instance de **FormGroup** à la balise **<form>**. Cela permet de contrôler les entrées du formulaire et de gérer la validation, l'état, et le groupement des contrôles du formulaire.

Introduction to Template Driven Forms - the ngForm Directive

Validation : Avec ngForm, nous pouvons utiliser des validateurs intégrés comme **required**, **minlength**, **maxlength**, et bien d'autres, ou des validateurs personnalisés pour imposer des contraintes sur les données saisies par l'utilisateur.

État du formulaire : La directive expose l'état du formulaire à travers des propriétés telles que **valid**, **invalid**, **pristine**, **dirty**, **touched**, et **untouched**. Ces propriétés sont utiles pour offrir un retour d'information à l'utilisateur, comme désactiver un bouton de soumission tant que le formulaire n'est pas valide ou afficher des messages d'erreur spécifiques lorsque des champs sont incorrects.

Introduction to Template Driven Forms - the ngForm Directive

Fonctionnalités

Binding de données : **ngForm** permet le **Two-Way Data Binding** grâce à la directive **ngModel** appliquée aux champs de formulaire. Cela synchronise les valeurs des champs de formulaire avec le modèle de données du composant.

Gestion des événements : nous pouvons gérer les événements de soumission de formulaire en utilisant (**ngSubmit**), qui peut être lié à une méthode dans le composant Angular. Cela est particulièrement utile pour effectuer des opérations comme l'envoi de données à un serveur après la validation du formulaire.

Accès au formulaire dans le composant : nous pouvons accéder à l'instance de **ngForm** dans votre composant en utilisant une variable locale dans le template, par exemple `<form #myForm="ngForm">`, et en référençant **myForm** dans notre composant.

Introduction to Template Driven Forms - the ngForm

```
<mat-card class="login-page">
  <mat-card-title>Login</mat-card-title>

  <mat-card-content>

    <form class="login-form data-form" #loginform="ngForm">

      <mat-form-field>
        <input matInput type="email" name="email"
               placeholder="Email">
      </mat-form-field>

      <mat-form-field>
        <input matInput type="password" placeholder="Password">
      </mat-form-field>

      <button mat-raised-button color="primary" (click)="Loginclick(loginform)">
        Login
      </button>

    </form>

  </mat-card-content>
</mat-card>
```

```
export class LoginFormEnrermEnreComponent {
  Loginclick(loginform: NgForm) {
    console.log(loginform.value ,
    loginform.valid);
  }
}
```

Understanding how the Angular ngModel Directive Works

La directive **ngModel** est une partie intégrante des formulaires dans Angular, offrant une liaison bidirectionnelle entre les éléments de formulaire dans le template et les propriétés de données dans le composant. aspects principaux :

Fonctionnalité principale

ngModel lie une propriété d'un modèle de données à un champ de formulaire HTML, permettant :

Liaison bidirectionnelle : Les changements dans le champ de formulaire mettent à jour la propriété du modèle, et les modifications de la propriété du modèle mettent à jour le champ de formulaire.

Validation : Intégration avec les validations HTML natives et les validations personnalisées pour contrôler la saisie de l'utilisateur.

```
<mat-card class="login-page">
  <mat-card-title>Login</mat-card-title>

  <mat-card-content>
    <form class="login-form" data-form="#loginform" #ngForm="ngForm">
      <mat-form-field>
        <input matInput type="email" name="email" ngModel placeholder="Email">
      </mat-form-field>

      <mat-form-field>
        <input matInput type="password" ngModel placeholder="Password" name="password">
      </mat-form-field>

      <button mat-raised-button color="primary" (click)="Loginclick(loginform)">
        Login
      </button>
    </form>
  </mat-card-content>
</mat-card>
```

```
export class
  LoginFormEnrermEnreComponent {
  Loginclick(loginform: NgForm)
  {
    console.log(loginform.value
    , loginform.valid);
  }
}
```

```
</button>

<div class="form-val">
  {{ loginform.value | json}}
</div>

</form>
</mat-card-content>
</mat-card>
```

ngForm

Rôle : ngForm est une directive qui est automatiquement appliquée à toute balise `<form>` dans les templates Angular, sauf si le formulaire est explicitement désactivé. Elle crée un objet **FormGroup** au niveau du formulaire qui permet de contrôler l'état et la validité de tous les champs du formulaire.

Gestion de l'état : ngForm suit l'état de tous les contrôles dans le formulaire, y compris leur état de validité (valid, invalid), leur état de modification (pristine, dirty), et leur interaction avec l'utilisateur (touched, untouched).

ngModel

Rôle : ngModel est utilisé pour créer une liaison bidirectionnelle entre les éléments de formulaire HTML et les propriétés du modèle de données dans le composant. Chaque instance de ngModel maintient également son propre état de validation et d'interaction.

Intégration dans ngForm : Lorsqu'il est utilisé à l'intérieur d'un `<form>` qui utilise ngForm, chaque contrôle ngModel est enregistré auprès de l'objet FormGroup parent. Cela permet à ngForm de surveiller et de réagir aux changements d'état de tous les contrôles individuels.

Lien entre ngForm et ngModel

Validation collective :

ngForm regroupe les états de validation de tous les contrôles ngModel sous sa gestion. Cela signifie que vous pouvez afficher des messages d'erreur ou contrôler l'accessibilité des actions de formulaire (comme la soumission du formulaire) en fonction de l'état global du formulaire.

Propriété form :

Dans le composant Angular, ngForm expose une propriété form qui est un FormGroup. Ce groupe contient les instances de FormControl pour chaque contrôle ngModel utilisé dans le formulaire, permettant un accès programmatique à leur état et leur valeur.

Événements et réactivité : Les modifications apportées aux valeurs des contrôles via ngModel sont immédiatement reflétées dans l'objet FormGroup de ngForm, et vice versa. Ceci est crucial pour des fonctionnalités telles que la désactivation conditionnelle des boutons de soumission ou l'affichage des erreurs de validation qui dépendent de plusieurs champs.

```
<mat-form-field>

  <input matInput type="email" name="email" ngModel
    #email="ngModel" placeholder="Email">

</mat-form-field>

<mat-form-field>

  <input matInput type="password" ngModel #pasw="ngModel" placeholder="Password" name="password">

</mat-form-field>

<button mat-raised-button color="primary" (click)="Loginclick(loginform)">
  Login
</button>

<div class="form-val">
  {{ email.value}} {{ pasw.value}}
</div>
```

Introduction to Form Validation

la validation de formulaires avec Angular aborde comment assurer que les données saisies par les utilisateurs respectent les critères spécifiques avant que le formulaire ne soit soumis. Angular offre un cadre solide pour la validation de formulaires, tant pour les formulaires pilotés par le modèle (reactive forms) que ceux pilotés par le template.

Validation de formulaire dans Angular

Validation HTML5 : Utilise des attributs de validation HTML5 pour assurer des règles de validation de base.

Validateurs intégrés : Angular fournit des validateurs intégrés tels que Validators.required, Validators.minLength, Validators.maxLength, et Validators.pattern qui peuvent être appliqués aux contrôles de formulaire dans les formulaires réactifs.

Validateurs personnalisés : Pour des règles de validation plus complexes qui ne peuvent être couvertes par les validateurs intégrés ou HTML5, Angular permet de créer des validateurs personnalisés. Ces derniers peuvent être des fonctions qui prennent un FormControl et retournent un objet de clés d'erreur si la validation échoue.

```
<form class="login-form data-form" #loginform="ngForm">

  <mat-form-field>

    <input matInput type="email" name="email" ngModel required
      #email="ngModel" placeholder="Email">

  </mat-form-field>

  <mat-form-field>

    <input matInput type="password" required ngModel #pasw="ngModel" placeholder="Password"
name="password">

  </mat-form-field>

  <button mat-raised-button color="primary" (click)="Loginclick(loginform)">
    Login
  </button>

  <div class="form-val">
    {{ email.valid}}
  </div>
```

Understanding Angular Forms CSS State Classes - ng-valid ng-dirty ng-touched

Angular ajoute automatiquement certaines classes CSS aux éléments de formulaire basés sur leur état. Ces classes sont très utiles pour fournir un retour visuel à l'utilisateur, ainsi que pour styliser les éléments de formulaire en fonction de l'interaction de l'utilisateur et de la validité des données entrées. Exemple des trois principales classes CSS d'état utilisées par Angular: **ng-valid**, **ng-dirty**, et **ng-touched**.

1. ng-valid et ng-invalid

ng-valid: Cette classe est ajoutée à un élément de formulaire lorsque la valeur de l'élément satisfait toutes les contraintes de validation. Cela signifie que selon les règles de validation définies pour ce champ de formulaire, les données entrées sont correctes.

ng-invalid: Cette classe est ajoutée à l'élément de formulaire lorsque la valeur de l'élément ne respecte pas une ou plusieurs contraintes de validation. C'est l'opposé de ng-valid et est utilisée pour indiquer que quelque chose ne va pas avec les données saisies par l'utilisateur.

Understanding Angular Forms CSS State Classes - ng-valid ng-dirty ng-touched

2. ng-dirty et ng-pristine

ng-dirty: Angular ajoute cette classe à un élément de formulaire dès que sa valeur est modifiée par l'utilisateur. Cela signifie que l'utilisateur a interagi avec cet élément de formulaire, modifiant sa valeur initiale.

ng-pristine: Cette classe est présente tant que l'utilisateur n'a pas encore modifié la valeur initiale de l'élément de formulaire. Elle est utile pour déterminer si un formulaire a été altéré depuis son chargement initial.

3. ng-touched et ng-untouched

ng-touched: Cette classe est ajoutée à un élément de formulaire lorsque l'utilisateur a focalisé puis quitté cet élément. Elle est utile pour la validation de formulaires où vous ne voulez montrer des erreurs que après que l'utilisateur ait "touché" le champ.

ng-untouched: La classe opposée à ng-touched, qui indique que l'élément n'a jamais été focalisé. Cela peut être utilisé pour ne pas afficher d'erreurs de validation tant que l'utilisateur n'a pas interagi avec le champ.

```
input.ng-invalid {  
    border: 3px solid red;  
}  
  
input.ng-invalid.ng-touched {  
    border: 3px solid red;  
}  
  
input.ng-valid.ng-touched {  
    border: 1px solid green;  
}
```

How to Display Error Messages in an Angular Form

```
<mat-form-field>

  <input matInput type="email" name="email" ngModel required
    #email="ngModel" placeholder="Email">
    <mat-error> the email is mandatory</mat-error>

</mat-form-field>
```

```
<mat-form-field>

  <input matInput type="email" name="email" ngModel required
    #email="ngModel" placeholder="Email">

    <mat-error *ngIf="email.errors"> the email is mandatory</mat-error>

</mat-form-field>
```

Learn All the Built-in Template-Driven Form Validators

minlength et maxlength

Ces validateurs contrôlent la longueur minimum et maximum du texte entré dans un champ de formulaire. Ils sont utiles pour les validations de mot de passe, de numéro de téléphone, ou d'autres champs nécessitant une longueur spécifique.

```
<mat-form-field>

  <input matInput type="email" name="email" ngModel #email="ngModel"
    required
    minlength="3"
    placeholder="Email">

    <mat-error *ngIf="email.errors?.['required']">The email is mandatory.</mat-error>
    <mat-error *ngIf="email.errors?.['minlength']">Email must be at least 3 characters
long.
  </mat-error>

</mat-form-field>
```

Learn All the Built-in Template-Driven Form Validators

email

Le validateur email vérifie si la valeur saisie est une adresse email valide. Ce validateur utilise une expression régulière relativement générale pour vérifier le format de l'email.

```
mat-form-field>
```

```
<input matInput type="email" name="email"  ngModel  #email="ngModel"
required
minlength="3"
maxlength="50"
email
placeholder="Email">

<mat-error *ngIf="email.errors?.['email']">this is not a valid  email.</mat-error>
```

Learn All the Built-in Template-Driven Form Validators

Pattern

Le validateur pattern permet de définir une expression régulière contre laquelle la valeur du champ doit correspondre. Ceci est utile pour des formats spécifiques comme les codes postaux, les numéros de téléphone, les emails, etc.

```
<input matInput type="email" name="email" ngModel #email="ngModel"
       required
       minlength="3"
       maxlength="50"
       email
       pattern="[a-z]+"
       placeholder="Email">
```

How to use ngSubmit - Disabling the Form Submit button

Dans Angular, la directive **ngSubmit** est utilisée conjointement avec les formulaires pour gérer les événements de soumission de formulaire. Elle permet d'exécuter une fonction lorsqu'un formulaire est soumis, souvent utilisée pour valider et traiter les données du formulaire. De plus, Angular vous permet de désactiver le bouton de soumission jusqu'à ce que le formulaire soit valide, améliorant ainsi les interactions avec le formulaire en empêchant les utilisateurs de soumettre des données incomplètes ou invalides.

```
<form class="login-form data-form" #loginform="ngForm" (ngSubmit)="Loginclick(loginform,$event)">

<mat-form-field>

  <input matInput type="email" name="email" ngModel #email="ngModel"
    required
    minlength="3"
    maxlength="50"
    email
    placeholder="Email">

    <mat-error *ngIf="email.errors?.['email']">this is not a valid email.</mat-error>
    <mat-error *ngIf="email.errors?.['required']">The email is mandatory.</mat-error>
    <mat-error *ngIf="email.errors?.['minlength']">Email must be at least 3 characters long.
  </mat-error>

</mat-form-field>

<mat-form-field>

  <input matInput type="password" required ngModel #pasw="ngModel" placeholder="Password" name="password">

</mat-form-field>

<button mat-raised-button color="primary" type="submit" [disabled]="!loginform.valid">
  Login
</button>
```

Advanced ngModel - ngModelChange and the ngModelOptions updateOn Property

Dans Angular, **ngModel** est souvent utilisé pour créer des liaisons bidirectionnelles de données entre le modèle et la vue dans les formulaires. Pour des cas d'utilisation plus avancés, nous pouvons tirer parti de **ngModelChange** et de la propriété **ngModelOptions**, notamment avec l'option **updateOn**. Ces outils permettent un contrôle plus fin sur le comportement de liaison et de validation du formulaire.

ngModelChange:

ngModelChange est un événement qui est émis chaque fois que la valeur d'un champ de formulaire est modifiée. Cela peut être utile pour effectuer une logique personnalisée dès que la valeur change, avant même que la valeur ne soit mise à jour dans le modèle.

```
<mat-form-field>

  <input matInput type="email" name="email" ngModel #email="ngModel"
    (ngModelChange)="onEmailChange($event)"
    required
    minlength="3"
    maxlength="50"
    email
    placeholder="Email">

    <mat-error *ngIf="email.errors?.['email']">this is not a valid email.</mat-error>
    <mat-error *ngIf="email.errors?.['required']">The email is mandatory.</mat-error>
    <mat-error *ngIf="email.errors?.['minlength']">Email must be at least 3 characters long.
  </mat-error>

</mat-form-field>
```

```
onEmailChange(newValue: string) {
  console.log('Nouvelle valeur:', newValue);
  // Vous pouvez ici appliquer une logique supplémentaire, comme transformer la
  // valeur.
}
```

Advanced ngModel - ngModelChange and the ngModelOptions updateOn Property

ngModelOptions avec l'option updateOn

La propriété **ngModelOptions** permet de configurer le comportement de mise à jour du modèle. L'option **updateOn** spécifie le moment où le modèle doit être mis à jour. Les valeurs possibles sont :

change (par défaut) : met à jour le modèle à chaque changement dans le champ de formulaire.

blur : met à jour le modèle lorsque le champ perd le focus.

submit : met à jour le modèle uniquement lors de la soumission du formulaire.

```
<input matInput type="email" name="email" ngModel #email="ngModel"
[ngModelOptions]="{updateOn: 'blur'}"
(ngModelChange)="onEmailChange($event)"
required
minlength="3"
maxlength="50"
email
placeholder="Email">

<mat-error *ngIf="email.errors?.['email']">this is not a valid email.</mat-error>
<mat-error *ngIf="email.errors?.['required']">The email is mandatory.</mat-error>
<mat-error *ngIf="email.errors?.['minlength']">Email must be at least 3 characters
long.
</mat-error>

</mat-form-field>
```

Advanced ngModel - ngModelOptions In Detail

Standalone

Cette option permet de spécifier si le contrôle doit être enregistré dans un groupe de contrôles Angular ou non. Par défaut, tout contrôle utilisant **ngModel** et situé à l'intérieur d'une balise `<form>` sera automatiquement enregistré.

true : Le contrôle n'est pas enregistré dans un groupe de contrôles parent.

false : Le contrôle est enregistré dans un groupe de contrôles parent (comportement par défaut).

```
<mat-form-field>

  <input matInput type="email" name="email" ngModel #email="ngModel"
    [ngModelOptions]="{updateOn: 'blur'}"
    (ngModelChange)="onEmailChange($event)"
    required
    minlength="3"
    maxlength="50"
    email
    [ngModelOptions]="{standalone: true}"
    placeholder="Email">

  <mat-error *ngIf="email.errors?.['email']">this is not a valid email.</mat-error>
  <mat-error *ngIf="email.errors?.['required']">The email is mandatory.</mat-error>
  <mat-error *ngIf="email.errors?.['minlength']">Email must be at least 3 characters long.
  </mat-error>

</mat-form-field>
```

Advanced ngModel - ngModelOptions In Detail

name

Cette option permet de définir un nom pour le contrôle qui diffère de celui utilisé dans l'attribut name. Ceci est particulièrement utile lorsqu'il y a des conflits de nommage ou lorsque vous souhaitez utiliser un nom différent dans le modèle Angular de celui du formulaire HTML.

```
<mat-form-field>

  <input matInput type="email"    ngModel #email="ngModel"
    [ngModelOptions]="{name: 'userName'}"
    (ngModelChange)="onEmailChange($event)"
    required
    minlength="3"
    maxlength="50"
    email
    [ngModelOptions]="{standalone: true}"
    placeholder="Email">

    <mat-error *ngIf="email.errors?.['email']">this is not a valid email.</mat-error>
    <mat-error *ngIf="email.errors?.['required']">The email is mandatory.</mat-error>
    <mat-error *ngIf="email.errors?.['minlength']">Email must be at least 3 characters long.
    </mat-error>

</mat-form-field>
```

Advanced ngModel - ngModelOptions In Detail

Liaison Unidirectionnelle avec ngModel

La liaison unidirectionnelle est utilisée principalement pour afficher une valeur de données dans l'interface utilisateur sans la modifier directement à partir de l'interface utilisateur. Cela est utile pour les champs qui doivent être affichés mais pas modifiés par l'utilisateur.

```
val = {  
  email: 'millimono64.sm@gmail.com',  
  password: 12345,  
};  
Loginclick(loginform: NgForm, submit: any) {  
  console.log(loginform.value,  
loginform.valid, submit);  
  console.log('val :', this.val);  
}
```

```
<input matInput type="email" name="email"  
  [ngModel]="val.email"  
  #email="ngModel"  
  required  
  minlength="3"  
  maxlength="50"  
  email  
  [ngModelOptions]={`${standalone:  
placeholder="Email"}`>
```

Advanced ngModel - ngModelOptions In Detail

Liaison Bidirectionnelle avec ngModel

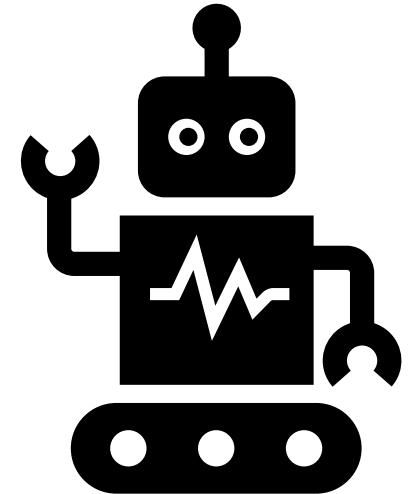
La liaison bidirectionnelle, comme son nom l'indique, permet une communication à double sens : le modèle et la vue peuvent se mettre à jour mutuellement. C'est utile pour les formulaires et autres entrées utilisateur, où nous voulons non seulement afficher une valeur mais aussi la mettre à jour en réponse aux actions de l'utilisateur.

```
val = {  
  email: 'millimono64.sm@gmail.com',  
  password: 12345,  
};  
Loginclick(loginform: NgForm, submit: any) {  
  console.log(loginform.value,  
loginform.valid, submit);  
  console.log('val :', this.val);  
}
```

```
<input matInput type="email" name="email"  
[(ngModel)]="val.email"  
#email="ngModel"  
required  
minlength="3"  
maxlength="50"  
email  
[ngModelOptions]={`${standalone:  
placeholder="Email"}`
```

Multiple error messages per form field - Understanding the problem

2. Formulaires Réactifs (Reactive Forms).



Les formulaires réactifs offrent une approche plus robuste, scalable, et réutilisable pour gérer les formulaires dans les applications Angular. Ils sont basés sur des modèles observables et permettent de manipuler plus facilement les données de formulaire avec une logique complexe, des validations dynamiques, et des changements de configuration à la volée.

Avantages :

- Plus de flexibilité et de contrôle sur la structure du formulaire.
- Facilite les tests unitaires car ils sont synchrones et ne dépendent pas du DOM.
- Utilisation de l'approche basée sur les flux de données réactifs.

Comment ça marche ? :

On crée des instances de **FormGroup** et **FormControl** dans notre composant pour représenter le formulaire et ses champs. Les validations sont également définies dans le composant plutôt que dans le template.

Introduction to Reactive Forms - Step-by-Step Example

```
form = new FormGroup({  
  email: new FormControl('', {  
    validators: [Validators.required, Validators.email],  
  }),  
  password: new FormControl('', {  
    validators: [Validators.required, Validators.minLength(8)],  
  }),  
});
```

```
imports: [  
  BrowserModule,  
  AppRoutingModule,  
  HttpClientModule,  
  MatCardModule,  
  FormsModule,  
  MatInputModule,  
  ReactiveFormsModule,  
],
```

Les formulaires réactifs utilisent **FormGroup** pour représenter le formulaire lui-même, **FormControl** pour ses champs, et **FormArray** pour les listes de champs dynamiques. Chaque **FormControl** dans un **FormGroup** correspond à un élément du formulaire dans le modèle.

```
<mat-card class="login-page">
  <mat-card-title>Login (Reactive)</mat-card-title>
  <mat-card-content>

    <form class="login-form data-form" [FormGroup]="form">

      <mat-form-field>

        <input matInput type="email" name="email"
               placeholder="Email" formControlName="email">

      </mat-form-field>

      <mat-form-field>

        <input matInput type="password" placeholder="Password" formControlName="password">

      </mat-form-field>

      <button mat-raised-button color="primary">
        Login
      </button>

    </form>

  </mat-card-content>
</mat-card>
```

Reactive Forms - The Form Control Directive and Custom Validators

```
<mat-form-field>
  <input matInput type="email" name="email"
         placeholder="Email" [formControl]="email">
</mat-form-field>

<mat-form-field>

  <input matInput type="password" placeholder="Password" [formControl]="password">
</mat-form-field>

email = new FormControl('', {
  validators: [Validators.required, Validators.email],
});

password = new FormControl('', {
  validators: [Validators.required, Validators.minLength(8)],
});

form = new FormGroup({
  email: this.email,
  password: this.password,
});
```

Reactive Forms - The Form Control Directive and Custom Validators

```
email = new FormControl('', {  
  validators: [Validators.required, Validators.email],  
  updateOn: 'blur',  
});
```

```
email = new FormControl('', {  
  validators: [Validators.required, Validators.email],  
});  
  
password = new FormControl('', {  
  validators: [Validators.required, Validators.minLength(8)],  
});  
  
form = new FormGroup({  
  email: this.email,  
  
  password: this.password,  
});
```

The Form Builder API - Writing much more concise Reactive Forms

L'API FormBuilder dans Angular simplifie le processus de création de formulaires réactifs en fournissant une syntaxe plus concise et lisible par rapport à l'utilisation directe des classes FormGroup, FormControl et FormArray. Cette API est un service dans Angular qui injecte FormBuilder pour aider à réduire le code répétitif nécessaire pour construire des formulaires complexes.

Le service FormBuilder fournit des méthodes telles que group(), control() et array() qui correspondent respectivement aux classes FormGroup, FormControl et FormArray. Ces méthodes facilitent la construction d'un modèle de formulaire.

The Form Builder API - Writing much more concise Reactive Forms

```
export class LoginReactiveComponent {
  email = new FormControl('', {
    validators: [Validators.required, Validators.email],
    updateOn: 'blur',
  });

  password = new FormControl('', {
    validators: [Validators.required, Validators.email],
  });

  form = this.fb.group({
    email: ['', [Validators.required, Validators.email]],
    password: ['', [Validators.required, Validators.minLength(8)]]
  });

  constructor(private fb : FormBuilder){
  }
}
```

The Form Builder API - Writing much more concise Reactive Forms

```
form = this.fb.group({  
  email: [  
    '',  
    {  
      validators: [Validators.required, Validators.email],  
      updateOn: 'blur',  
    },  
  ],  
  password: ['', [Validators.required, Validators.minLength(8)]],  
});
```

```
<input matInput type="email" name="email"  
      placeholder="Email" formControlName="email">  
  
</mat-form-field>  
  
<mat-form-field>  
  
<input matInput type="password" placeholder="Password" formControlName="password">
```

Comparing Reactive and Template Driven Forms

```
<mat-form-field>

  <input matInput type="email" name="email"
    placeholder="Email" formControlName="email">

</mat-form-field>

<mat-error *ngIf="email.errors?.['email']">this is not a valid email.</mat-error>
<mat-error *ngIf="email.errors?.['required']">The email is mandatory.</mat-error>
<mat-error *ngIf="email.errors?.['minlength']">Email must be at least 3 characters
long.
</mat-error>

<mat-form-field>

get email() {
  return this.form.controls['email'];
}
```

Angular Strictly Typed Forms

Non Nullable Form Builder

Appels HTTP et Observables (8 heures)

Professeur Titulaire : Millimono Sory (PhD, AI)

Plan Du Cours



○ **Formulaires réactifs : création et validation.**

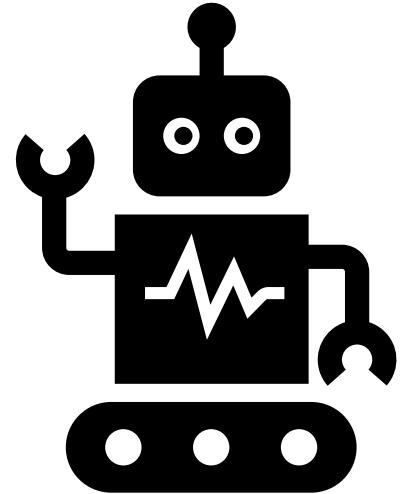
○ **Formulaires pilotés par le template.**

○ **Projet pratique sur les formulaires.**

○ **Projet pratique sur les formulaires.**

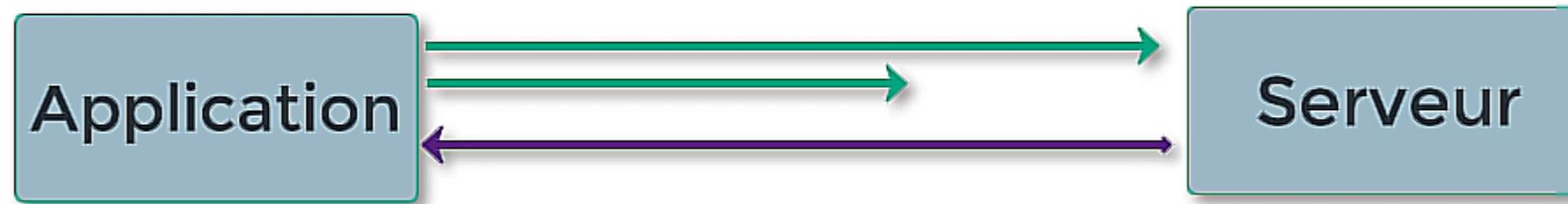


La Programmation Réactive



Le fonctionnement des promesses

La gestion est asynchrone, plusieurs requêtes ont lieu simultanément, et sont non-bloquantes.



Les promesses en JavaScript représentent la valeur finale d'une opération asynchrone. Elles sont très utilisées pour gérer des séquences d'opérations asynchrones et simplifier la gestion des erreurs comparativement aux anciennes techniques telles que les callbacks.

Création d'une promesse

Une promesse est créée en utilisant le constructeur Promise. Ce constructeur prend une fonction exécuteur qui elle-même accepte deux fonctions en arguments: resolve et reject :

```
let promesse = new Promise((resolve, reject) => {
  // Logique asynchrone ici
  const condition = true; // Condition hypothétique pour l'exemple
  if (condition) {
    resolve("Opération réussie");
  } else {
    reject("Opération échouée");
  }
});
```

2. Consommation d'une promesse

Pour utiliser une promesse, vous vous y abonnez en utilisant les méthodes `.then()`, `.catch()`, et `.finally()` :

`.then()` : permet de définir ce qui doit se passer après que la promesse ait été résolue. Elle prend deux fonctions en argument, une pour le cas de réussite et une autre pour l'échec :

```
promesse.then(  
    result => console.log(result), // Exécuté si `resolve` est appelé  
    error => console.error(error) // Exécuté si `reject` est appelé  
);
```

`.catch()` : permet de capturer et de gérer les erreurs si la promesse est rejetée :

```
promesse.catch(  
    error => console.error(error)  
);
```

.finally() : permet d'exécuter un code quelle que soit l'issue de la promesse, que celle-ci soit résolue ou rejetée. Elle est souvent utilisée pour effectuer des nettoyages :

```
promesse.finally(  
  () => console.log("Promesse terminée, avec ou sans succès.")  
);
```

Chaînage des promesses

Le chaînage est une technique puissante qui permet de connecter plusieurs opérations asynchrones successives :

```
fetch('https://api.example.com/data') // API fetch retourne une promesse
  .then(response => response.json()) // transforme la réponse en JSON
  .then(data => {
    console.log(data);
    return data;
  })
  .catch(error => console.error("Erreur lors de la récupération des données:", error));
```

Promesses simultanées

Pour exécuter plusieurs promesses en parallèle et attendre que toutes soient résolues, on utilise `Promise.all` :

```
Promise.all([promesse1, promesse2])
  .then(results => {
    console.log(results[0]); // Résultat de promesse1
    console.log(results[1]); // Résultat de promesse2
  })
  .catch(error => console.error("Une des promesses a échoué:", error));
```

Dans le contexte d'Angular, une promesse représente une opération qui n'a pas encore été complétée mais qui est attendue dans le futur. Cela permet à Angular de gérer des opérations asynchrones, telles que les appels HTTP, de manière plus prévisible sans bloquer l'exécution du reste du code.

Comment les Promesses sont Utilisées dans Angular

Intégration avec le Service HTTP : Angular utilise des promesses via son service HTTP pour effectuer des requêtes asynchrones. Par défaut, Angular renvoie des Observable à partir de ses méthodes HTTP, mais ces observables peuvent être convertis en promesses en utilisant **firstValueFrom()** ou **lastValueFrom()** de RxJS. Cela peut être préféré dans les cas où une seule réponse est attendue et où l'on ne souhaite pas gérer un flux d'événements continu.

Gestion des Promesses dans les Composants :

Dans les composants Angular, les promesses peuvent être utilisées pour charger ou traiter des données lors de l'initialisation. L'utilisation de `async` et `await` permet de gérer les promesses de manière plus lisible et plus concise, en évitant la nécessité d'utiliser des callbacks ou des structures complexes pour gérer les états de succès ou d'erreur.

Utilisation dans les Guards et les Résolveurs :

Angular utilise également des promesses dans ses route guards et résolveurs, où elles permettent de retarder le chargement de routes jusqu'à ce que certaines données soient chargées ou certaines conditions soient remplies, offrant ainsi une expérience utilisateur plus fluide et prédictive.

Avantages des Promesses dans Angular

Simplicité : Les promesses sont souvent plus simples à comprendre et à utiliser que les observables pour les développeurs qui ne sont pas familiers avec la programmation réactive.

Gestion des erreurs : La gestion des erreurs avec des promesses est souvent plus directe grâce à la syntaxe try/catch en utilisant async/await.

Intégration avec Async/Await : Angular prend pleinement en charge l'usage d'async et await avec des promesses, ce qui simplifie le code en évitant les imbriques de callbacks et en rendant le code asynchrone aussi facile à lire que du code synchrone.

La programmation réactive

La programmation réactive est un paradigme de programmation orienté autour des flux de données et la propagation du changement. Cela signifie qu'il est principalement utilisé pour développer des applications qui réagissent aux changements de manière continue, efficace et scalable. quelques aspects clés de la programmation réactive :

Flux de données

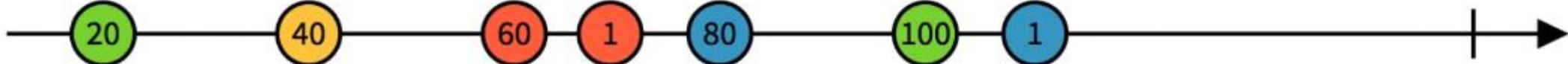
Dans la programmation réactive, les données sont traitées comme des flux continus qui peuvent être observés. Tout système qui réagit aux changements dans ces flux de données peut être considéré comme réactif. Cela inclut les changements dans l'interface utilisateur, les requêtes réseau, les entrées utilisateur, etc.

La programmation réactive

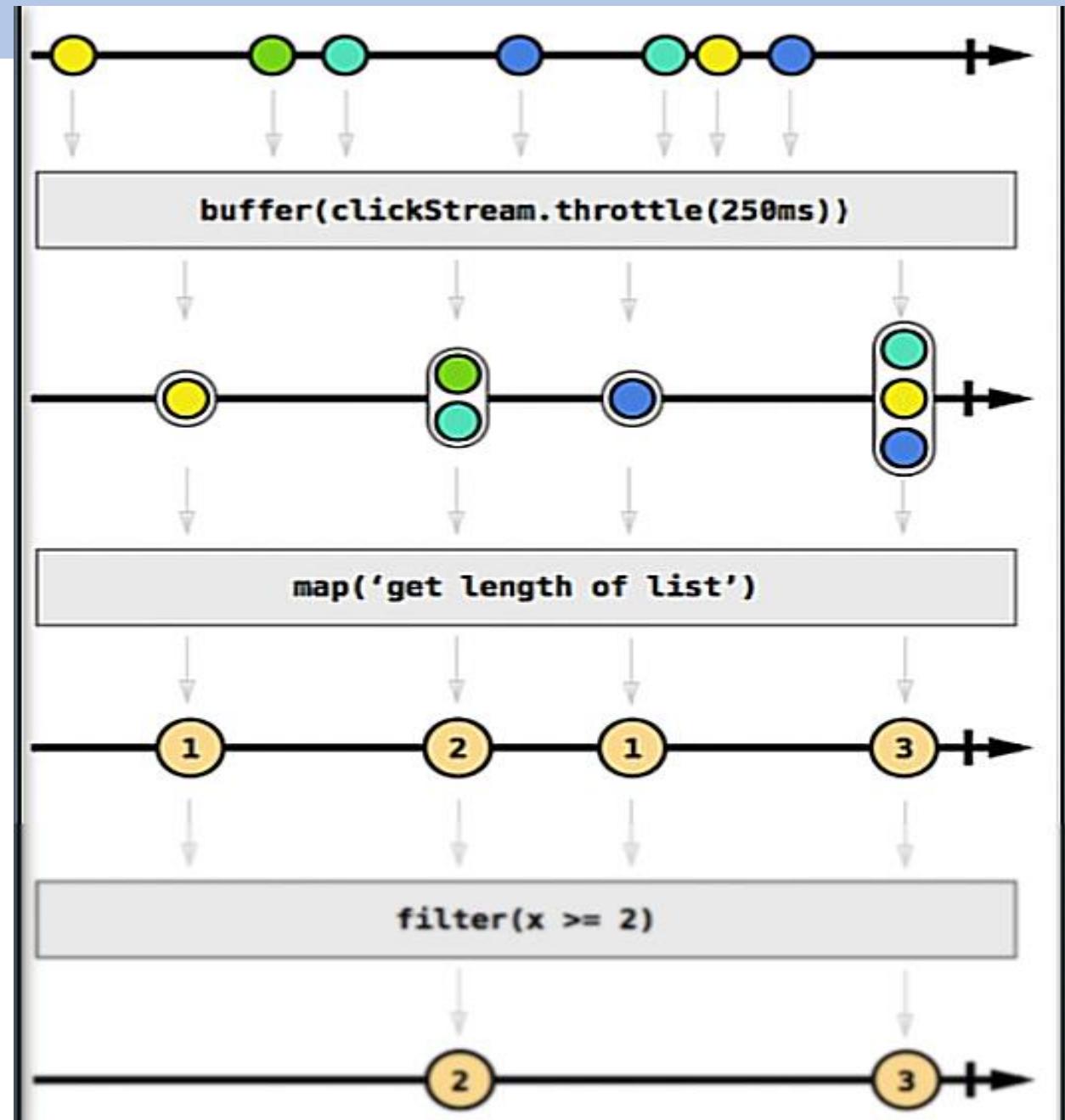
Interactive diagrams of Rx Observables



Programmation réactive =
Programmation avec des
flux de données asynchrones



Flux de données



La programmation réactive

Propagation du changement

Quand des données changent quelque part dans l'application, les parties de l'application qui dépendent de ces données réagissent automatiquement à ces changements. Cela est souvent réalisé à travers des mécanismes tels que les observables et les sujets (subjects) dans RxJS, une bibliothèque largement utilisée pour la programmation réactive en JavaScript et Angular.

Non-blocage

La programmation réactive encourage l'utilisation de modèles non-bloquants. Cela signifie que les opérations, particulièrement celles qui sont intensives comme les requêtes réseau ou les accès disque, sont traitées de manière à ne pas bloquer l'exécution du programme pendant qu'elles sont en attente d'une réponse ou d'une ressource.

La programmation réactive

Asynchronisme

Les applications réactives gèrent naturellement les opérations asynchrones. En utilisant des observables, par exemple, les développeurs peuvent facilement composer et chaîner des opérations asynchrones sans se perdre dans des callbacks ou des gestions d'erreurs complexes.

Backpressure

La gestion de la "contre-pression" ou "backpressure" est un concept avancé en programmation réactive, où le consommateur de données peut signaler au producteur la quantité de données qu'il est capable de gérer, évitant ainsi la surcharge et les problèmes de performance.

La programmation réactive

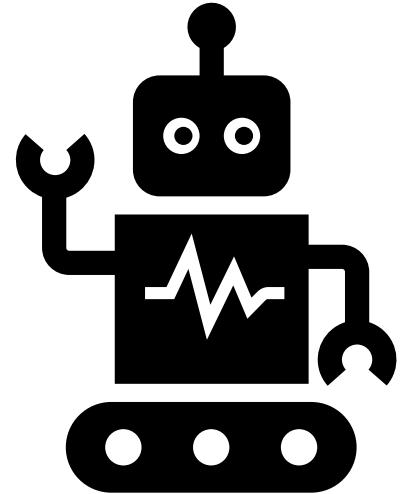
Exemples d'utilisation

Interfaces Utilisateurs Réactives: Les frameworks modernes comme Angular, React, et Vue.js utilisent des principes de programmation réactive pour mettre à jour l'interface utilisateur en réaction aux changements d'état de l'application.

Développement de serveurs: Des frameworks comme Spring WebFlux et Node.js avec certains packages supportent la programmation réactive pour gérer des requêtes HTTP de manière non-bloquante, permettant de servir plus d'utilisateurs avec moins de ressources.

La programmation réactive est particulièrement utile dans les environnements où la performance, l'efficacité et la scalabilité sont critiques, notamment dans les applications web modernes et les systèmes de traitement de données en temps réel.

La librairie RxJS Les Observables



Les Observables sont une fonctionnalité clé de la bibliothèque RxJS, largement utilisée dans le développement Angular pour gérer des opérations asynchrones et des flux de données. Un Observable est fondamentalement un producteur de multiples valeurs sur une période de temps, contrairement à une Promesse qui ne produit qu'une seule valeur. Les Observables peuvent émettre zéro, une ou plusieurs valeurs, et ils fonctionnent sur le principe de la publication et de l'abonnement.

Concepts clés des Observables :

Création d'Observables :

Un Observable peut être créé en utilisant la fonction **new Observable** de RxJS, qui prend une fonction comme argument. Cette fonction définit comment l'Observable doit produire des données.

Abonnement :

Pour qu'un Observable commence à émettre des valeurs, il doit être souscrit. L'abonnement se fait en appelant la méthode **.subscribe()** sur l'instance Observable. Cette méthode prend jusqu'à trois arguments : les fonctions de callback pour les valeurs émises (**next**), les erreurs (**error**) et la complétion (**complete**) du flux.

Opérateurs RxJS :

RxJS propose une vaste collection d'opérateurs qui permettent de transformer, filtrer, combiner et gérer les flux de données de diverses manières. Par exemple, **map**, **filter**, **merge**, **concat**, **switchMap**, etc.

Désabonnement :

Pour éviter les fuites de mémoire et d'autres problèmes de performance, il est important de se désabonner des Observables une fois qu'ils ne sont plus nécessaires. Cela se fait en appelant la méthode **.unsubscribe()** sur la souscription.

1. map

L'opérateur map transforme les valeurs émises par un Observable en appliquant une fonction à chaque valeur émise. Cet opérateur est similaire à la méthode map des tableaux en JavaScript, mais il s'applique à des éléments d'un flux Observable.

2. filter

L'opérateur filter permet de sélectionner des valeurs émises par un Observable selon un critère spécifique, et n'émet que les valeurs qui satisfont ce critère.

3. merge

merge est un opérateur qui combine plusieurs Observables en un seul Observable. Les valeurs sont émises dans l'Observable résultant dès qu'elles sont émises par n'importe lequel des Observables sources.

```
import { merge, of } from 'rxjs';
const first = [1, 2, 3];
const second = [4, 5, 6];
const merged = merge(first, second);
merged.subscribe(console.log); // Output: 1, 2, 3, 4, 5, 6
```

4. concat

concat combine plusieurs Observables en un seul Observable, mais contrairement à merge, il attend la complétion de chaque Observable avant de s'abonner au suivant. Cela garantit que les valeurs sont émises dans l'ordre séquentiel des Observables sources.

5. switchMap

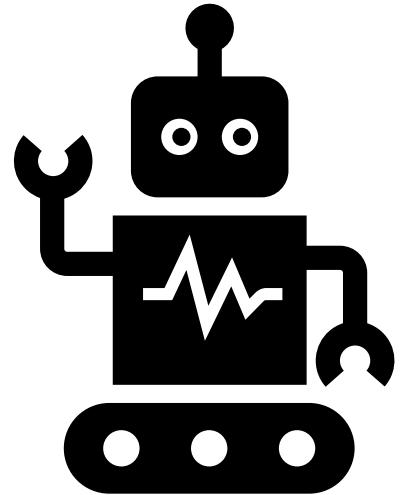
switchMap est utilisé pour traiter les valeurs émises par un Observable source et les mapper à un nouvel Observable. L'opérateur s'abonne à cet Observable et commence à émettre les valeurs qu'il produit. À chaque nouvelle valeur émise par l'Observable source, switchMap annule la souscription à l'Observable précédent et s'abonne au nouvel Observable créé.

```
const source = [1, 2, 3];
const switched = source.pipe( switchMap(val => {
    return timer(1000).pipe(map(() => val * 10)); } ) );
```

switched.subscribe(console.log); // Output: 10, 20, 30 après des intervalles de 1 seconde

```
1 Observable.fromArray([1, 2, 3, 4, 5])
2   .filter(x => x > 2) // 3, 4, 5 ←
3   .map(x => x * 2) // 6, 8, 10
4   .subscribe(x => console.log(x)); // affiche le résultat
5
```

Les Requêtes HTTP



Réaliser des appels HTTP et les intégrer avec les services dans une application Angular est un aspect fondamental pour communiquer avec des serveurs distants et traiter des données dynamiques. Angular fournit le module `HttpClientModule` qui encapsule les fonctionnalités HTTP dans des services réutilisables et facilement testables.

Étape 1 : Importer `HttpClientModule`

Tout d'abord, importer `HttpClientModule` dans notre application. Ceci est généralement fait dans le fichier `app.module.ts`.

```
imports: [ BrowserModule,  
          HttpClientModule  
          // Ajouter HttpClientModule ici ],
```

Étape 2 : Créer un Service pour les Appels HTTP

Ensuite, créer un service qui utilisera `HttpClient` pour faire des appels HTTP. Angular utilise l'injection de dépendances pour fournir des instances de `HttpClient` à nos classes de services.

HttpClient supporte les méthodes HTTP standard telles que GET, POST, PUT, DELETE, etc.

Chaque méthode est disponible sous forme de méthode sur l'objet HttpClient.

GET pour récupérer des données.

POST pour envoyer des données au serveur.

PUT pour mettre à jour des données.

DELETE pour supprimer des données.

4. Gestion des Réponses

Les méthodes de HttpClient retournent des Observable, qui permettent une gestion flexible et puissante des réponses asynchrones. nous pouvons utiliser des opérateurs RxJS pour manipuler ces réponses (comme map, filter, tap, etc.).

5. Headers et Paramètres

HttpClient permet également de configurer facilement des en-têtes HTTP et des paramètres de requête, ce qui est crucial pour les requêtes qui nécessitent des tokens d'authentification ou d'autres configurations spécifiques.

```
public getProtectedData(): Observable<any> {
  const headers = new HttpHeaders({
    'Authorization': 'Bearer your-token-here'
  });

  return this.http.get('/api/protected-data', { headers });
}
```

6. Gestion des Erreurs

La gestion des erreurs se fait par le biais de l'opérateur catchError de RxJS, permettant de traiter les erreurs survenues lors des requêtes HTTP de manière élégante.

7. Avantages de HttpClient

Interception de Requêtes: Capacité d'intercepter les requêtes pour ajouter des en-têtes, logger les requêtes, ou gérer les erreurs globalement.

Typage Fort: Support pour les types génériques qui permet d'associer les types TypeScript aux requêtes et réponses, améliorant ainsi la sécurité de type.

Testabilité: Facilité de tester les interactions HTTP grâce à HttpClientTestingModule.

Projets (8 heures)

Professeur Titulaire : Millimono Sory (AI Seacher)

```
Mkdir demo-angular
```

```
Mkdir frontend-angular
```

```
Npm –version
```

```
Ng version
```

```
ng new frontend-angular --directory=./ --no-standalone
```

```
ng add @angular/material
```

ng g c ecran-template/ecran-template

ng g c home/ home

ng g c profil/ profil

ng g c login/login

ng g c dashboard/dashboard

ng g c student/ student

ng g c paiment/ paiment

ng g c load-student/ load-student

ng g c load-paiment/ load-paiment

ng g s service/auth

ng g g guards/auth

ng g g guards/autorisation

<https://www.youtube.com/@AyyazTech/playlists>

1. Initialisation du projet

```
ng new project-management-app --no-standalone  
cd project-management-app
```

2. Crédit des modules principaux

CoreModule

```
ng g m core
```

Créez les services AuthService et HttpService

```
ng g s core/services/auth  
ng g s core/services/http
```

Créez le guard AuthGuard

```
ng g g core/guards/auth
```

3. Création des Feature Modules

ProjectModule

```
ng g m project  
ng g c project/components/project-list  
ng g c project/components/project-detail  
ng g s project/services/project
```

TaskModule

```
ng g m task  
ng g c task/components/task-list  
ng g c task/components/task-detail  
ng g s task/services/task
```

NotificationModule

```
ng g m notification  
ng g c notification/components/notification-list  
ng g s notification/services/notification
```

DashboardModule

```
ng g m dashboard  
ng g c dashboard/components/dashboard  
ng g s dashboard/services/dashboard
```

CalendarModule

ng g m calendar

ng g c calendar/components/calendar

ng g s calendar/services/calendar

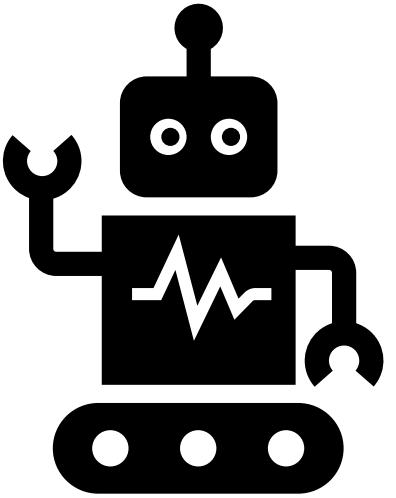
ReportModule

ng g m report

ng g c report/components/report

ng g s report/services/report

Dashboard



Ouvrir le fichier dashboard.module.ts et le mettre à jour pour inclure le composant et le service .

Ajouter les routes pour le module Dashboard

Ensute, configurer le routage interne du module Dashboard en ajoutant un fichier de routage dédié

```
ng g m dashboard/dashboard-routing --flat --module=dashboard
```

Mettez à jour le fichier dashboard-routing.module.ts

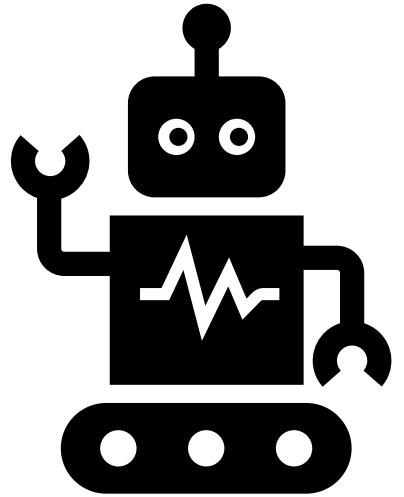
```
const routes: Routes = [ { path: '', component: DashboardComponent } ];
```

Mettez à jour dashboard.module.ts pour inclure DashboardRoutingModule

Implémentez le composant DashboardComponent

```
ng add @angular/material
```


Project



Configurer le module Project

Configurer les routes pour le module Project

```
ng g m project / project -routing --flat --module=project
```

Configurer le module Project

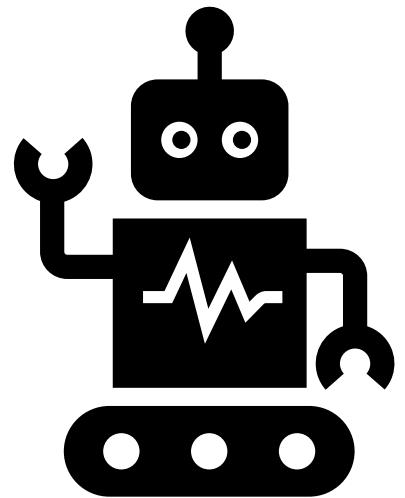
Implémenter les composants ProjectList et ProjectDetail

Implémenter le service ProjectService

Etape2:

```
ng generate component project/components/project-form
```

task



Configurer le module task

Configurer les routes pour le module task

```
ng g m task/task-routing --flat --module= task
```

Configurer le module Project

Implémenter les composants

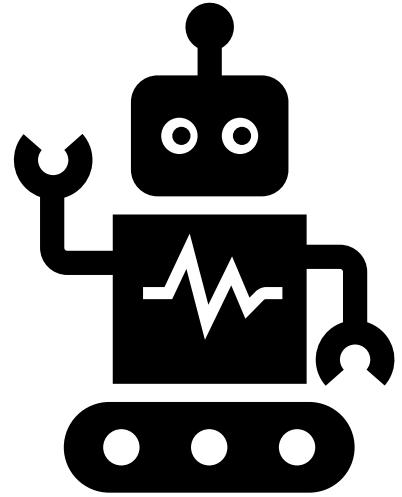
Implémenter le service

Etape2:

```
ng generate component task/components/task-form
```



calendar



Configurer le module calendar

Configurer les routes pour le module calendar

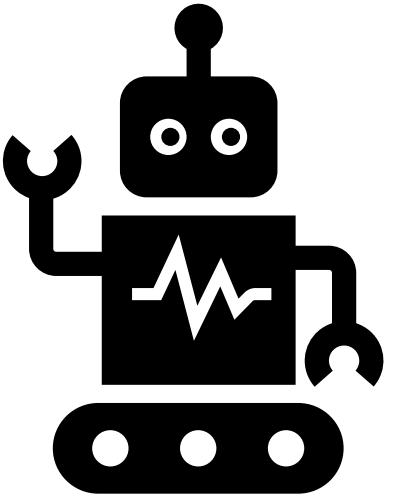
```
ng g m calendar /calendar -routing --flat --module= calendar
```

Configurer le module Project

Implémenter les composants

Implémenter le service

Supabase



Étape 1 : Configurer Supabase dans votre application Angular

Installez le SDK Supabase

```
npm install @supabase/supabase-js
```

Configurez Supabase dans votre application Angular :

Créez un fichier supabase.service.ts pour initialiser Supabase.

```
src/app/supabase.service.ts
```

```
// src/app/app-routing.module.ts
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';

const routes: Routes = [
  { path: 'projects', loadChildren: () => import('./project/project.module').then(m => m.ProjectModule) },
  { path: 'tasks', loadChildren: () => import('./task/task.module').then(m => m.TaskModule) },
  { path: 'notifications', loadChildren: () => import('./notification/notification.module').then(m => m.NotificationModule) },
  { path: 'dashboard', loadChildren: () => import('./dashboard/dashboard.module').then(m => m.DashboardModule) },
  { path: 'calendar', loadChildren: () => import('./calendar/calendar.module').then(m => m.CalendarModule) },
  { path: 'reports', loadChildren: () => import('./report/report.module').then(m => m.ReportModule) },
  { path: '**', redirectTo: 'dashboard' }
];

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

WeatherService

