

Le développement d'applications Web avec Angular



Introduction : Présentation d'Angular

Présentation d'Angular

Qu'est-ce qu'Angular ?

Angular est un framework JavaScript
qui permet de créer
des *single page applications (SPA)* réactives

Présentation d'Angular

Qu'est-ce qu'une SPA ?

Une « single page application » est une ***application web*** sur laquelle les utilisateurs vont naviguer en ayant l'impression de visiter différentes pages.

En réalité, la page ne change jamais.

C'est un ***seul fichier HTML*** qui est envoyé par le serveur au chargement de l'application, avec du code JavaScript.

C'est ce ***code JavaScript, exécuté côté client*** (par le navigateur) qui va modifier le contenu de la page HTML au cours de l'utilisation de l'application.

Présentation d'Angular

Quel est l'intérêt des SPA, et donc d'Angular ?

La SPA donne une ***expérience utilisateur très réactive*** !

L'***exécution du code JavaScript directement dans le navigateur*** est bien plus rapide que la communication avec un serveur pour chaque changement de page ou de données à afficher.

Similaire à l'***expérience des applications mobiles*** :
tout se fait instantanément.

Présentation d'Angular

Que se passe-t-il si l'on doit communiquer avec le serveur pour charger ou mettre à jour des données ?

Les communications avec le(s) serveur(s) se font via des *appels d'API*.

Ceux-ci ont lieu au moyen de *requêtes AJAX* exécutées *en arrière-plan*.
L'utilisateur ne perd pas l'expérience d'une application web réactive !

Présentation d'Angular

Les versions d'Angular

IMPORTANT : Distinguer « AngularJS » et « Angular »

AngularJS

C'est la toute première version, sortie en 2010.
Elle est aussi appelée « Angular 1 ».

Angular

La version 2 d'Angular est sortie en 2016.
Il s'agit d'une ***réécriture complète*** du framework.

Une nouvelle version sort environ tous les 6 mois. Les changements sont mineurs
d'une version à l'autre.

On parle parfois d' « Angular 2+ » (Angular 2 et toutes les versions suivantes).

Architecture, installation et premier test

Architecture, installation et premier test

Installation minimale d'Angular
Création d'un premier projet

Installation de Visual Studio Code

Optionnelle, mais recommandée.

Node.js

Utilisé par le CLI pour compiler et optimiser le projet.

NPM (Node Package Manager) sera utilisé pour gérer les différentes dépendances du projet Angular (le framework lui-même et d'autres librairies qu'il utilisera).

Installation :

<https://nodejs.org>

Télécharger la LTS

L'installer

Installation d'Angular CLI

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

Présentation d'Angular CLI

ng help

Création d'un premier projet Angular !

Navigation vers le répertoire contenant :

cd [chemin_vers_votre_répertoire]

Création du projet :

ng new my-first-project

Architecture, installation et premier test

Architecture typique d'une application Angular

Manipulations préparatoires

Ouvrir l'application avec l'éditeur de code ou l'IDE

Lancer l'application localement : ***ng serve***

Ouvrir le navigateur (de préférence Chrome) et saisir l'URL :
<http://localhost:4200>

Survol de certains fichiers de configuration

package.json

Dépendances du projet (ex : dernière version d'Angular, paquets tiers)

Dépendances de développement : utilisées seulement pour le développement

La commande « **npm install** » installe les dépendances listées dans le répertoire « node_modules » (qui n'est pas archivé avec git)

La commande « **npm install [NOM_D'UNE_DEPENDANCE]** » ajoute la dépendance dans ce fichier et l'installe

angular.json

Configurations des différents projets d'un application Angular

tsconfig.json

Configurations de TypeScript

Exploration du code source de l'application

main.ts

C'est le point d'entrée de l'application.

Il amorce le module « AppModule ».

app.module.ts

Il déclare notamment le composant « AppComponent ».

index.html

C'est la fameuse « single page ».

La balise « <app-root> », imbriquée dans « <body> », correspond au sélecteur du « AppComponent » et à l'actuel rendu visuel de l'application.

app.component.html

Explication de l'affichage dans le navigateur.

Architecture, installation et premier test

Test d'un code simple

app.component.ts

```
1 import { Component } from '@angular/core';
2
3 @Component({
4   selector: 'app-root',
5   templateUrl: './app.component.html',
6   styleUrls: ['./app.component.scss'],
7 })
8 export class AppComponent {
9   favoritePet: string = 'Dogs';
10
11   onChangeFavoritePet(): void {
12     this.favoritePet = 'Cats';
13   }
14 }
```

app.component.html

```
1  <h1>{{ favoritePet }} are my favorite pets</h1>
2  <button (click)="onChangeFavoritePet()">Prefer Cats</button>
```

TypeScript

TypeScript

TypeScript : le JavaScript typé

Installation préliminaire de TypeScript

Documentation : <https://www.typescriptlang.org>

Créer un répertoire « ts-project » et naviguer à l'intérieur

npm install typescript

Créer un fichier « example.ts »

npx tsc --init

Qu'est-ce que TypeScript ?

TypeScript est une ***surcouche de JavaScript***.

Il propose des ***fonctionnalités supplémentaires***, notamment :

- Le typage fort
- Les interfaces
- Les classes *

* mais elle existent aussi en JavaScript depuis ECMAScript 6 (2015)

Lorsqu'il est exécuté, il est seulement ***transcompilé en JavaScript***.

TypeScript n'est pas exécuté par le navigateur !

La compilation en JavaScript est prise en charge par le CLI d'Angular.

Exécution d'un fichier TypeScript : exemple

example.ts

```
1 const myNum = 3;
2
3 function addOne(num: number): number {
4   return num + 1;
5 }
```

Exécution du fichier TypeScript

```
npx tsc example.ts
```

Fichier example.js généré

```
1 var myNum = 3;
2 function addOne(num) {
3   return num + 1;
4 }
5 console.log(addOne(myNum));
```

TypeScript

Les types

Les différents types (1) : Types primitifs

```
1 // Les types primitifs : string, number, boolean
2 const myString: string = 'TypeScript';
3 const myNumber: number = 10;
4 const myBoolean: boolean = false;
```

Attention : Les noms de ces types doivent être tout en minuscule
(sinon, on pointe vers un objet).

Les différents types (2) : Types plus complexes

```
6 // Les types plus complexes : array, objet
7 const myArrayOfStrings: string[] = ['Angular', 'TypeScript', 'JavaScript'];
8 const myObject: {} = {
9   course: 'Angular',
10  section: 'TypeScript',
11  year: 2022,
12  isEasy: true,
13};
```

NB : On peut typer plus précisément un objet de la manière suivante :

```
8 const myObject: {
9   course: string;
10  section: string;
11  year: number;
12  isEasy: boolean;
13}
```

Le typage explicite : intérêt et effets

On peut typer toute variable, argument et retour de fonction.

```
1 let myVar: number;
2 myVar = 5;
3
4 function transformNumberToString(myNumber: number): string {
5   return myNumber.toString();
6 }
```

Le typage explicite interdit toute (ré)assignation d'un autre type (intentionnelle ou non).

```
1 let myVar: number;
2 myVar = '5';
3
4 function transformNumberToString(myNumber: number): string {
5   return myNumber;
6 }
```

L'inférence des types, le type « any » et les typages alternatifs

Lorsqu'on assigne une variable directement en la déclarant, on peut s'affranchir de la typer explicitement.
TypeScript *inférera* son type et le protégera quand-même !

```
1  let myVar = 100;  
2  
3  myVar = 'one hundred';  
4
```

On peut outrepasser le typage fort
au moyen du mot-clé “**any**”.

Cette solution, bien que possible, est à proscrire !

```
1  let myVar: any = 100;  
2  
3  myVar = 'one hundred';
```

On peut également recourir au **typage alternatif**.
C'est préférable à “any”, mais tout de même à éviter dans ce
type de cas.

```
1  let myVar: number | string = 100;  
2  
3  myVar = 'one hundred';
```

L'alias « type »

Certaines types,
comme les objets,
peuvent être longs
et répétitifs à expliciter.

L'alias “type” permet de déclarer un type qui sera réutilisable.

```
1 let firstCourse: {  
2   course: string;  
3   section: string;  
4   year: number;  
5   isEasy: boolean;  
6 };  
7  
8 let secondCourse: {  
9   course: string;  
10  section: string;  
11  year: number;  
12  isEasy: boolean;  
13};
```

```
1 type Course = {  
2   course: string;  
3   section: string;  
4   year: number;  
5   isEasy: boolean;  
6 };  
7  
8 let firstCourse: Course;  
9  
10 let secondCourse: Course;
```

TypeScript

Les interfaces et les classes

Les interfaces

Une interface est un **contrat** à respecter pour implémenter des objets (on parle aussi d'**abstraction**).

Elle **ne peut pas être instanciée** directement.
Elle n'a pas de constructeur.

Elle peut définir des **propriétés** et leur type respectif, ainsi que des signatures de **méthodes**.

On peut typer explicitement un objet avec le nom d'une interface.

```
1  interface IPet {
2    id: string;
3    name: string;
4    age: number;
5    species: 'dog' | 'cat' | 'rabbit' | 'fish';
6    display(): string;
7 }
```

```
9  const rex: IPet = {
10    id: '1',
11    name: 'Rex',
12    age: 4,
13    species: 'dog',
14    display(): string {
15      return `${this.name} - ${this.species} - ${this.age} years old`;
16    },
17  };
```

Les classes (1)

Une classe est une définition d'objet qui peut être *instancié* grâce à son *constructeur*.

Elle peut *implémenter une interface*.

Elle peut définir des *propriétés* - leur type, mais aussi leur *valeur* – ainsi que des *méthodes complètes*.

```
19 class Rabbit implements IPet {
20   id: string;
21   name: string;
22   age: number;
23   species: 'dog' | 'cat' | 'rabbit' | 'fish';
24
25   constructor(name: string, age: number) {
26     this.id = Math.floor(Math.random() * 100000).toString();
27     this.species = 'rabbit';
28     this.name = name;
29     this.age = age;
30   }
31
32   display(): string {
33     return `A ${this.age} years old rabbit named ${this.name}`;
34   }
35 }
```

On peut instancier un objet d'une classe grâce au mot clé *new* qui appelle le constructeur :

```
37 const roger = new Rabbit('Roger', 2);
```

Une méthode s'appelle sur un objet de la manière suivante :

```
39 const displayedRoger: string = roger.display(); // A 2 years old rabbit named Roger
```

Les classes (2) : Les modificateurs d'accès

Par défaut, les propriétés et méthodes d'une classe sont "public", mais on peut les rendre "**private**".

```
1 class Car {  
2   id: string;  
3   private brand: string;  
4  
5   constructor(brand: string) {  
6     this.id = Math.floor(Math.random() * 100000).toString();  
7     this.brand = brand;  
8   }  
9 }
```

Une propriété ou méthode "private" n'est *pas accessible en dehors de la classe.*

```
11 const myCar = new Car('Peugeot');  
12 myCar.brand = 'Rolls Royce';
```

Les modificateurs d'accès peuvent aussi servir de raccourci pour **déclarer une propriété directement dans le constructeur.**

```
1 class Car {  
2   id: string;  
3  
4   constructor(public brand: string) {  
5     this.id = Math.floor(Math.random() * 100000).toString();  
6   }  
7 }
```

NB : Cette technique est très fréquemment utilisée en Angular pour l'*injection de dépendance*.

```
9 const myNewCar = new Car('Rolls Royce');  
10 console.log(myNewCar.brand); // Rolls Royce
```

TypeScript

Les imports et exports
(modules TypeScript)

Les imports et exports TypeScript (1)

Créer un fichier “cat.ts” et définir la classe **Cat**

```
1 class Cat {  
2     constructor() {  
3         console.log('Miaou');  
4     }  
5 }
```

Instancier un **Cat** dans “example.ts”

```
1 new Cat();
```

Exécuter les fichiers TypeScript, puis “example.js” avec Node

```
npx tsc  
node example.js
```

Erreur : **Cat** n'est pas accessible dans “example.ts”

```
ReferenceError: Cat is not defined
```

Les imports et exports TypeScript (2)

Export TypeScript

```
1  export class Cat {  
2    constructor() {  
3      console.log('Miaou');  
4    }  
5  }
```



Erreur de compilation

```
1  new Cat();
```



Import TypeScript

```
1  import { Cat } from "./cat";  
2  
3  new Cat();
```

Exécution des fichiers

```
npx tsc  
node example.js
```



Constructeur appelé !

Miaou

TypeScript

L'asynchrone et les promesse

Définitions

La ***programmation asynchrone*** est une technique qui permet à un programme de démarrer une tâche à l'exécution potentiellement longue et, au lieu d'avoir à attendre la fin de la tâche, de pouvoir continuer à réagir aux autres évènements pendant l'exécution de cette tâche. Une fois la tâche terminée, le programme en reçoit le résultat.

Une ***promesse*** est un objet qui représente la complétion ou l'échec d'une opération asynchrone.

Le code asynchrone

Une variable ***data*** est déclarée mais non définie.



```
1 let data;
```

Une fonction ***createDataAsync*** attribue une valeur à ***data*** au bout d'une seconde. Cette fonction est asynchrone.



```
3 function createDataAsync() {  
4   setTimeout(() => {  
5     data = 'data';  
6   }, 1000);  
7 }
```

Une fonction synchrone ***displayData*** affiche la variable ***data*** en console.



```
9 function displayData() {  
10  console.log(data);  
11 }
```

La fonction asynchrone est appelée avant la fonction synchrone.



```
13 createDataAsync();  
14 displayData();
```

Quand on exécute le code, la console affiche pourtant “undefined”.



undefined

displayData a été appelée avant que l'exécution de ***createDataAsync*** ne soit terminée.
Aucune valeur n'a donc eu le temps d'être attribuée à ***data***.

La gestion de l'asynchrone avec des callbacks

createDataAsync prend désormais une callback en argument.

La callback n'est appelée qu'une fois qu'une valeur est attribuée à *data*.

```
3  function createDataAsync(cb) {  
4    setTimeout(() => {  
5      data = 'data';  
6      cb();  
7    }, 1000);  
8 }
```

Cette fois-ci, *displayData* n'est plus appelée directement, mais en callback de *createDataAsync*. *displayData* ne sera donc appelée que lorsque *data* aura une valeur.

```
14  createDataAsync(displayData);
```

Lorsqu'on exécute le code, la valeur de *data* est bien affichée en console.

data

Le problème de la gestion de l'asynchrone avec des callbacks

En ajoutant de la complexité, la gestion de l'asynchrone avec des callbacks devient très vite laborieuse.

Ici, on crée une nouvelle fonction ***updateDataAsync*** qui a besoin d'accéder à la variable ***data*** pour la mettre à jour. Ensuite, on veut afficher la variable mise à jour avec ***displayData***, qui a donc besoin de la valeur mise à jour par ***updateDataAsync***.

updateDataAsync doit donc être appelée en callback de ***createDataAsync***. Mais ***displayData*** doit être appelée en callback de ***updateDataAsync***, et donc en 2e callback de ***createDataAsync*** pour pouvoir être passée à la 1ère ...

Le code pourrait devenir encore beaucoup plus compliqué si on multipliait les callbacks ou si ***displayData*** prenais des arguments en paramètres, qui devraient être passés d'abord à ***createDataAsync***, puis à ***updateDataAsync*** ...

La solution à ce problème ... ce sont les [promesses](#) !

```
1  let data;
2
3  function createDataAsync(cb, cb2) {
4    setTimeout(() => {
5      data = 'data';
6      cb(cb2);
7    }, 1000);
8  }
9
10 function updateDataAsync(cb) {
11   setTimeout(() => {
12     data = 'updated ' + data;
13     cb();
14   }, 1000);
15 }
16
17 function displayData() {
18   console.log(data);
19 }
20
21 createDataAsync(updateDataAsync, displayData);
```

La gestion de l'asynchrone avec des promesses (1)

Modifions d'abord le code de *createDataAsync* et *updateDataAsync* :

Chacune de ces fonctions ne prend plus de callback en paramètres.

À la place, elle retourne une promesse instanciée grâce à son constructeur (mot-clé “new”).

Le constructeur de **Promise** prend une fonction en paramètre, qui prend elle même une callback **resolve** (convention) en 1er paramètre.

resolve est ici appelée une fois que la valeur de **data** a été attribué ou mise à jour.

```
3  function createDataAsync() {
4    return new Promise((resolve) => {
5      setTimeout(() => {
6        data = 'data';
7        resolve();
8      }, 1000);
9    });
10 }
```

```
12 function updateDataAsync() {
13   return new Promise((resolve) => {
14     setTimeout(() => {
15       data = 'updated ' + data;
16       resolve();
17     }, 1000);
18   });
19 }
```

La gestion de l'asynchrone avec des promesses (2)

La méthode ***then*** d'un objet ***Promise*** permet d'appeler une fonction passée en paramètre, seulement si et quand la promesse sera résolue (c'est-à-dire que ***resolve*** aura été appelée).

Cette méthode peut elle-même retourner une nouvelle promesse, ce qui permet de chaîner plusieurs promesses de manière successive : chacune attend la résolution de la précédente avant d'être exécutée.

Pour cela, il ne faut pas oublier de retourner la nouvelle promesse à chaque fois !



```
25  createDataAsync()
26  .then(() => {
27    return updateDataAsync();
28  })
29  .then(() => {
30    displayData();
31 });
```

En exécutant à nouveau le code, les données affichées sont bien à jour.



updated data

Le rejet des promesses (1)

Simulons un dysfonctionnement du serveur.



```
2 let isServerDown = true;
```

On passe une fonction **reject** (convention) en 2e paramètre de la callback du constructeur de **Promise**.



On appelle **reject** si le serveur ne marche pas.

```
4 function createDataAsync() {
5   return new Promise((resolve, reject) => {
6     setTimeout(() => {
7       if (isServerDown) {
8         reject('Data creation failed.');
9       }
10      data = 'data';
11      resolve();
12    }, 1000);
13  });
14}
```

On exécute le code.
La méthode **reject** a renvoyé une erreur.



```
[UnhandledPromiseRejection: This error originated either by throwing inside of an async function without a catch block, or by rejecting a promise which was not handled with .catch(). The promise rejected with the reason "Data creation failed.".] {
  code: 'ERR_UNHANDLED_REJECTION'
}
```

Le rejet des promesses (2) : « catch »

Répétons la même opération sur *updateDataAsync* avec un message d'erreur différent.



```
16 function updateDataAsync() {  
17   return new Promise((resolve) => {  
18     setTimeout(() => {  
19       if (isServerDown) {  
20         reject('Data update failed.');//  
21       }  
22       data = 'updated ' + data;  
23       resolve();  
24     }, 1000);  
25   });  
26 }
```

À la fin du chaînage des méthodes *then*, on appelle la méthode *catch*.



```
32 createDataAsync()  
33   .then(() => {  
34     return updateDataAsync();  
35   })  
36   .then(() => {  
37     displayData();  
38   })  
39   .catch((err) => [  
40     console.log(err);  
41   ])
```

La méthode *catch* attrape la première erreur renvoyée par *reject* dans le chaînage des promesses.



Data creation failed.

Le résolution et le rejet des promesses : « finally »

Ajoutons un appel à la méthode ***finally*** à la fin de celui à la méthode catch.



```
32  createDataAsync()
33  .then(() => {
34    return updateDataAsync();
35  })
36  .then(() => {
37    displayData();
38  })
39  .catch((err) => {
40    console.log(err);
41  })
42  .finally(() => {
43    console.log('finally');
44});
```

En exécutant le code,
finally est appelée après le rejet.



Data creation failed.
finally

Réparons notre serveur fictif.



```
2 let isServerDown = false;
```

finally est appelée aussi
après la résolution.



updated data
finally

La simplification des promesses : « `async` » et « `await` »

À la place du chaînage des `then`, on peut simplement utiliser la syntaxe `async/await`.

Pour fonctionner, le mot-clé `await` doit forcément apparaître à l'intérieur d'une fonction déclarée asynchrone avec le mot-clé `async`.

Le mot-clé `await` permet d'attendre la fin de l'exécution de la ligne avant de passer à la ligne suivante.

Pour gérer les rejets et le code exécuté après toute résolution ou rejet, on utilise alors les mots-clés `try/catch/finally`.

```
32  async function displayUpdatedData() {  
33    await createDataAsync();  
34    await updateDataAsync();  
35    displayData();  
36  }  
37  
38  displayUpdatedData();
```

```
32  async function displayUpdatedData() {  
33    try {  
34      await createDataAsync();  
35      await updateDataAsync();  
36      displayData();  
37    } catch (err) {  
38      console.log(err);  
39    } finally {  
40      console.log('finally');  
41    }  
42  }
```

Le retour des résolutions de promesse

Les promesses résolues peuvent retourner une valeur. Cette valeur est une propriété du type **Promise** et elle a un type générique (d'où la syntaxe du typage de la fonction **getDataAsync**).

La valeur de retour est passée en argument de **resolve**.

Lors du chaînage avec **then**, cette valeur de retour peut-être passée en argument de la callback.

Quand au mot-clé **await** suivi de la promesse, il renvoie directement la valeur de retour.

(NB : Une fonction **async** renvoie toujours une **Promise**.)

```
1  function getDataAsync(): Promise<number> {
2    return new Promise((resolve) => {
3      resolve(3);
4    });
5 }
```

```
7  getDataAsync().then((data) => {
8    console.log(data);
9 });
```

```
7  async function displayData(): Promise<void> {
8    const data = await getDataAsync();
9    console.log(data);
10 }
11
12 displayData();
```

Aperçu des modules Angular

Définition

Une application Angular est modulaire : Elle est constituée de modules appelés ***NgModules***.

Un NgModule est un ***conteneur pour un bloc de code*** (souvent relatif à un domaine d'application). Il peut contenir des composants, des services, des directives, et d'autres fichiers de code dont il définit la portée.

Angular analyse les NgModules pour « comprendre » l'application et ses fonctionnalités. Une application requière donc au moins un module (le ***AppModule***) pour fonctionner.

Mais elle sera souvent divisée en de multiples modules : Un NgModule peut ***importer*** des fonctionnalités exportées par d'autres et ***exporter*** des fonctionnalités importées par d'autres.

Exemple : Le *AppModule*

```
1 √ import { NgModule } from '@angular/core';
2   import { BrowserModule } from '@angular/platform-browser';
3
4   import { AppComponent } from './app.component';
5
6 √ @NgModule({
7   declarations: [AppComponent],
8   imports: [BrowserModule],
9   providers: [],
10  bootstrap: [AppComponent],
11})
12 export class AppModule {}
```

Le rôle majeur des composants

Rôle majeur des composants

Définition d'un composant Angular

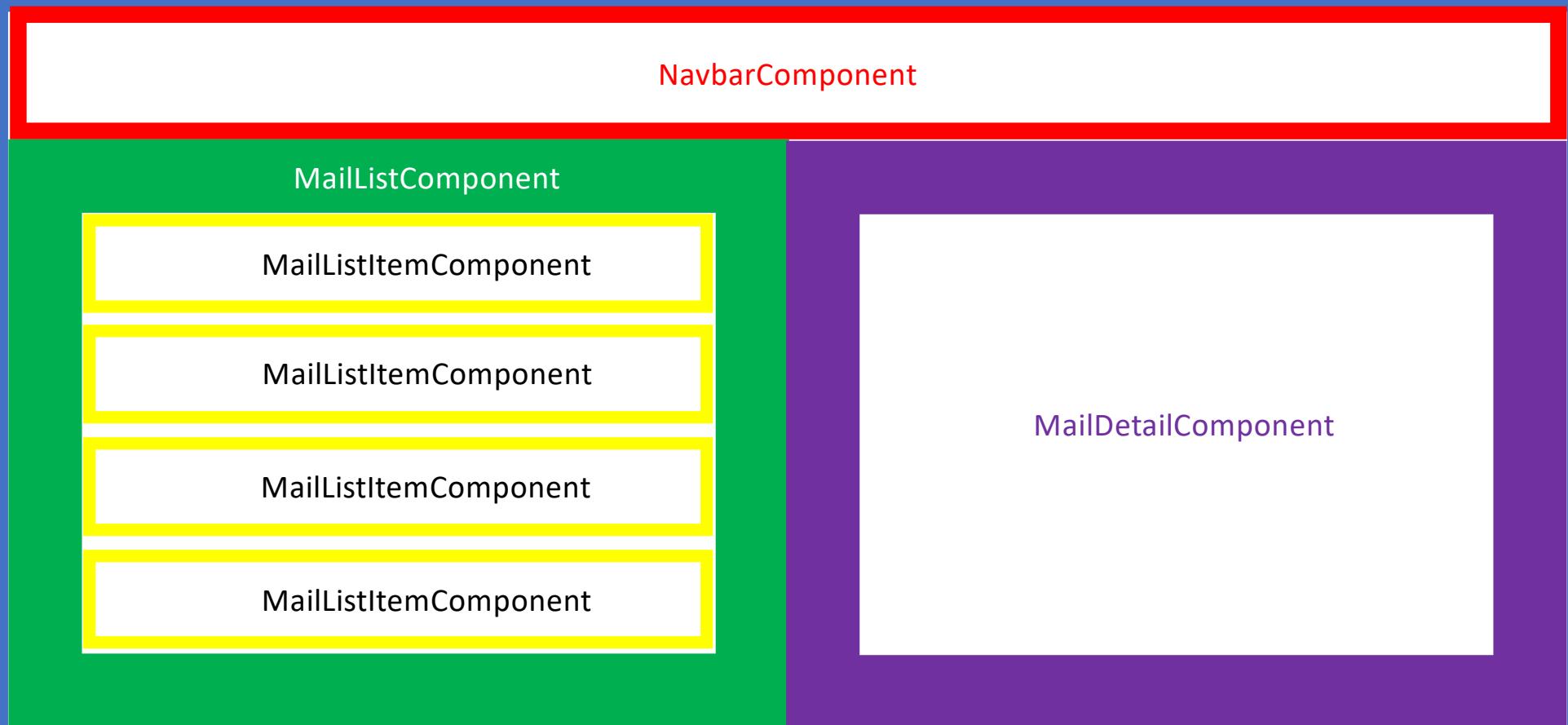
Les composants sont les *éléments de base de l'interface utilisateur* d'une application Angular. Une application Angular contient un arbre de composants. Ce sont des *unités de découpage visuel*.

Les composants Angular sont un sous-ensemble de *directives* (fichiers TypeScript), toujours associées à un *template* (fichier HTML). Ils peuvent aussi avoir leur propre *feuille de style* (fichier CSS).

Classiquement, un composant est instancié par la balise de son *sélecteur* dans le template d'un autre composant.

Un composant *doit appartenir à un NgModule* pour qu'il soit disponible pour un autre composant ou une autre application.

AppComponent



Un composant est une ***classe TypeScript***
avec des attributs, des méthodes et un constructeur.

Il possède aussi des ***métadonnées*** qu'on lui injecte
via le ***décorateur @Component***.

Rôle majeur des composants

Le décorateur d'un composant

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

Pour utiliser le décorateur ***Component***,
Il est nécessaire de l'importer avec TypeScript
depuis la librairie “@angular/core”

Le sélecteur du composant :

Il est déclaré grâce à la propriété **selector**.

Il est utilisé pour instancier le composant à partir du template d'un autre composant.

Ici, ses balises seraient `<app-navbar></app-navbar>`.

Par défaut, il porte le nom **app-[nom-du-composant]**.

Il est conseillé de garder son suffixe pour le faire toujours correspondre au nom du composant.

On peut changer son préfixe en fonction du module auquel il appartient.

Il est optionnel (un composant peut être instancié autrement), mais fortement recommandé !

Le template du composant :

Il est déclaré soit par **templateUrl** (le chemin relatif d'un fichier HTML), soit par **template** (un contenu HTML directement inséré dans le décorateur en tant que chaîne de caractères). La première solution est vivement préférable !

Il s'agit du rendu visuel du composant.

Sa déclaration dans le décorateur, par l'une des 2 propriétés ci-dessus, est obligatoire.

Les feuilles de style du composant :

Elles sont déclarées via la propriété **styleUrls**, qui prend un tableau de chemins relatifs vers des fichiers CSS.

Il s'agit du CSS appliqué uniquement à ce composant (il est par défaut encapsulé).

Un composant peut avoir plusieurs feuilles de styles bien que cette pratique soit rarissime.

Elles sont optionnelles (mais il est recommandé d'en avoir une).

```
3  @Component({
4    selector: 'app-navbar',
5    templateUrl: './navbar.component.html',
6    styleUrls: ['./navbar.component.scss'],
7  })
8  export class NavbarComponent {}
```

Rôle majeur des composants

La classe TypeScript d'un composant

Les propriétés d'un composant

Comme toute classe TypeScript, un composant peut posséder des propriétés.

Une propriété est d'abord déclarée simplement avec son nom, sans mot-clé de type “var” ou “this”.

Par convention, on déclare toutes les propriétés au début d'une classe, avant le constructeur.

La propriété est ensuite accessible par toute la classe, avec le mot-clé **this**.

Les propriétés déclarées dans la classe du composant sont également accessibles depuis le template de ce seul composant. Pour cela, le mot-clé “this” n'est pas nécessaire.

```
8  export class ExampleComponent {  
9    data: string = 'My Component Property';  
10     
11   constructor() {  
12     console.log(this.data)  
13   }  
14 }
```

```
1   <p>{{ data }}</p>
```

Les méthodes d'un composant

Comme toute classe TypeScript, un composant peut posséder des méthodes. Une méthode est d'abord déclarée avec son nom. Par convention, on les déclare après le constructeur (et souvent après les méthodes de cycle de vie étudiées plus loin). La méthode est ensuite accessible par toute la classe, avec le mot-clé **this**.

```
11  constructor() {}
12
13  updateData(newData: string): void {
14    this.data = newData;
15  }
16 }
```

Comme les propriétés,
les méthodes sont accessibles depuis le template de leur composant,
sans le mot-clé “this”.

```
2  <button (click)="updateData('My Updated Property')">Update</button>
```

Les modificateurs d'accès des propriétés et méthodes d'un composant

Comme pour toute classe TypeScript,
les méthodes et propriétés des composants peuvent avoir un modificateur d'accès,
qui est **public** par défaut.

Mais en Angular, ils sont surtout utilisés pour autoriser ou interdire l'accès par le template :
Une propriété ou méthode privée n'est pas accessible depuis le template du composant.

```
13  private updateData(newData: string): void {  
14    |  this.data = newData;  
15  }
```

```
2  <button (click)="updateData('My Updated Property')">Update</button>
```

Les méthodes du cycle de vie d'un composant (1)

Chaque *instance d'un composant* a un *cycle de vie*.

- Il commence lorsque Angular instancie la classe du composant et effectue le rendu de la vue du composant et de ses vues enfant.
- Il se poursuit avec la détection des changements, Angular vérifiant si les propriétés liées aux données changent et mettant à jour la vue et l'instance de composant si nécessaire.
- Il se termine lorsque Angular détruit l'instance du composant et supprime son modèle rendu du DOM.

Une application Angular peut utiliser des *méthodes de cycle de vie* pour exploiter les événements clés du cycle de vie d'un composant afin :

- d'initialiser de nouvelles instances,
- de lancer la détection des changements si nécessaire,
- de répondre aux mises à jour pendant la détection des changements
- et de faire du nettoyage avant la suppression des instances.

Les méthodes du cycle de vie d'un composant (2)

Les méthodes de cycle de vie sont les suivantes :

(<https://angular.io/guide/lifecycle-hooks>)

ngOnChanges()
ngOnInit()
ngDoCheck()
ngAfterContentInit()
ngAfterContentChecked()
ngAfterViewInit()
ngAfterViewChecked()
ngOnDestroy()

La plus fréquemment utilisée est ***ngOnInit()*** (elle est même créée par défaut avec le CLI).

Afin d'éviter toute erreur lorsqu'on utilise une méthode de cycle de vie,
il est très recommandé de **faire implémenter l'interface correspondante au composant.**

Les interfaces portent le même nom que les méthodes,
auquel on retire le préfixe “ng” (par exemple : l'interface ***OnInit***).

Les méthodes du cycle de vie d'un composant (3)

```
3  @Component({
4    selector: 'app-example',
5    templateUrl: './example.component.html',
6    styleUrls: ['./example.component.scss'],
7  })
8  export class ExampleComponent implements OnInit, OnDestroy {
9    data: string = 'My Component Property';
10
11  constructor() {}
12
13  ngOnInit(): void {
14    this.updateData('My New Property from OnInit');
15  }
16
17  ngOnDestroy(): void {
18    console.log('ExampleComponent has been Destroyed');
19  }
20
21  updateData(newData: string): void {
22    this.data = newData;
23  }
24}
```

Rôle majeur des composants

Le template
et les feuilles de style
d'un composant

L'interpolation

La **string interpolation**, est une technique pour intégrer des expressions JavaScript dans le template.

Elle utilise par défaut les doubles accolades `{{ }}` comme délimiteurs.

Angular remplacera leur contenu par la valeur de la chaîne de caractères correspondant.

Classiquement, on l'utilise pour intégrer la valeur d'une propriété du composant dans du texte balisé.

Le nom de la propriété s'écrit simplement entre les doubles accolades, sans le mot-clé “this”.

Elle peut aussi s'insérer en tant que valeur d'un attribut HTML.

Elle peut aussi intégrer n'importe quelle expression.

```
8  export class ExampleComponent implements OnInit {  
9    myLink: string = '';  
10   constructor() {}  
11  
12   ngOnInit(): void {  
13     this.myLink = 'https://angular.io/docs';  
14   }  
15 }  
16  
1 <div>{{ myLink }}</div>
```

```
1 <a href="{{ myLink }}">Angular documentation</a>
```

```
1 {{ 5 * 5 }}
```

Les directives structurelles (1)

Les directives structurelles sont des directives qui ***modifient la disposition du DOM*** en ajoutant et en supprimant des éléments du DOM.

Les principales sont ***NgIf***, ***NgForOf*** et ***NgSwitch***.
Les plus courantes sont les 2 premières, que l'on présentera ici.

Elles sont fournies par le ***CommonModule***,
lui-même exporté vers le ***BrowserModule***,
lui-même importé par le module racine ***AppModule***.

Elles s'écrivent en camelCase avec une étoile comme préfixe (*).

Les directives structurelles (2) : NgIf

Inclut conditionnellement un élément HTML (et tous ses descendants) en fonction de la valeur d'une expression convertie en booléen.

```
8  export class ExampleComponent {  
9    isVisible = false;  
10  
11   onToggleVisibility(): void {  
12     this.isVisible = !this.isVisible;  
13   }  
14 }
```

```
1 <div *ngIf="isVisible">J'existe !</div>  
2 <button (click)="onToggleVisibility()">Update Visibility</button>
```

Les directives structurelles (3) : NgForOf

Rend un élément HTML (et tous ses descendants) pour chaque élément d'une collection.

La directive est placée sur l'élément à cloner.

```
8  export class ExampleComponent {  
9    dogs = [  
10      {name: 'Kiki', race: 'Pitbull'},  
11      {name: 'Thor', race: 'Chihuahua'},  
12      {name: 'Rex', race: 'Teckel'},  
13      {name: 'Choupette', race: 'Doberman'},  
14    ];  
15 }
```

```
1  <ul>  
2    <li *ngFor="let dog of dogs">  
3      <h2>{{ dog.name }}</h2>  
4      <p>{{ dog.race }}</p>  
5    </li>  
6  </ul>
```

Les feuilles de style d'un composant

Elles contient le ***style applicable au seul composant*** (ni à ses parents, ni à ses enfants) !

Il est recommandé et très courant d'utiliser le format **SCSS** (Sassy Cascading Style Sheets).

Pour appliquer un style à plusieurs composants,
on utilisera le fichier **styles.scss** (ou .css, .sass, etc. selon le format choisi),
qui se trouve à la racine du code source, dans le répertoire “src” du projet.

Rôle majeur des composants

La communication entre les composants
parents et enfants

L'intégration d'un composant enfant

On utilise simplement le **sélecteur du composant enfant** comme **balise HTML** dans le **template du composant parent**.

Exemple :

AppComponent est le parent de *ExampleComponent* (qui est donc son enfant).

app-example est le sélecteur
du *ExampleComponent*.

```
3  @Component({
4    selector: 'app-example',
5    templateUrl: './example.component.html',
6    styleUrls: ['./example.component.scss'],
7  })
8  export class ExampleComponent {
```

Lors de l'affichage,
le template de *ExampleComponent* remplacera
les balises *app-example* dans le *AppComponent*.

```
Go to component
1 <h1>Mon Titre</h1>
2 <app-example></app-example>
```

Valeurs passées du parent à l'enfant : @Input

TypeScript du parent



```
8  export class AppComponent {  
9    |   sportList: string[] = ['Football', 'Tennis', 'Basketball', 'Natation'];  
10 }  
  
11  
12  
13  
14  
15
```

HTML du parent



```
1 <h1>Mon Titre</h1>  
2 <app-example [sports]="sportList"></app-example>  
3  
4  
5  
6  
7
```

TypeScript de l'enfant



```
1 import { Component, Input, OnInit } from '@angular/core';  
2  
3 export class ExampleComponent implements OnInit {  
4   |   @Input() sports: string[] = [];  
5   |   favoriteSport: string = '';  
6  
7   |   ngOnInit(): void {  
8       |       this.favoriteSport = this.sports[0];  
9   }  
10 }  
11  
12  
13  
14  
15
```

HTML de l'enfant



```
1 <div>Mon sport préféré: {{ favoriteSport }}</div>
```

Écoute des évènements de l'enfant par le parent : @Output

TypeScript de l'enfant



```
1 import { Component, EventEmitter, Input, Output } from '@angular/core';
8 export class ExampleComponent {
9   @Input() sports: string[] = [];
10  @Output() changeSport = new EventEmitter<string>();
11
12  onSelectSport(sport: string) {
13    this.changeSport.emit(sport);
14  }
15 }
```

HTML de l'enfant



```
1 <ul>
2   <li *ngFor="let sport of sports">
3     <h2>{{sport}}</h2>
4     <button (click)="onSelectSport(sport)">Définir comme Sport Préféré</button>
5   </li>
6 </ul>
```

TypeScript du parent



```
8 export class AppComponent implements OnInit {
9   sportList: string[] = ['Football', 'Tennis', 'Basketball', 'Natation'];
10  favoriteSport: string = '';
11
12  ngOnInit(): void {
13    this.favoriteSport = this.sportList[0];
14  }
15
16  onChangeFavoriteSport(event: string) {
17    this.favoriteSport = event;
18  }
19 }
```

HTML du parent



```
1 <h1>Mon Titre</h1>
2 <div>Mon sport préféré: {{ favoriteSport }}</div>
3
4 <app-example
5   [sports]="sportList"
6   (changeSport)="onChangeFavoriteSport($event)"
7 ></app-example>
```

Rôle majeur des composants

La génération d'un composant avec Angular CLI

« ng generate component ... »

Pour la commande “**ng generate component pets/pet-list**”, Angular CLI va :

- créer un répertoire “pets” - s’il n’existe pas - à la racine du répertoire “app” ;
- créer un répertoire “pet-list” dans “pets” et y générer plusieurs fichiers :
 - ***pet-list.component.ts*** qui contient un composant appelé PetListComponent
 - ***pet-list.component.html*** automatiquement défini dans le TypeScript comme le template du composant
 - ***pet-list-component.(s)css*** automatiquement défini dans le TypeScript comme la feuille de style du composant
 - ***pet-list.component.spec.ts*** (selon le paramétrage initial du CLI) qui est un fichier de test préparé pour le composant
- déclarer le PetListComponent dans le AppModule (si aucun autre module n’a été créé), ou dans le module sélectionné avec l’option “--module” (dans le cas contraire).

NB : L’option **--dry-run** permet d’afficher l’effet de la commande en console sans la lancer.
“ng generate component pets/pet-list –dry-run”

Travaux pratiques

Travaux pratiques

Projet Fil Rouge Partie 1

Les templates

Les templates

Property & Event Binding

Le *property binding* (1) : Définition

Le « property binding » est une fonctionnalité Angular permettant de **définir les valeurs des propriétés des éléments HTML**.

Pour se lier à la propriété d'un élément,
il faut la mettre **entre crochets []**,
ce qui identifie la propriété comme une propriété cible :
la propriété du DOM à laquelle on souhaite attribuer une valeur.

Elle permet aussi d'**accéder à la propriété d'un composant**
depuis son élément HTML !
C'est pour cela qu'elle fonctionne pour les inputs ...

La liaison de propriétés déplace une valeur dans un sens,
de la propriété d'un composant vers la propriété d'un élément cible.

Le *property binding* (2) : Exemple

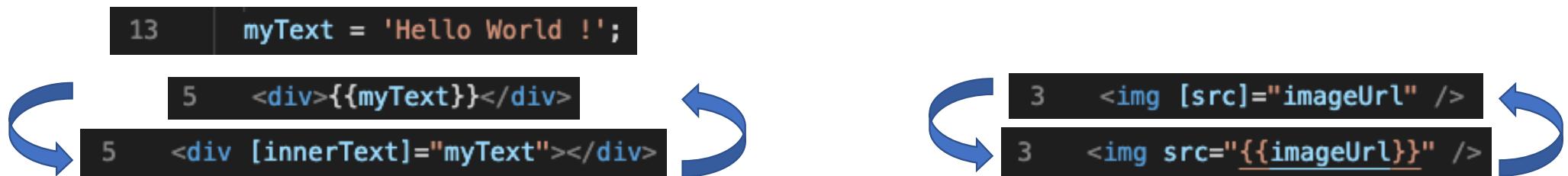
```
8  export class ExampleComponent implements OnInit {
9    isFormValid: boolean = false;
10   imageUrl =
11     'https://upload.wikimedia.org/wikipedia/commons/thumb/c/cd/Stray_kitten_Rambo002.jpg/1200px-Stray_kitten_Rambo002.jpg';
12
13  constructor() {}
14
15  ngOnInit(): void {
16    setTimeout(() => {
17      this.isFormValid = true;
18    }, 3000);
19  }
20}
```

```
1    <button [disabled]="!isFormValid">Click Me</button>
```

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

Le *property binding* (3) : Comparaison avec l'interpolation

Le *property binding* et l'*interpolation* sont souvent interchangeables :



Sémantiquement, il est cependant préférable d'utiliser :

L'*interpolation* : pour afficher du texte.

Le *property binding* : pour modifier une propriété.

```
5  <div>{{myText}}</div>
```

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

Les deux ne se cumulent pas (le *property binding* prend déjà une expression TypeScript).

```
5  <div [innerText]="{{myText}}></div>
```

L' *event binding* (1) : Définition

L' « event binding » est une fonctionnalité Angular qui permet ***d'écouter et de répondre aux actions de l'utilisateur*** (frappes au clavier, mouvements de la souris, clics, touchers, etc.).

Pour se lier à un événement, on utilise le nom de l'événement cible ***entre parenthèses ()***. L'évènement lui-même est accessible dans le template grâce au mot-clé ***\$event***.

Il permet aussi d'***accéder aux évènements d'un composant*** depuis son élément HTML ! C'est pour cela qu'il fonctionne pour les outputs ...

L' *event binding* (2) : Exemple

TypeScript

```
12  isImageVisible: boolean = false;
13
14  onToggleImage(event: any): void {
15    this.isImageVisible = !this.isImageVisible;
16    console.log(event);
17 }
```

HTML

```
1  <button (click)="onToggleImage($event)">Click Me</button>
2
3  <img [hidden]="!isImageVisible" [src]="imageUrl" />
```

Évènement en console

```
example.component.ts:15
▶ PointerEvent {isTrusted: true, pointerId: 1, width: 1, height: 1, pressure:
  0, ...}
```

Les templates

Utiliser des variables locales

Les *variables locales* (1) : Définition

La variable locale (ou « template variable ») est une fonctionnalité Angular qui permet d'*utiliser les données d'une partie du template dans une autre partie de celui-ci.*

Elle peut être utilisée, par exemple, pour *répondre aux entrées de l'utilisateur ou paramétrer les formulaires.*

Elle peut faire *référence* aux éléments suivants :

- un élément DOM dans un template
- une directive ou un composant
- un TemplateRef d'un ng-template
 - un composant Web

Dans le template, on utilise le symbole **dièse #**, pour déclarer une variable locale.

Les *variables locales* (2) : Exemple

TypeScript

```
8  export class ExampleComponent {  
9    myTitle = '';  
10     
11   onClick(newTitle: string): void {  
12     this.myTitle = newTitle;  
13   }  
14 }
```

HTML avec la variable locale *myInput*

```
1  <h1>{{ myTitle }}</h1>  
2  <input type="text" #myInput />  
3  <button (click)="onClick(myInput.value)">Update Title</button>
```

Les templates

Utiliser des pipes

Les *pipes* (1) : Définition

Le pipe est une fonctionnalité Angular qui permet de **transformer des données à afficher** :

- chaînes de caractères
- montants en devises
 - Dates
 - etc.

C'est une **fonction utilisée dans le template**
pour accepter une **valeur d'entrée** et renvoyer une **valeur transformée**.

Pour appliquer un pipe, on utilise le caractère **pipe /** dans une expression du template, suivi du nom du pipe.

Les pipes peuvent parfois prendre des **paramètres** :

On utilise alors le caractère « **deux points** » (:) entre le nom du pipe et la valeur de ses paramètres.

Si le pipe accepte plusieurs paramètres, leurs valeurs sont aussi séparées par des « : ».

Les *pipes* (2) : Pipes intégrés à Angular

Angular fournit des pipes intégrés pour les transformations de données typiques :

<https://angular.io/api/common#pipes>

```
8  export class ExampleComponent {
9    myProduct = {
10      name: 'Aspirateur',
11      creationDate: new Date(),
12      price: 99.9876,
13      description:
14        'Une très longue description dont je ne veux afficher que 20 caractères.',
15    };
16  }

1  <!-- Sans paramètre -->
2  <h1>{{ myProduct.name | uppercase }}</h1>
3
4  <!-- 1 paramètre -->
5  <p>{{ myProduct.price | currency: "EUR" }}</p>
6  <p>{{ myProduct.creationDate | date: "dd/MM/yyyy" }}</p>
7
8  <!-- 2 paramètre -->
9  <p>{{ myProduct.description | slice: 0:20 }}</p>
```

Les *pipes* (3) : Créer des pipes personnalisés

On peut également créer soi-même des pipes pour encapsuler des transformations personnalisées, puis les utiliser dans des expressions de templates.

ng generate pipe ...

```
1 import { Pipe, PipeTransform } from '@angular/core';
2
3 @Pipe({
4   name: 'shorten',
5 })
6 export class ShortenPipe implements PipeTransform {
7   transform(value: string, ...args: number[]): string {
8     let start = 0;
9     let end = 3;
10
11    if (args.length >= 2) {
12      start = args[0];
13      end = args[1];
14    }
15
16    return value.split('').splice(start, end).join('') + '...';
17  }
18}
```

```
9   <p>{{ myProduct.description | shorten: 0:20 }}</p>
```

Les services

Les services

Qu'est-ce qu'un service ?

Les *services* : Définition

Un service
est une ***classe dont l'objectif est défini par rapport à une logique métier.***
Il doit avoir une responsabilité bien spécifique.

Angular distingue les composants des services
afin d'accroître la modularité et la réutilisabilité :
Alors qu'un composant sert d'intermédiaire entre la vue et la logique applicative
il doit ***utiliser des services pour les tâches qui n'impliquent pas la vue.***

Les services

À quoi sert un service ?

Les *services* : Utilité

Les services sont utiles pour des tâches telles que, par exemple :

- l'*extraction* ou l'*insertion de données* depuis ou vers le serveur
- le *partage des données* de l'application entre les composants
 - le *traitement des données* de l'application
 - la factorisation et l'encapsulation d'un *code métier*

Les services

La champ d'application (« scope ») des services

Les *services* : « scope »

Les services sont fournis (« provided ») grâce à des *injecteurs*.
Un service a autant d'instances que d'injecteurs auxquels il est fourni.

Ils sont généralement fournis aux injecteurs des *modules*,
c'est-à-dire à des parties de l'application.

Plus rarement, ils sont fournis directement
aux injecteurs des *composants* ou des *directives*.

Services « singleton » :

Quand un service est fourni au *module racine* (« *root* »),
c'est-à-dire le module « *app* »,
il n'a qu'*une seule instance pendant toute durée de vie de l'application*.

On dit alors qu'il est « *singleton* ».

C'est le cas de la plupart des services.

Un service créé depuis le CLI est singleton par défaut.

Il est indispensable de créer des services singleton
pour partager des données dans toute l'application.

Les services

La création de services

Création d'un service singleton

ng generate service ...

ng generate service product

```
1 import { Injectable } from '@angular/core';
2
3 @Injectable({
4   providedIn: 'root'
5 })
6 export class ProductService {
7
8   constructor() { }
9 }
```

Les services

L'injection de services

L'injection de *services* : injection de dépendance (1)

Un service est injecté dans un composant (ou une directive) via l'*injection de dépendance*.

Il peut également être injecté dans un autre service :
Attention : À utiliser très précautionneusement, voire à éviter (références circulaires).

Un service est instancié quand un composant (ou une directive) dans lequel il est injecté est instancié pour la première fois.

L'injection de *services* : injection de dépendance (2)

Démonstration :

Communication entre 2 composants « product-list » et « product-detail »
via un service « product »

L'injection de *services*

```
src > app > A product.service.ts > ProductService
  1 import { Injectable } from '@angular/core';
  2
  3 @Injectable({
  4   providedIn: 'root',
  5 }
  6 export class ProductService {
  7   private _products = [
  8     { id: 1, name: 'Voiture' },
  9     { id: 2, name: 'Avion' },
 10     { id: 3, name: 'Bateau' },
 11   ];
 12
 13   private _selectedProductId: number | null = null;
 14
 15   get products() {
 16     return this._products.map((product) => {
 17       return { ...product };
 18     });
 19   }
 20
 21   get selectedProduct() {
 22     return {
 23       ...this._products.find(
 24         (product) => product.id === this._selectedProductId
 25       ),
 26     };
 27   }
 28
 29   selectProduct(productId: number) {
 30     this._selectedProductId = productId;
 31   }
 32 }
```

L'injection de *services*

```
src > app > product-list > A product-list.component.ts > ...
1 import { Component } from '@angular/core';
2
3 import { ProductService } from '../product.service';
4
5 @Component({
6   selector: 'app-product-list',
7   templateUrl: './product-list.component.html',
8   styleUrls: ['./product-list.component.scss'],
9 })
10 export class ProductListComponent {
11   constructor(private productService: ProductService) {}
12
13   get products() {
14     return this.productService.products;
15   }
16
17   onSelectProduct(productId: number): void {
18     this.productService.selectProduct(productId);
19   }
20 }
```

```
src > app > product-list > B product-list.component.html > ...
Go to component
1 <ul>
2   <li *ngFor="let product of products" (click)="onSelectProduct(product.id)">
3     {{ product.name }}
4   </li>
5 </ul>
```

L'injection de *services*

```
src > app > product-detail > A product-detail.component.ts > ...
1 import { Component } from '@angular/core';
2
3 import { ProductService } from '../product.service';
4
5 @Component({
6   selector: 'app-product-detail',
7   templateUrl: './product-detail.component.html',
8   styleUrls: ['./product-detail.component.scss'],
9 })
10 export class ProductDetailComponent {
11   constructor(private productService: ProductService) {}
12
13   get selectedProductName() {
14     return this.productService.selectedProduct.name;
15   }
16 }
```

```
src > app > product-detail > B product-detail.component.html > ...
Go to component
1 <p>{{ selectedProductName }}</p>
2
```

L'injection de *services*

L'injection de *services*

L'injection de *services*

Travaux pratiques

Travaux pratiques

Projet Fil Rouge Partie 2

Les formulaires

Création de formulaires

Le FormsModule

Module du paquet « @angular/forms »

Il exporte les fournisseurs (« providers ») et les directives nécessaires pour les ***formulaires pilotés par des templates (« template-driven »)***, les rendant disponibles à l'importation par les NgModules qui importent ce module.

Le FormsModule

La directive ***ngModel*** est au cœur des « template-driven forms ».

Elle permet principalement
de "binder" dans les deux sens le "model" avec la "view".
C'est ce que l'on appelle le ***two-way binding***.

Le FormsModule

Démo :

Création d'un template-driven form dont on affiche les valeurs à la soumission.

3 champs :

- Prénom
- Nom
- Âge

Le FormsModule

```
src/app/app.module.ts
1  import { NgModule } from '@angular/core';
2  import { BrowserModule } from '@angular/platform-browser';
3  import { FormsModule } from '@angular/forms';
4
5  import { AppComponent } from './app.component';
6
7  @NgModule({
8    declarations: [AppComponent],
9    imports: [BrowserModule, FormsModule],
10   providers: [],
11   bootstrap: [AppComponent],
12 })
13 export class AppModule {}
```

Le FormsModule

```
1 import { Component } from '@angular/core';
2 import { NgForm } from '@angular/forms';
3
4 @Component({
5   selector: 'app-root',
6   templateUrl: './app.component.html',
7   styleUrls: ['./app.component.scss'],
8 })
9 export class AppComponent {
10   person = { firstName: '', lastName: '', age: 0 };
11
12   constructor() {}
13
14   onSubmit(ngForm: NgForm) {
15     this.person = ngForm.form.value;
16   }
17 }
```

Le FormsModule

```
1 <div>{{ person.firstName + " " + person.lastName }}</div>
2 <div *ngIf="person.age">{{ person.age }} ans</div>
3
4 <form #signUpForm="ngForm" (ngSubmit)="onSubmit(signUpForm)">
5   <input type="text" placeholder="Prénom" name="firstName" ngModel />
6   <input type="text" placeholder="Nom" name="lastName" ngModel />
7   <input type="number" placeholder="Âge" name="age" ngModel />
8   <button>Submit</button>
9 </form>
```

Le FormBuilder

Classe fournie par le module ***ReactiveFormsModule***
du paquet « @angular/forms »

Il s'injecte dans les composants pour créer
des ***FormControl***, ***FormGroup*** ou ***FormArray***
à partir du modèle de ces composants.

Le FormBuilder

Les *formulaires réactifs* offrent une *approche guidée par le modèle* pour traiter les entrées de formulaire dont les valeurs changent au fil du temps.

Le FormBuilder

Démo :

Transformation du template-driven form en reactive-form.

Le FormBuilder

```
1 import { NgModule } from '@angular/core';
2 import { ReactiveFormsModule } from '@angular/forms';
3 import { BrowserModule } from '@angular/platform-browser';
4
5 import { AppComponent } from './app.component';
6
7 @NgModule({
8   declarations: [AppComponent],
9   imports: [BrowserModule, ReactiveFormsModule],
10  providers: [],
11  bootstrap: [AppComponent],
12})
13 export class AppModule {}
```

Le FormBuilder

```
1 import { Component } from '@angular/core';
2 import { FormBuilder, FormGroup } from '@angular/forms';
3
4 @Component({
5   selector: 'app-root',
6   templateUrl: './app.component.html',
7   styleUrls: ['./app.component.scss'],
8 })
9 export class AppComponent {
10   person = { firstName: '', lastName: '', age: 0 };
11   form: FormGroup = this.fb.group({
12     firstName: '',
13     lastName: '',
14     age: '',
15   });
16
17   constructor(private fb: FormBuilder) {}
18
19   onSubmit() {
20     this.person = this.form.value;
21   }
22 }
```

Le FormBuilder

```
1 <div>{{ person.firstName + " " + person.lastName }}</div>
2 <div *ngIf="person.age">{{ person.age }} ans</div>
3
4 <form [formGroup]="form" (ngSubmit)="onSubmit()">
5   <input type="text" placeholder="Prénom" formControlName="firstName" />
6   <input type="text" placeholder="Nom" formControlName="lastName" />
7   <input type="number" placeholder="Âge" formControlName="age" />
8   <button>Submit</button>
9 </form>
```

Les formulaires

Validation & gestion des erreurs

Validation & gestion des erreurs

Un **validateur** est une fonction (@angular/forms) qui traite un FormControl ou une collection de contrôles et renvoie une map d'erreurs ou nulle. Une map nulle signifie que la validation a réussi.

Validation & gestion des erreurs

Démo :

Ajout de 2 champs avec validateurs :

- email : requis, avec le bon format
- password : requis, avec au moins 6 caractères

Quand le formulaire est invalide :

- On n'affiche pas le résultat à la soumission
- On affiche un message d'erreur personnalisé en dessous du champ concerné
 - On désactive le bouton de soumission

Validation & gestion des erreurs

```
1 import { Component } from '@angular/core';
2 import { FormBuilder, FormGroup, Validators } from '@angular/forms';
3
4 @Component({
5   selector: 'app-root',
6   templateUrl: './app.component.html',
7   styleUrls: ['./app.component.scss'],
8 })
9 export class AppComponent {
10   person = { email: '', password: '', firstName: '', lastName: '', age: 0 };
11   form: FormGroup = this.fb.group({
12     email: ['', [Validators.required, Validators.email]],
13     password: ['', [Validators.required, Validators.minLength(6)]],
14     firstName: '',
15     lastName: '',
16     age: '',
17   });
18
19   constructor(private fb: FormBuilder) {}
20
21   onSubmit() {
22     if (this.form.valid) {
23       this.person = this.form.value;
24     }
25   }
26 }
```

Validation & gestion des erreurs

```
1  <div>{{ person.email }}</div>
2  <div>{{ person.firstName + " " + person.lastName }}</div>
3  <div *ngIf="person.age">{{ person.age }} ans</div>
4
5  <form [formGroup]="form" (ngSubmit)="onSubmit()">
6    <input type="email" placeholder="Email" formControlName="email" />
7    <div *ngIf="form.get('email')?.hasError('required')">
8      | L'email est obligatoire.
9    </div>
10   <div *ngIf="form.get('email')?.hasError('email')">
11     | L'email doit avoir le bon format.
12   </div>
13   <div>
14     <input
15       type="password"
16       placeholder="Mot de passe"
17       formControlName="password"
18     />
19   </div>
20   <div *ngIf="form.get('password')?.errors">
21     | Le mot de passe doit faire au moins 6 caractères.
22   </div>
23   <div>
24     <input type="text" placeholder="Prénom" formControlName="firstName" />
25     <input type="text" placeholder="Nom" formControlName="lastName" />
26     <input type="number" placeholder="Âge" formControlName="age" />
27   </div>
28   <div>
29     | <button [disabled]="form.invalid">Submit</button>
30   </div>
31 </form>
```

Travaux pratiques

Travaux pratiques

Projet Fil Rouge Partie 3

Rôle de RxJS

La ***programmation réactive*** est un paradigme de programmation asynchrone qui s'intéresse aux flux de données et à la propagation des changements.

RxJS (Reactive Extensions for JavaScript) est une bibliothèque de programmation réactive utilisant des observables qui facilite la composition de code asynchrone ou basé sur des callbacks.

RxJS fournit :

- une implémentation du ***type Observable***,
- des ***fonctions utilitaires*** pour créer et travailler avec des observables :
 - convertir le code existant pour les opérations asynchrones en observables,
 - l'itération dans les valeurs d'un flux,
 - mapper des valeurs vers différents types,
 - filtrer les flux,
 - composer plusieurs flux.

Rôle de RxJS

Présentation des flux de données asynchrones

Présentation des flux de données asynchrones

La programmation asynchrone est une technique qui permet à un programme de lancer une tâche potentiellement longue tout en étant capable de répondre à d'autres événements pendant l'exécution de cette tâche, plutôt que d'attendre que celle-ci soit terminée.

Une fois la tâche terminée, le programme reçoit le résultat.

Présentation des flux de données asynchrones

Démo :

Affichage d'un texte dans le navigateur
en lançant d'abord une tâche asynchrone de 2 secondes,
puis une tâche synchrone.

Présentation des flux de données asynchrones

```
src > app > A app.component.ts > ...
1 import { Component, OnInit } from '@angular/core';
2
3 @Component({
4   selector: 'app-root',
5   templateUrl: './app.component.html',
6   styleUrls: ['./app.component.scss'],
7 })
8 export class AppComponent implements OnInit {
9   text = '';
10
11   ngOnInit(): void {
12     // Lancement d'une tâche asynchrone qui prend 2 secondes
13     setTimeout(() => {
14       this.text = 'Asynchrone';
15     }, 2000);
16
17     // Lancement d'une tâche synchrone qui n'attend pas le résultat de la tâche asynchrone
18     this.text = 'Synchrone';
19   }
20 }
```

```
src > app > S app.component.html > ...
Go to component
1 <p>{{ text }}</p>
```

Présentation des flux de données asynchrones

```
src > app > app.component.html > ...
          Go to component
1   <p>{{ text }}</p>
```

Rôle de RxJS

Propagation des changements avec RxJS

Propagation des changements avec RxJS

Les ***Observables*** sont des fonctions qui émettent des valeurs par flux.

Contrairement aux Promises,
les observables ne sont pas encore inhérents à JavaScript.

C'est pourquoi Angular s'appuie sur la bibliothèque ***RxJS*** pour les mettre en œuvre.

Propagation des changements avec RxJS

Un objet appelé ***Observer***
s'abonne aux valeurs émises par un Observable
grâce à la méthode « `subscribe()` » de ce dernier

- C'est un ensemble de 3 callbacks :
- ***next*** : écoute toutes les valeurs émises,
 - ***error*** : écoute les erreurs jetées,
 - ***complete*** : écoute la dernière valeur si l'Observable est complété.

Propagation des changements avec RxJS

Attention :

Quand on s'abonne à un Observable dans un composant ou une directive, il est nécessaire de ***se désabonner*** à la fin du cycle de vie de ce dernier pour éviter les fuites mémoires.

Propagation des changements avec RxJS

Opérateur RxJS

C'est une ***fonction qui prend un Observable en entrée et renvoie un autre Observable.***

La ***méthode « pipe()*** d'un observable permet de chaîner plusieurs opérateurs RxJS.

Propagation des changements avec RxJS

L'opérateur « `map()` » est probablement le plus utilisé.

Il applique une projection sur chaque valeur de la source émise pour la transformer et en émettre une nouvelle.

Propagation des changements avec RxJS

Démo :

Affichage de l'heure formatée dans un composant "clock"
écoutant un Observable
qui émet une nouvelle date toutes les secondes depuis un service "time".

Le composant est détruit et recréé grâce à un bouton dans son parent "app" et doit se désabonner à chaque destruction pour éviter les fuites mémoires.

Propagation des changements avec RxJS

```
src > app > A time.service.ts > ...
1  import { Injectable } from '@angular/core';
2  import { Subject } from 'rxjs';
3
4  @Injectable({
5    providedIn: 'root',
6  })
7  export class TimeService {
8    time$ = new Subject<Date>();
9
10   constructor() {
11     setInterval(() => {
12       this.time$.next(new Date());
13     }, 1000);
14   }
15 }
```

Propagation des changements avec RxJS

```
src > app > A app.component.ts > ...
1 import { Component, OnDestroy, OnInit } from '@angular/core';
2
3 @Component({
4   selector: 'app-root',
5   templateUrl: './app.component.html',
6   styleUrls: ['./app.component.scss'],
7 })
8 export class AppComponent {
9   show = true;
10 }
```

Propagation des changements avec RxJS

```
src > app > app.component.html > ...
  Go to component
1  <button (click)="show = !show">Show/Hide</button>
2  <app-clock *ngIf="show"></app-clock>
```

Propagation des changements avec RxJS

```
src > app > clock > clock.component.ts > ...
1  import { Component, OnDestroy, OnInit } from '@angular/core';
2  import { map, Subscription } from 'rxjs';
3  import { TimeService } from '../time.service';
4
5  @Component({
6    selector: 'app-clock',
7    templateUrl: './clock.component.html',
8    styleUrls: ['./clock.component.scss'],
9  })
10 export class ClockComponent implements OnInit, OnDestroy {
11   displayedTime: string = '';
12   private subscription!: Subscription;
13
14   constructor(private timeService: TimeService) {}
15
16   ngOnInit(): void {
17     this.subscription = this.timeService.time$.
18       pipe(
19         map((date: Date) => {
20           return `${date.getHours()} heures, ${date.getMinutes()} minutes et ${date.getSeconds()} secondes`;
21         })
22       )
23       .subscribe((result: string) => {
24         console.log('Abonnement en cours');
25         this.displayedTime = result;
26       });
27   }
28
29   ngOnDestroy(): void {
30     this.subscription.unsubscribe();
31   }
32 }
```

Propagation des changements avec RxJS

```
src > app > clock > clock.component.html > ...
      Go to component
1   <h1>{{displayedTime}}</h1>
-
```

Travail avec HTTP

Travail avec HTTP

Le service HTTP

Le service HTTP

La plupart des applications front-end doivent ***communiquer avec un serveur par le biais du protocole HTTP***, afin de télécharger ou d'envoyer des données et d'accéder à d'autres services back-end.

Angular fournit une API client HTTP pour les applications Angular : la ***classe de service HttpClient*** dans le paquet « @angular/common/http ».

Pour rendre ce service disponible, le ***HttpClientModule*** doit être importé de ce même paquet.

Travail avec HTTP

Communication avec une API en asynchrone

Communication avec une API en asynchrone

Le service ***HttpClient s'injecte en dépendance*** d'une classe.

Il permet de communiquer avec une API,
via des ***méthodes HTTP RESTful***, notamment :

- post()
- get()
- put()
- patch()
- delete()

Ces méthodes ***renvoient un Observable***.

Communication avec une API en asynchrone

Démo :

Formulaire de création de film avec des données persistantes.

Récupération et formatage des données.

Communication avec une API en asynchrone

```
src > app > A app.module.ts > ...
1 | import { HttpClientModule } from '@angular/common/http';
2 | import { NgModule } from '@angular/core';
3 | import { BrowserModule } from '@angular/platform-browser';
4 |
5 | import { AppComponent } from './app.component';
6 |
7 | @NgModule({
8 |   declarations: [AppComponent],
9 |   imports: [BrowserModule, HttpClientModule],
10 |   providers: [],
11 |   bootstrap: [AppComponent],
12 | })
13 | export class AppModule {}
```

Communication avec une API en asynchrone

```
src > app > ts movie.ts > ...
1   export interface Movie {
2     id: string;
3     title: string;
4 }
```

Communication avec une API en asynchrone

```
src > app > A movie.service.ts > M MovieService > getMovies > map() callback
  1 import { HttpClient } from '@angular/common/http';
  2 import { Injectable } from '@angular/core';
  3 import { map, Observable } from 'rxjs';
  4
  5 import { Movie } from './movie';
  6
  7 @Injectable({
  8   providedIn: 'root',
  9 })
10 export class MovieService {
11   private movieUrl =
12     'https://formation-6e588-default-rtdb.firebaseio.europe-west1.firebaseio.app/movies.json';
13   movies: Movie[] = [];
14
15   constructor(private http: HttpClient) {}
16
17   postMovie(movieTitle: string): Observable<any> {
18     return this.http.post(this.movieUrl, { title: movieTitle }).pipe(
19       map((res) => {
20         this.getMovies().subscribe();
21         return res;
22       })
23     );
24   }
25 }
```

Communication avec une API en asynchrone

```
25
26     getMovies(): Observable<Movie[]> {
27         return this.http.get(this.movieUrl).pipe(
28             map((res: any) => {
29                 const movies = [];
30                 for (const key in res) {
31                     const movie = {
32                         id: key,
33                         title: res[key].title,
34                     };
35                     movies.push(movie);
36                 }
37                 this.movies = movies;
38                 return movies;
39             })
40         );
41     }
42 }
```

Communication avec une API en asynchrone

```
src > app > A app.component.ts > ...
1  import { Component, OnInit } from '@angular/core';
2
3  import { MovieService } from './movie.service';
4
5  @Component({
6    selector: 'app-root',
7    templateUrl: './app.component.html',
8    styleUrls: ['./app.component.scss'],
9  })
10 export class AppComponent implements OnInit {
11   constructor(private movieService: MovieService) {}
12
13   get movies() {
14     return this.movieService.movies;
15   }
16
17   ngOnInit(): void {
18     this.movieService.getMovies().subscribe();
19   }
20
21   onAddMovie(movieTitle: string) {
22     this.movieService.postMovie(movieTitle).subscribe();
23   }
24 }
```

Communication avec une API en asynchrone

```
src > app > app.component.html > ...
    Go to component
1   <input type="text" #titleInput />
2   <button (click)="onAddMovie(titleInput.value)">Add Movie</button>
3
4   <ul>
5     <li *ngFor="let movie of movies">{{ movie.title }}</li>
6   </ul>
```

Travaux pratiques

Projet Fil Rouge Partie 4

Le routage

Le routage

Bases du routage :
configuration des routes et navigation

Bases du routage : configuration des routes et navigation

Le **RouterModule** est importé grâce au paquet « @angular/router ».

Il permet d'utiliser le service **Router** pour la navigation entre les vues de l'application.

Pour ce faire,
on enregistre les routes lors de l'importation de ce module.

Bases du routage : configuration des routes et navigation

La directive ***RouterOutlet*** agit comme un espace réservé qu'Angular remplit dynamiquement en fonction de l'état actuel du routeur.

Elle s'utilise en plaçant des balises ***<router-outlet></router-outlet>*** dans le template d'un composant.

Bases du routage : configuration des routes et navigation

On peut naviguer vers une route de 2 façons :

- Par le ***template*** : en utilisant la directive RouterLink
 - Par le ***modèle*** : en injectant le service Router

Bases du routage : configuration des routes et navigation

Démo :

Mise en place d'une navigation entre un index et une page de profil via une navbar.
L'une se fait par le template, l'autre par le modèle.

Une mauvaise route ou une route vide renvoient vers la page d'index.

Bases du routage : configuration des routes et navigation

```
src > app > A app-routing.module.ts > ...
1  import { NgModule } from '@angular/core';
2  import { RouterModule, Routes } from '@angular/router';
3
4  import { IndexComponent } from './index/index.component';
5  import { ProfileComponent } from './profile/profile.component';
6
7  const routes: Routes = [
8    { path: 'index', component: IndexComponent },
9    { path: 'profile', component: ProfileComponent },
10   { path: '', redirectTo: 'index', pathMatch: 'full' },
11   { path: '**', redirectTo: 'index' },
12 ];
13
14 @NgModule({ imports: [RouterModule.forRoot(routes)], exports: [RouterModule] })
15 export class AppRoutingModule {}
```

Bases du routage : configuration des routes et navigation

```
src > app > 🏛 app.module.ts > ...
  1 import { NgModule } from '@angular/core';
  2 import { BrowserModule } from '@angular/platform-browser';
  3
  4 import { AppComponent } from './app.component';
  5 import { NavbarComponent } from './navbar/navbar.component';
  6 import { IndexComponent } from './index/index.component';
  7 import { ProfileComponent } from './profile/profile.component';
  8 import { AppRoutingModule } from './app-routing.module';
  9
 10 @NgModule({
 11   declarations: [
 12     AppComponent,
 13     NavbarComponent,
 14     IndexComponent,
 15     ProfileComponent
 16   ],
 17   imports: [
 18     BrowserModule,
 19     AppRoutingModule
 20   ],
 21   providers: [],
 22   bootstrap: [AppComponent]
 23 })
 24 export class AppModule { }
 25
```

Bases du routage : configuration des routes et navigation

```
src > app > app.component.html > ...
    Go to component
1   <app-navbar></app-navbar>
2   <router-outlet></router-outlet>
```

Bases du routage : configuration des routes et navigation

```
src > app > navbar > A navbar.component.ts > ...
1 import { Component, OnInit } from '@angular/core';
2 import { Router } from '@angular/router';
3
4 @Component({
5   selector: 'app-navbar',
6   templateUrl: './navbar.component.html',
7   styleUrls: ['./navbar.component.scss'],
8 })
9 export class NavbarComponent implements OnInit {
10   constructor(private router: Router) {}
11
12   ngOnInit(): void {}
13
14   onClickProfile() {
15     this.router.navigate(['profile']);
16   }
17 }
```

Bases du routage : configuration des routes et navigation

```
src > app > navbar > A navbar.component.ts > ...
1 import { Component, OnInit } from '@angular/core';
2 import { Router } from '@angular/router';
3
4 @Component({
5   selector: 'app-navbar',
6   templateUrl: './navbar.component.html',
7   styleUrls: ['./navbar.component.scss'],
8 })
9 export class NavbarComponent implements OnInit {
10   constructor(private router: Router) {}
11
12   ngOnInit(): void {}
13
14   onClickProfile() {
15     this.router.navigate(['profile']);
16   }
17 }
```

Le routage

Arborescence des routeurs

Parenté entre les routes

Les composants affichés grâce à une route peuvent eux-mêmes contenir des balises “router-outlet”.

Les routes correspondantes sont donc des enfants de cette route mère.

Elles s'enregistrent ainsi dans le ***propriété “children”*** de cette route.

Parenté entre les routes

Démo :

Mise en place d'une route "product" avec une sous-route "detail"
après avoir généré les composants correspondants
et un "product-list" enfant de "product".

Parenté entre les routes

```
src > app > A app-routing.module.ts > ...
1   import { NgModule } from '@angular/core';
2   import { RouterModule, Routes } from '@angular/router';
3
4   import { IndexComponent } from './index/index.component';
5   import { ProductDetailComponent } from './product/product-detail/product-detail.component';
6   import { ProductComponent } from './product/product.component';
7   import { ProfileComponent } from './profile/profile.component';
8
9   const routes: Routes = [
10     { path: 'index', component: IndexComponent },
11     { path: 'profile', component: ProfileComponent },
12     {
13       path: 'product',
14       component: ProductComponent,
15       children: [
16         {
17           path: 'detail',
18           component: ProductDetailComponent,
19         },
20         { path: '**', redirectTo: '' },
21       ],
22     },
23     { path: '', redirectTo: 'index', pathMatch: 'full' },
24     { path: '**', redirectTo: 'index' },
25   ];
26
27   @NgModule({ imports: [RouterModule.forRoot(routes)], exports: [RouterModule] })
28   export class AppRoutingModule {}
```

Parenté entre les routes

```
src > app > navbar > navbar.component.html > a
      Go to component
1   <a routerLink="index">Index</a>
2   <a (click)="onClickProfile()">Profile</a>
3   <a routerLink="product">Product</a>
```

Parenté entre les routes

```
src > app > product >  product.component.html > ...
  Go to component
1  <app-product-list></app-product-list>
2  <router-outlet></router-outlet>
```

Parenté entre les routes

```
src > app > product > product-list > product-list.component.html > ...
      Go to component
1   <div>
2   |   <a routerLink="detail">Open detail</a>
3   </div>
```

Le routage

Récupération des paramètres d'une route

Récupération des paramètres d'une route

Lorsqu'une route est enregistrée,
son chemin peut contenir des **paramètres**.

Il s'agit de variables,
qui sont identifiées avec la syntaxe « **:nom-du-paramètre** ».

On peut ainsi naviguer vers cette route
en remplaçant le nom du paramètre par la valeur souhaitée.

Récupération des paramètres d'une route

La classe ***ActivatedRoute***
peut être injectée dans un objet
pour ***accéder aux paramètres*** de la route active.

Récupération des paramètres d'une route

Démo :

Création d'une interface et d'un service “product” contenant une liste de 3 objets (id et nom).

Connexion des produits du service à la liste.

Navigation vers la route “detail” avec un paramètre “id”.

Récupération de l'id dans la route active depuis le composant “product-detail” et appel au service pour afficher el bon produit.

Récupération des paramètres d'une route

```
src > app > ts product.ts > ...
1  export interface Product {
2    id: number;
3    name: string;
4 }
```

Récupération des paramètres d'une route

```
src > app > A product.service.ts > ...
1  import { Injectable } from '@angular/core';
2  import { Product } from './product';
3
4  @Injectable({
5    providedIn: 'root',
6  })
7  export class ProductService {
8    products: Product[] = [
9      { id: 1, name: 'Burger' },
10     { id: 2, name: 'Salad' },
11     { id: 3, name: 'Pizza' },
12   ];
13
14   findProductById(productId: number) {
15     return this.products.find((product) => product.id === productId);
16   }
17 }
```

Récupération des paramètres d'une route

```
src > app > product > product-list > product-list.component.html > ...
Go to component
1  <div>
2  |  <ul>
3  |  |  <li *ngFor="let product of products" (click)="onOpenDetail(product.id)">{{product.name}}</li>
4  |  </ul>
5  </div>
```

Récupération des paramètres d'une route

```
src > app > product > product-list > A product-list.component.ts > ...
  1 import { Component, OnInit } from '@angular/core';
  2 import { ActivatedRoute, Router } from '@angular/router';
  3
  4 import { Product } from 'src/app/product';
  5 import { ProductService } from 'src/app/product.service';
  6
  7 @Component({
  8   selector: 'app-product-list',
  9   templateUrl: './product-list.component.html',
10   styleUrls: ['./product-list.component.scss'],
11 })
12 export class ProductListComponent {
13   constructor(
14     private productService: ProductService,
15     private router: Router,
16     private route: ActivatedRoute
17   ) {}
18
19   get products(): Product[] {
20     return this.productService.products;
21   }
22
23   onOpenDetail(productId: number) {
24     this.router.navigate([productId], { relativeTo: this.route });
25   }
26 }
```

Récupération des paramètres d'une route

```
src > app > A app-routing.module.ts > ...
1  import { NgModule } from '@angular/core';
2  import { RouterModule, Routes } from '@angular/router';
3
4  import { IndexComponent } from './index/index.component';
5  import { ProductDetailComponent } from './product/product-detail/product-detail.component';
6  import { ProductComponent } from './product/product.component';
7  import { ProfileComponent } from './profile/profile.component';
8
9  const routes: Routes = [
10    { path: 'index', component: IndexComponent },
11    { path: 'profile', component: ProfileComponent },
12    {
13      path: 'product',
14      component: ProductComponent,
15      children: [
16        {
17          path: ':id',
18          component: ProductDetailComponent,
19        },
20        { path: '**', redirectTo: '' },
21      ],
22    },
23    { path: '', redirectTo: 'index', pathMatch: 'full' },
24    { path: '**', redirectTo: 'index' },
25  ];
26
27  @NgModule({ imports: [RouterModule.forRoot(routes)], exports: [RouterModule] })
28  export class AppRoutingModule {}
```

Récupération des paramètres d'une route

```
src > app > product > product-detail > product-detail.component.ts > ...
  1 import { Component } from '@angular/core';
  2 import { ActivatedRoute } from '@angular/router';
  3 import { Product } from 'src/app/product';
  4 import { ProductService } from 'src/app/product.service';
  5
  6 @Component({
  7   selector: 'app-product-detail',
  8   templateUrl: './product-detail.component.html',
  9   styleUrls: ['./product-detail.component.scss'],
10 })
11 export class ProductDetailComponent {
12   constructor(
13     private route: ActivatedRoute,
14     private productService: ProductService
15   ) {}
16
17   get product(): Product | undefined {
18     const id = +this.route.snapshot.params['id'];
19     return this.productService.findProductById(id);
20   }
21 }
```

Récupération des paramètres d'une route

```
src > app > product > product-detail > product-detail.component.html > ...
    Go to component
1   <h1>{{product?.name}}</h1>
2
```

Le routage

Mise en place des guards

Mise en place des guards

Des « *guards* » peuvent être ajoutés lors de la configuration du routage pour contrôler l'accès à certaines routes.

Mise en place des guards

ng generate guard ...

Mise en place des guards

Démo :

Simulation de connexion avec un service “auth” et un bouton vert ou rouge dans la navbar selon le statut.

Création d'un guard "auth" implémentant l'interface "CanActivate"
et injectant le service "auth"
pour empêcher l'accès à la route "product" au visiteur déconnecté.

Mise en place des guards

```
src > app > A auth.service.ts > ...
1   import { Injectable } from '@angular/core';
2
3   @Injectable({
4     providedIn: 'root',
5   })
6   export class AuthService {
7     isAuthenticated = false;
8
9     toggleAuth() {
10       this.isAuthenticated = !this.isAuthenticated;
11     }
12 }
```

Mise en place des guards

```
src > app > navbar > A navbar.component.ts > ...
1  import { Component, OnInit } from '@angular/core';
2  import { Router } from '@angular/router';
3
4  import { AuthService } from '../auth.service';
5
6  @Component({
7    selector: 'app-navbar',
8    templateUrl: './navbar.component.html',
9    styleUrls: ['./navbar.component.scss'],
10   })
11 export class NavbarComponent {
12   constructor(private router: Router, private authService: AuthService) {}
13
14   get isAuthenticated(): boolean {
15     return this.authService.isAuthenticated;
16   }
17
18   onClickProfile() {
19     this.router.navigate(['profile']);
20   }
21
22   onToggleAuth() {
23     this.authService.toggleAuth();
24   }
25 }
```

Mise en place des guards

```
src > app > navbar > navbar.component.html > ...
    Go to component
1   <button [ngClass]="{ green: isAuthenticated, red: !isAuthenticated }" (click)="onToggleAuth()">
2     | Status
3   </button>
4   <a routerLink="index">Index</a>
5   <a (click)="onClickProfile()">Profile</a>
6   <a routerLink="product">Product</a>
```

Mise en place des guards

```
src > app > navbar > navbar.component.scss > ...
1   .green {
2     |   background-color: green;
3   }
4
5   .red {
6     |   background-color: red;
7 }
```

Mise en place des guards

```
src > app > A auth.guard.ts > ...
1  import { Injectable } from '@angular/core';
2  import { ActivatedRouteSnapshot, CanActivate, RouterStateSnapshot, UrlTree } from '@angular/router';
3  import { Observable } from 'rxjs';
4
5  import { AuthService } from './auth.service';
6
7  @Injectable({
8    providedIn: 'root'
9  })
10 export class AuthGuard implements CanActivate {
11   constructor(private authService: AuthService) {}
12
13   canActivate(
14     route: ActivatedRouteSnapshot,
15     state: RouterStateSnapshot): Observable<boolean | UrlTree> | Promise<boolean | UrlTree> | boolean | UrlTree {
16     return this.authService.isAuthenticated();
17   }
18 }
```

Mise en place des guards

```
src > app > 🚫 app-routing.module.ts > ...
1  import { NgModule } from '@angular/core';
2  import { RouterModule, Routes } from '@angular/router';
3
4  import { IndexComponent } from './index/index.component';
5  import { ProductDetailComponent } from './product/product-detail/product-detail.component';
6  import { ProductComponent } from './product/product.component';
7  import { ProfileComponent } from './profile/profile.component';
8  import { AuthGuard } from './auth.guard';
9
10 const routes: Routes = [
11   { path: 'index', component: IndexComponent },
12   { path: 'profile', component: ProfileComponent },
13   {
14     path: 'product',
15     component: ProductComponent,
16     canActivate: [AuthGuard],
17     children: [
18       {
19         path: ':id',
20         component: ProductDetailComponent,
21       },
22       { path: '**', redirectTo: '' },
23     ],
24   },
25   { path: '', redirectTo: 'index', pathMatch: 'full' },
26   { path: '**', redirectTo: 'index' },
27 ];
28
29 @NgModule({ imports: [RouterModule.forRoot(routes)], exports: [RouterModule] })
30 export class AppRoutingModule {}
```

Travaux pratiques

Projet Fil Rouge Partie 5

Merci pour votre écoute !