[Angular](https://angular.io/" \t "_blank) est un **framework JavaScript** qui vous permet de développer des applications "efficaces et sophistiquées", comme dit la documentation. Il permet notamment de créer ce qu'on appelle des **Single Page Applications** (ou **SPA**) : des applications entières qui tournent dans une seule page HTML grâce au JavaScript

**Faites connaissance du TypeScript**

Le TypeScript (TS) ! vient ajouter des syntaxes supplémentaires au JavaScript ; notamment, comme son nom l'indique, le **typage strict**. Le typage strict nous oblige, entre autres, à spécifier le **type** de toutes les variables, contrairement au typage dynamique de JavaScript. Cette contrainte peut paraître gênante, mais elle permet de réduire considérablement le nombre d'erreurs au runtime, et facilite énormément le développement.

TypeScript has a range of options for controlling how strict the type checking

Voici une liste non exhaustive de raisons de choisir Angular :

* Angular est un framework **complet** – on peut créer des applications web complètes sans avoir besoin de librairies tierces supplémentaires. C'est notamment ce qui différencie un *framework* d'une *library*.
* Les **best practices** (les bonnes pratiques) Angular sont extrêmement précises – les structures des applications Angular ont tendance à se ressembler fortement, donc il est facile de passer d'un projet à un autre, et de s'assurer que son propre projet suit les meilleurs pratiques.
* Le framework a été conçu pour fonctionner avec le **TypeScript** – il est tout à fait possible d'utiliser le TypeScript pour React, Vue ou Svelte, mais Angular a été conçu pour ce langage, donc son intégration est forcément plus profonde.

## Installez les dépendances

* Pour installer le CLI d'Angular, il vous faut une version LTS de npm (node package manager).
* On installe le CLI avec la commande  npm i -g @angular/cli
* On peut vérifier la version installée du CLI avec  ng v

## Construisez une application Angular avec le CLI

Depuis une ligne de commande, naviguez dans le dossier où vous souhaitez créer votre projet Angular, et exécutez la commande suivante :

ng new snapface --style=scss --skip-tests=true

La commande  ng new  génère une nouvelle application Angular, installe les dépendances du projet, et initialise un dépôt Git. Le flag  --style=scss  dit au CLI que vous souhaitez utiliser le SCSS pour les styles (plutôt que le CSS ou le LESS, par exemple), et le flag  --skip-tests=true  dit que vous ne voulez pas générer de fichiers de tests unitaires pour ce projet. Oui, c'est mal de ne pas écrire de tests, mais chaque chose en son temps !

Un **flag** dans ce contexte est une option de configuration passée à une commande en ligne de commande. Ces flags sont souvent précédés de  -  ou de  --

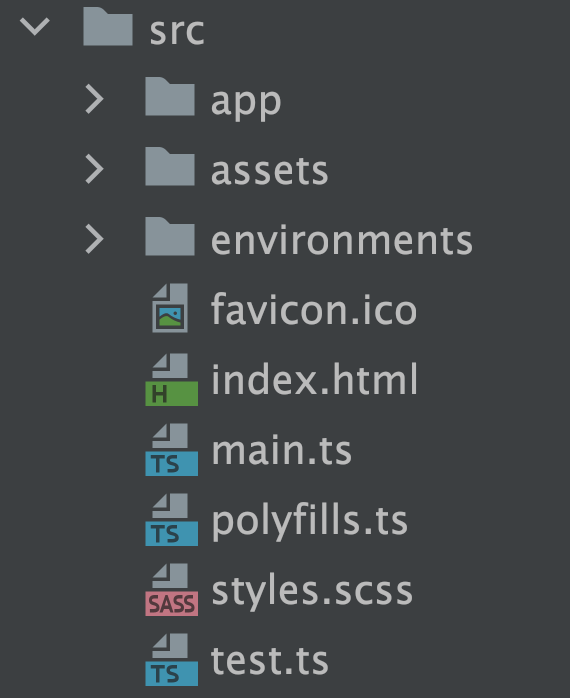
### Lancez le serveur de développement

Ouvrez une ligne de commande dans le dossier du projet (dans mon cas, le dossier Snapface), et exécutez :

ng serve

### Explorez le dossier du projet

Une application Angular contient beaucoup de fichiers (notamment de configuration) qui ne nous intéresseront pas dans le cadre de ce cours. Vous allez majoritairement travailler dans le dossier  src  :

Le dossier src

Le premier fichier qui nous saute aux yeux est  index.html . Ici, comme sur un site classique,  index.html  est le fichier principal de votre projet. Mais si vous regardez dedans …

<!doctype html>

<html lang="en">

<head>

<meta charset="utf-8">

<title>Snapface</title>

<base href="/">

<meta name="viewport" content="width=device-width, initial-scale=1">

<link rel="icon" type="image/x-icon" href="favicon.ico">

</head>

<body>

<app-root></app-root>

</body>

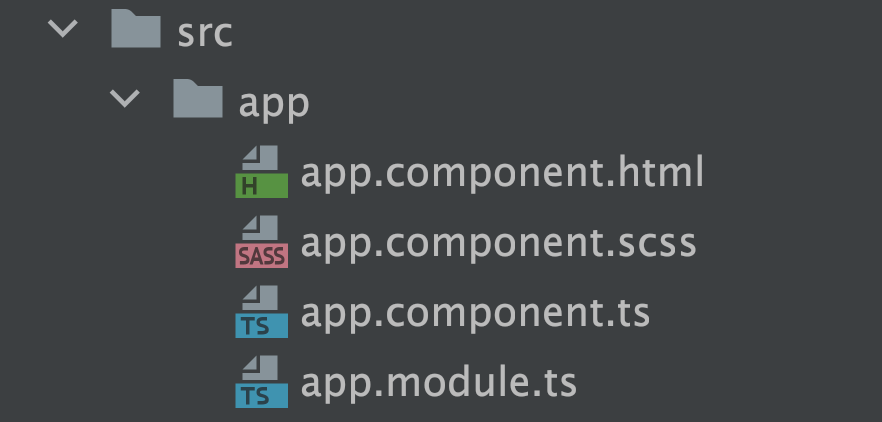
</html>

… le  <body>  ne contient qu'une balise  <app-root>  vide. D'où vient tout ce que l'on voit dans le navigateur, dans ce cas ??

Eh bien comme son nom l'indique, la balise  <app-root>  correspond à la racine de votre application, ou plus précisément à l'AppComponent, le **component** racine de votre application. Angular sait qu'il doit remplacer cette balise par le fameux AppComponent, donc allons voir où il se trouve !

Vous ne savez pas encore ce qu'est un *component*, mais c'est le sujet du prochain chapitre, donc patience !

Dans le sous-dossier  app  :

Le sous-dossier app

À part le fichier app.module.ts  auquel on s'intéressera plus tard, vous avez trois fichiers correspondant à AppComponent : un fichier HTML, un fichier SCSS, et un fichier TS.

Les fichiers avec comme extension .ts sont des fichiers TypeScript.

Si vous regardez dans le fichier app.component.html, vous y trouverez le contenu que vous voyez dans votre navigateur ! Mais comment Angular sait-il qu'il faut remplacer  <app-root>  par ce contenu ? Le secret se trouve dans  app.component.ts  :

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

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

styleUrls: ['./app.component.scss']

})

export class AppComponent {

title = 'snapface';

}

Vous verrez dans la prochaine partie du cours comment fonctionne ce fichier plus précisément, mais vous devriez déjà voir quelques éléments intéressants !

Petit teaser du prochain chapitre : changez le contenu de  title  , enregistrez le fichier, et regardez ce qui change dans votre navigateur…

## Construisez votre premier component

### Qu'est-ce qu'un component ?

Une application Angular peut être vue comme une **arborescence** de components avec AppComponent comme component racine.

Il n'y a pas de règle d'or dans le découpage d'une application en components : on essaie **d'équilibrer** entre ne pas avoir un trop grand nombre de components, et avoir quand même des components simples et pas trop gros.

### Créez un component

En pratique, un component va associer un fichier HTML, un fichier SCSS et un fichier TypeScript : c'est le cas de l'AppComponent généré par le CLI, par exemple.

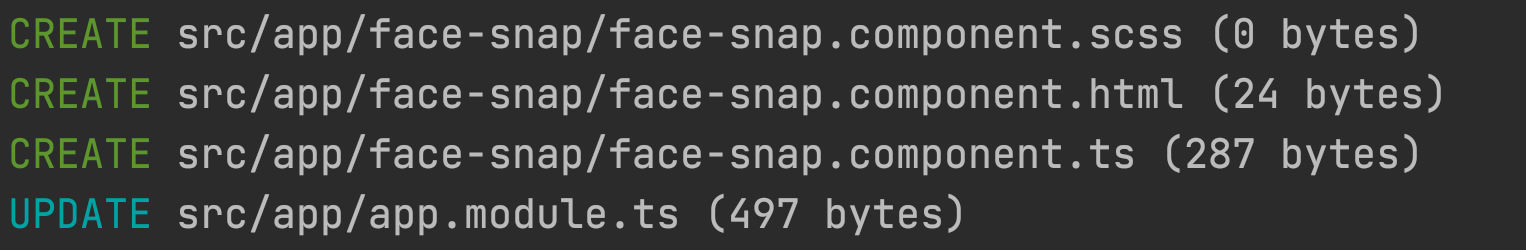
Vous avez peut-être remarqué que la variable  title  n'est pas précédée de  var,  let,  const… Si vous vous demandez pourquoi, c'est parce qu'il s'agit d'un **attribut** de la classe AppComponent. Lorsqu'on déclare un attribut de classe, on n'a pas besoin de mot-clé : on le déclare simplement par son nom.

Le CLI d'Angular propose des commandes de création pour faciliter le développement, et vous allez en utiliser une maintenant, depuis une ligne de commande dans le dossier du projet :

ng generate component face-snap

Notez bien qu'on n'a pas besoin d'ajouter "Component" au nom : c'est fait automatiquement par le CLI.

Si vous regardez bien le log du CLI, vous verrez exactement ce qu'il a fait :



Il a **créé**, dans un sous-dossier  face-snap  , les trois fameux fichiers du component, et il a **mis à jour**  app.module.ts  . Allons voir ce qu'il a changé dans AppModule :

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

import { BrowserModule } from '@angular/platform-browser';

import { AppComponent } from './app.component';

import { FaceSnapComponent } from './face-snap/face-snap.component';

@NgModule({

declarations: [

AppComponent,

FaceSnapComponent,

],

imports: [

BrowserModule

],

providers: [],

bootstrap: [AppComponent]

})

export class AppModule { }

Pour utiliser un component dans une application Angular, il faut le **déclarer** dans un **module** – AppModule étant le module principal de l'application, tout comme AppComponent est le component principal. Tous les components que vous créerez dans ce cours seront déclarés ici.

Si vous regardez maintenant dans le dossier snap, vous y trouverez les fichiers de FaceSnapComponent. Dans le fichier HTML (qu'on appelle le **template**), vous trouvez une balise  <p>  qui permet simplement de vérifier que le component fonctionne, et le fichier SCSS est vide. Regardons maintenant le fichier  face-snap.component.ts  :

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

@Component({

selector: 'app-face-snap',

templateUrl: './face-snap.component.html',

styleUrls: ['./face-snap.component.scss']

})

export class FaceSnapComponent implements OnInit {

constructor() { }

ngOnInit(): void {

}

}

On voit ici la construction du component. On voit une **classe** TypeScript qui s'appelle SnapComponent, avec un  constructor  et une méthode  ngOnInit.

Cette classe est déclarée avec un décorateur  @Component  à qui on passe un objet de configuration avec un **sélecteur**, un fichier de **template** et un fichier de **styles**.

Un **décorateur** en TypeScript permet, entre autres, d'apporter des **modifications** à une classe. Ici, le décorateur  @Component  vient ajouter tous les comportements nécessaires à l'utilisation de ce component dans l'application. Il est importé depuis le package  @angular/core. Tout se passe sous le capot, on n'a pas à s'en occuper !

Comme vous le savez, votre application est une arborescence de components avec AppComponent comme racine : c'est donc dans  app.component.html  qu'on va venir ajouter le **sélecteur** de notre nouveau component comme balise HTML :

<app-face-snap></app-face-snap>

### En résumé

* Une application Angular peut être vue comme une arborescence de components.
* Un component lie un template HTML et des styles SCSS à du comportement TS.
* Pour créer un component, on utilise la commande  ng generate component nom-du-component
* Le sélecteur d'un component correspond à la balise HTML personnalisée qu'on utilisera pour l'insérer dans l'application.

**Affichez des données dynamiques**

L'une des principales raisons d'être de toute application est de donner des informations à l'utilisateur, de lui **afficher des données dynamiques**. En pratique, ça signifie afficher ou utiliser des valeurs venant du TypeScript dans le template. À terme, ces données pourront venir d'une requête serveur, par exemple, ou d'une autre partie de l'application. Pour l'instant, vous allez travailler avec des données écrites en dur, et les passer au template.

En Angular, il y a deux façons principales d'exploiter une donnée venant du TypeScript dans le template : la **string interpolation** et la liaison par attribut, ou **attribute binding**. Mais pour afficher des données, il faut d'abord des données !

### Ajoutez des données

On va commencer par ajouter quelques **propriétés** à notre classe FaceSnapComponent, qui correspondront aux différentes propriétés d'un FaceSnap. Commençons par ajouter un titre, une description, une date de mise en ligne, et un nombre de "snaps" (on va dire qu'un utilisateur "snappe" une photo quand il la trouve chouette) :

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

@Component({

selector: 'app-face-snap',

templateUrl: './face-snap.component.html',

styleUrls: ['./face-snap.component.scss']

})

export class FaceSnapComponent {

*title*: *string*;

*description*: *string*;

*createdDate*: *Date*;

*snaps*: *number*;

}

Comme vous pouvez le constater, on crée un attribut de classe en associant le **nom** de l'attribut à son **type**.

Il y a des chances pour que votre IDE commence déjà à souligner en rouge le nom de vos attributs ici. En effet, si vous lisez l'erreur retournée, c'est parce que vous avez créé des propriétés sans les initialiser. Pour "promettre" à TypeScript qu'on va les initialiser, on peut ajouter un bang  !  à chaque propriété :

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

@Component({

selector: 'app-face-snap',

templateUrl: './face-snap.component.html',

styleUrls: ['./face-snap.component.scss']

})

export class FaceSnapComponent {

title!: *string*;

description!: *string*;

createdDate!: Date;

snaps!: *number*;

}

Pour initialiser ces propriétés en suivant les best practices Angular, vous allez utiliser la méthode  ngOnInit()  (que vous aviez supprimée précédemment). Pour l'utiliser, votre component doit implémenter l'interface  OnInit  :

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

@Component({

selector: 'app-face-snap',

templateUrl: './face-snap.component.html',

styleUrls: ['./face-snap.component.scss']

})

export class FaceSnapComponent implements OnInit {

title!: *string*;

description!: *string*;

createdDate!: Date;

snaps!: *number*;

ngOnInit() {

}

}

La méthode  ngOnInit()  est appelée automatiquement par Angular au moment de la **création de chaque instance** du component. Elle permet notamment d'initialiser des propriétés.

On va maintenant initialiser les quatre propriétés dans la méthode  ngOnInit()  :

ngOnInit() {

*this.*title = 'Archibald';

*this.*description = 'Mon meilleur ami depuis tout petit !';

*this.*createdDate = new Date();

*this.*snaps = 6;

}

### Affichez du contenu avec la string interpolation

La **string interpolation** est la méthode la plus simple pour afficher le contenu d'une variable dans le template. Il suffit de mettre le nom de la variable à afficher entre **doubles accolades** {{ }} dans le HTML . Faisons-le maintenant avec les données de notre FaceSnap. Dans  face-snap.component.html  :

<h2>{{ title }}</h2>

<p>Mis en ligne le {{ createdDate }}</p>

<p>{{ description }}</p>

<p>🤌 {{ snaps }}</p>

Mais qu'est-ce qui se passe si on veut attribuer une valeur dynamique à un **attribut** d'un élément HTML, comme la  src  d'une image ?

Allons voir ça tout de suite !

### Dynamisez des données avec l'attribute binding

Ajoutez d'abord une propriété  imageUrl  au component, et initialisez-la avec une URL d'image valable :

imageUrl!: *string*;

// ...

*this.*imageUrl = 'https://cdn.pixabay.com/photo/2015/05/31/16/03/teddy-bear-792273\_1280.jpg';

Pour lier notre URL, on pourrait imaginer quelque chose comme ceci :

<img src="{{ imageUrl }}" alt="{{ title }}">

… et *techniquement*, dans ce cas précis, cette syntaxe fonctionnerait, car les valeurs de  imageUrl  et de  title  sont des  string  ! Cependant, ce n'est pas la syntaxe préférée, car les valeurs associées à des attributs ne seront pas toujours des strings, et il est préférable d'avoir une syntaxe unique pour une fonctionnalité. On va donc préférer les **crochets**  []  :

<img [src]="imageUrl" [alt]="title">

### En résumé

* On déclare les propriétés d'un component en haut de sa classe, et on les initialise dans la méthode  ngOnInit()
* La méthode  ngOnInit()  est appelée une fois par instance de component au moment de la création de cette instance.
* La string interpolation avec les doubles accolades  {{ }}  permet d'insérer la valeur d'une propriété TypeScript dans le template.
* L'attribute binding permet de lier la valeur d'une propriété TypeScript à un attribut HTML, en mettant l'attribut entre crochets  []  et en passant le nom de la propriété.

## Réagissez aux événements

**Écoutez le template**

Pour qu'une application soit dynamique, il faut que l'utilisateur puisse interagir avec elle. Vous allez maintenant réagir à un événement venant du template avec du comportement TypeScript, en ajoutant un bouton "Oh Snap !".

<p>

<button>Oh Snap!</button>

🤌 {{ snaps }}

</p>

onAddSnap() {

*this.*snaps++;

}

Le nom de méthode qui commence par **on** signale que cette méthode répond à un événement.

Il faut maintenant **lier** cette méthode au clic sur le bouton avec la liaison par événement, ou **event binding**. Là où vous avez utilisé des crochets pour lier aux attributs, vous allez utiliser des **parenthèses**  ()  pour lier aux événements :

<button (click)="onAddSnap()">Oh Snap!</button>

Prenons quelques instants pour noter que chaque **instance** d'un component est totalement **indépendante**. Pour le vérifier, ajoutez quelques instances supplémentaires de FaceSnapComponent, et cliquez sur leurs boutons "Oh Snap!". Vous verrez que seul le nombre de snaps pour le component où vous cliquez augmente.

### En résumé

* Pour lier une méthode à un événement d'un élément du template, mettez l'événement entre **parenthèses**  ()  et passez la méthode en argument ; ex. : (click)="onClickButton()"

## Ajoutez des propriétés personnalisées

### Créez une classe FaceSnap

Angular vous permet d'ajouter des **propriétés personnalisées** à vos components de manière à ce que vous puissiez les injecter **depuis le component parent**. Ce que ça vous permettrait de faire avec  FaceSnapComponent, ce serait de faire en sorte que  AppComponent  centralise les données de plusieurs FaceSnaps, génère une instance du component pour chaque FaceSnap, et injecte ce FaceSnap à cette instance.

Mais avant de voir comment créer ce type de propriété, profitons de ce moment pour créer votre premier **type personnalisé**sous forme de classe. On n'a pas envie de devoir injecter un titre, une description, une URL, etc., manuellement à chaque component ! Ce qui serait plus utile, ce serait un **modèle de données**  FaceSnap, qui comporterait toutes ces propriétés, et que vous pourriez injecter directement dans  FaceSnapComponent, et même utiliser partout dans votre application. Heureusement, TypeScript vous permet de faire exactement ça !

Dans votre dossier  app, créez un sous-dossier  models, et dedans, créez un fichier  face-snap.model.ts dans lequel vous allez créer une classe FaceSnap. Cette classe comportera toutes les propriétés nécessaires d'un FaceSnap. Ça pourrait être intéressant également d'implémenter un **constructor**qui faciliterait la création d'objets FaceSnap.

export class FaceSnap {

*title*: *string*;

*description*: *string*;

*createdDate*: *Date*;

*snaps*: *number*;

*imageUrl*: *string*;

constructor(*title*: *string*, *description*: *string*, *imageUrl*: *string*, *createdDate*: *Date*, *snaps*: *number*) {

*this.*title = title;

*this.*description = description;

*this.*imageUrl = imageUrl;

*this.*createdDate = createdDate;

*this.*snaps = snaps;

}

}

Cette classe comporte tout ce qu'il faut, et nous permet de générer des objets FaceSnap avec la syntaxe  new FaceSnap().

Je voudrais en profiter pour vous montrer un **raccourci TypeScript**. Si vous avez des propriétés qui seront initialisées par les arguments passés au constructor comme ci-dessus, vous pouvez retirer leurs déclarations et initialisations, et leur ajouter simplement le modificateur  public  dans le  constructor  :

export class FaceSnap {

constructor(public *title*: *string*,

public *description*: *string*,

public *imageUrl*: *string*,

public *createdDate*: *Date*,

public *snaps*: *number*) {

}

}

### Ajoutez des propriétés personnalisées

Pour qu'une propriété puisse être injectée **depuis l'extérieur** d'un component, il faut lui ajouter le décorateur  @Input(). Créez maintenant une propriété  faceSnap  de type  FaceSnap  (votre nouveau type !) et mettez-lui ce décorateur :

@Input() faceSnap!: FaceSnap;

Input  doit être ajouté aux imports depuis  @angular/core  !

Vous allez maintenant utiliser l'**attribute binding** pour lier cet objet à la propriété personnalisée  faceSnap  de FaceSnapComponent dans  app.component.html  :

<app-face-snap [faceSnap]="mySnap"></app-face-snap>

### En résumé

* N'hésitez pas à créer vos propres types, sous forme de classe par exemple, pour faciliter la manipulation de données dans votre application.
* Une propriété personnalisée est rendue injectable depuis l'extérieur grâce au décorateur  @Input()
* Une propriété en  @Input()  est utilisable comme n'importe quelle autre propriété : on peut en afficher les éléments, les modifier…
* On lie ensuite une valeur à cette propriété depuis le component parent avec l'attribute binding, c'est-à-dire le nom de la propriété entre crochets  []  en passant la valeur entre les guillemets ; ex. : [faceSnap]="mySnap"

## Conditionnez l'affichage des éléments

### Mettez en place une structure dynamique

Une **directive**est une classe qui vient **ajouter du comportement** à l'élément sur lequel elle est posée. L'astérisque au début du nom  \*ngIf  nous montre qu'il s'agit d'une directive **structurelle**, qui viendra donc toucher à la structure du document.

### Ajoutez une propriété optionnelle

Le TypeScript nous permet d'ajouter des **propriétés optionnelles** aux classes, aux objets, et même des arguments optionnels aux méthodes. Je vous propose d'ajouter dès maintenant une propriété optionnelle à la classe FaceSnap, qui sera la localisation de l'image. On pourrait imaginer que les utilisateurs puissent choisir d'ajouter ou non cette localisation. Pour ajouter cette propriété optionnelle, il suffit d'utiliser un **point d'interrogation**:

export class FaceSnap {

constructor(public *title*: *string*,

public *description*: *string*,

public *imageUrl*: *string*,

public *createdDate*: *Date*,

public *snaps*: *number*,

public *location?*: *string*) {

}

}

Si vous oubliez une propriété requise, TypeScript vous préviendra que l'objet n'est pas du bon type ! Cependant, vu que  location  est optionnelle, TypeScript ne vous oblige pas à la remplir.

### Créez… ou ne créez pas

Si la localisation existe sur un FaceSnap, vous allez faire en sorte qu'Angular ajoute un paragraphe à votre FaceSnapComponent qui le montre. Si elle n'existe pas, ce paragraphe n'existera pas non plus. C'est aussi simple que :

<p \*ngIf="faceSnap.location">Photo prise à {{ faceSnap.location }}</p>

Vous pouvez y attribuer toute condition TypeScript valable. On pourrait imaginer un filtre où :

<p \*ngIf="faceSnap.location === 'Paris'">

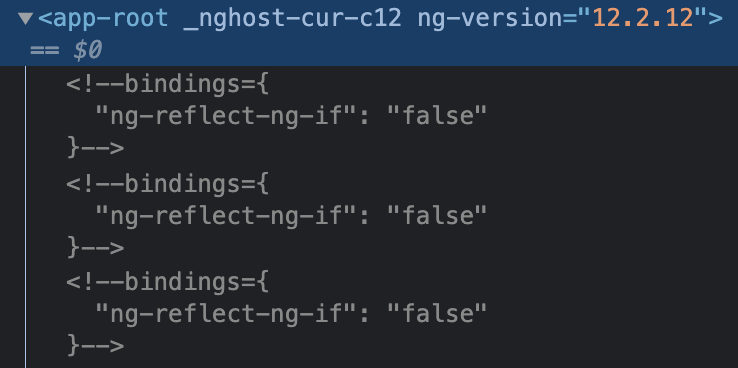
pour afficher uniquement les photos prises à Paris. On peut même mettre la directive  \*ngIf  sur des components. Dans  app.component.html, on pourrait imaginer un filtre pour montrer uniquement les FaceSnaps ayant plus de 5  snaps  :

<app-face-snap [faceSnap]="mySnap" \*ngIf="mySnap.snaps > 5"></app-face-snap>

<app-face-snap [faceSnap]="myOtherSnap" \*ngIf="myOtherSnap.snaps > 5"></app-face-snap>

<app-face-snap [faceSnap]="myLastSnap" \*ngIf="myLastSnap.snaps > 5"></app-face-snap>

Dans mon cas, aucun FaceSnap n'est affiché dans cette situation. Je peux d'ailleurs regarder l'inspecteur d'éléments de mon navigateur :

Aucun élément HTML

### En résumé

* Vous pouvez ajouter des propriétés optionnelles à une classe en y ajoutant un point d'interrogation  ?
* Ajouter la directive  \*ngIf  à un élément (qui peut aussi être un component) dit à Angular d'ajouter ou non cet élément au DOM selon la condition que vous lui passez, par exemple  <p \*ngIf="size === 'large'">

## Affichez des listes

Maintenant, dans le template, au lieu d'insérer plusieurs instances de  <app-face-snap> , vous allez utiliser le code suivant :

<app-face-snap \*ngFor="let faceSnap of faceSnaps" [faceSnap]="faceSnap"></app-face-snap>

## Ajoutez du style dynamique

Puisqu'on a un nom de classe, on pourrait se dire que ce code pourrait se trouver dans  app.component.scss, par exemple ; mais si vous essayez, vous verrez que ça ne fonctionne pas ! Pour expliquer brièvement, Angular limite les styles d'un component au component même : les enfants d'un component **n'héritent pas** des styles de leur parent. Vous allez donc avoir des styles et des classes qui sont **encapsulés**. La seule exception est le fichier de styles principal  styles.scss  qui sert justement à déclarer des styles et des classes globaux.

La fonctionnalité que je vous propose d'ajouter maintenant est que plus un FaceSnap a de "snaps", plus le nombre de "snaps" sera coloré en vert. Pour ceci, vous allez utiliser la directive**[ngStyle]**

La directive  **[ngStyle]**  prendra comme argument un objet où :

* les **clés**sont les styles CSS à modifier ;
* les **valeurs**sont les valeurs que doivent prendre ces styles.

Sous le capot, [ngStyle] ajoute une propriété en  @Input  qui s'appelle  ngStyle . La syntaxe entre crochets fonctionne comme pour la liaison par attribut (attribute binding), sauf qu'elle**crée** la propriété en même temps qu'elle y **lie** votre objet de styles. Ça fait partie des points forts de ce type de directive !

Pour utiliser directement le nombre de snaps comme valeur pour le vert de la couleur de texte, vous pouvez utiliser la fonction CSS  rgb()  :

<span [ngStyle]="{ color: 'rgb(0, ' + faceSnap.snaps + ', 0)' }">🤌 {{ faceSnap.snaps }}</span>

Vous avez implémenté un **style dynamique** qui change avec le nombre de snaps d'un FaceSnap. Il existe d'autres situations où le style n'est pas proportionnel à une valeur dynamique, mais où on souhaiterait pouvoir appliquer une **classe** dans certains cas et pas dans d'autres.

## Mettez de la classe

La meilleure façon de modifier plusieurs styles en même temps est d'utiliser des **classes CSS**, surtout s'il s'agit d'un élément avec plusieurs enfants. Dans ce chapitre, vous allez modifier plusieurs styles des éléments de vos cards quand un utilisateur clique sur "Oh Snap!" avec la directive**[ngClass]**.

### Ajoutez dynamiquement des classes

s FaceSnaps que l'utilisateur a "snapped". On pourra imaginer une classe snapped  qui viendra s'ajouter à la  <div class="face-snap-card">  pour modifier quelques couleurs, par exemple. On ajoute cette classe dans  face-snap.component.scss, car c'est là où elle servira :

.snapped {

*background-color*: rgba(lightgreen, 0.4);

*color*: darkgreen;

button {

*box-shadow*: lightgreen 3px 3px 7px;

*color*: darkgreen;

&:active {

*box-shadow*: lightgreen 0 0 5px;

}

}

}

Maintenant, il va falloir appliquer cette classe **conditionnellement**.

[ngClass]  prend un objet en argument, comme  [ngStyle], mais dont le fonctionnement est différent :

* les **clés** sont les **noms de classe CSS** à appliquer ;
* les **valeurs** sont les **conditions** qui doivent être remplies pour que les classes correspondantes s'appliquent.

En gros :<div [ngClass]="{ 'class-name': condition }"></div>

## Changez la casse

**Résolvez un souci de format**

Dans une application front-end dynamique, vous recevrez souvent des données qui ne sont pas **formatées** correctement pour être affichées. Il est toujours possible de transformer ces données, mais ça crée souvent des problèmes de compatibilité, notamment si on doit renvoyer à un moment ces données au serveur.

Angular vous fournit un type d'outil puissant pour ce cas de figure : les **pipes**. Un pipe est appliqué dans le HTML et va **formater** la valeur qu'on lui passe selon le pipe utilisé **sans toucher à la donnée sous-jacente**. Dans ce chapitre, vous verrez quelques pipes qui existent pour changer la casse d'une chaîne de caractères.

Il existe trois pipes fournis par Angular pour modifier la casse :

* **LowerCasePipe** : on affiche le texte en **minuscules**
* **UpperCasePipe** : on affiche le texte en **majuscules**
* **TitleCasePipe** : on affiche le texte avec une majuscule au début de chaque mot, avec le reste du mot en minuscules

On applique un pipe à une chaîne de caractères affichée avec la string interpolation. On rajoute le caractère pipe  |  puis le nom du pipe. Par exemple, si vous voulez afficher le titre des FaceSnap en majuscules, vous appliquez UpperCasePipe :

<h2>{{ faceSnap.title | uppercase }}</h2>

De même pour LowerCasePipe et TitleCasePipe :

<h2>{{ faceSnap.title | lowercase }}</h2>

<h2>{{ faceSnap.title | titlecase }}</h2>

Pour votre application, choisissez la casse qui marche le mieux avec votre design ! Pour mon design, je vais rester avec les majuscules : je trouve que ça fait bien ressortir les titres.

Je tiens à répéter que les pipes **ne changent rien** à la valeur des variables sous-jacentes. Les pipes existent uniquement pour modifier le **formatage affiché** d'une donnée : on ne peut pas les utiliser ailleurs que dans le template, et il est **fortement déconseillé** de les utiliser ailleurs que dans une string interpolation.

### En résumé

* Un pipe permet de formater l'affichage d'une donnée sans modifier la donnée sous-jacente.
* Angular fournit trois pipes pour modifier la casse affichée d'une chaîne de caractères : UpperCasePipe, LowerCasePipe, et TitleCasePipe pour les majuscules, les minuscules et le mélange type "titre".

## Formatez les dates

Dans votre application, vous affichez la date de création de chaque FaceSnap, mais le formatage affiché n'est vraiment pas top. Vous allez maintenant personnaliser ce formatage pour améliorer l'expérience de vos utilisateurs avec **DatePipe**.

Il est tout à fait possible d'utiliser DatePipe comme vous avez utilisé UpperCasePipe précédemment, et un formatage **par défaut** sera appliqué :

<p>Mis en ligne le {{ faceSnap.createdDate | date }}</p>

La date est formatée avec un format par défautDate formatée

C'est mieux, mais c'est pas encore ça.

Pour **configurer** un pipe, on ajoute **deux-points**  :  puis généralement une **chaîne de caractères** qui définit la configuration. Angular nous propose quelques configurations prédéfinies pour DatePipe, comme par exemple  longDate  :

<p>Mis en ligne le {{ faceSnap.createdDate | date: 'longDate' }}</p>

Le format 'longDate' donne le nom du mois au complet, le jour du mois, puis l'annéeLe format 'longDate'

Vous avez également la possibilité de créer vos propres configurations en passant une chaîne de caractères qui encode votre configuration souhaitée. Voici deux exemples :

<p>Mis en ligne le {{ faceSnap.createdDate | date: 'dd/MM/yy, à HH:mm' }}</p>

Date personnaliséeDate personnalisée

<p>Mis en ligne {{ faceSnap.createdDate | date: 'à HH:mm, le d MMMM yyyy' }}</p>

Une autre date personnaliséeUne autre date personnalisée

Pour consulter toutes les possibilités de configuration personnalisée, ainsi que la liste complète de formats prédéfinis, consultez la [**documentation DatePipe**](https://angular.io/api/common/DatePipe).

Vous aurez certainement remarqué que les dates apparaissent en anglais. En effet, la locale par défaut des applications Angular est **en-US**. Si vous le souhaitez, il est possible de changer la locale par défaut de votre application, ce qui change également le résultat des pipes.

### Changez la locale par défaut

Pour changer la locale de votre application en français, il faudra ajouter quelques lignes dans votre fichier  app.module.ts  :

import { LOCALE\_ID, NgModule } from '@angular/core';

import { BrowserModule } from '@angular/platform-browser';

import { AppComponent } from './app.component';

import { FaceSnapComponent } from './face-snap/face-snap.component';

import { registerLocaleData } from '@angular/common';

import \* as fr from '@angular/common/locales/fr';

@NgModule({

declarations: [

AppComponent,

FaceSnapComponent

],

imports: [

BrowserModule

],

providers: [

{ *provide*: *LOCALE\_ID*, useValue: 'fr-FR'}

],

bootstrap: [AppComponent]

})

export class AppModule {

constructor() {

registerLocaleData(fr.default);

}

}

Vous pouvez bien sûr remplacer **fr-FR** par une autre locale valable.

Nous ne rentrerons pas dans les détails des modifications apportées ici, car les concepts sortent du cadre de ce cours. Pour plus d'informations, consultez la [**documentation i18n**](https://angular.io/guide/i18n-common-format-data-locale).

Une fois la locale changé, vous verrez tout de suite la différence au niveau des dates. Le dernier format ci-dessus, par exemple, donne maintenant :

### En résumé

* DatePipe permet de formater les dates, et sans configuration fournit un formatage par défaut.
* DatePipe fournit des configurations prédéfinies avec des noms comme  short,  longDate  ou  mediumTime
* DatePipe permet également de personnaliser totalement le format d'affichage des dates avec des chaînes de caractères qui encodent le format souhaité, par exemple  'à HH:mm, le d MMMM yyyy'

## Formatez les chiffres

Dans le cadre de ce cours, je voudrais partager avec vous une dernière catégorie de pipes : ceux qui concernent le formatage des **nombres** selon les règles de locale de l'application. Il y en a trois :

* **DecimalPipe** – facilite l'affichage de nombres avec des **chiffres après la virgule** (qui met une virgule plutôt qu'un point, par exemple).
* **PercentPipe** – formate les chiffres en **pourcentage**.
* **CurrencyPipe** – permet d'afficher des nombres sous forme de **monnaie** très facilement.

Ce ne sont pas forcément des pipes que vous utiliserez dans cette application précisément, mais il est important de savoir qu'ils existent.

Sachez qu'il existe d'autres pipes qui seront importants dans la suite de votre apprentissage Angular – notamment le pipe async pour les Observables – que vous ne verrez pas dans ce cours d'introduction. Pour plus d'informations, vous pouvez vous référer à la [**documentation Angular**](https://angular.io/api/common#pipes).

**Comptez sur Angular**

Pour la démonstration des pipes liés aux nombres, ajoutez simplement un paragraphe contenant un **nombre**. Il faut que ce nombre soit entre **doubles accolades** pour que vous puissiez y attribuer un pipe (et aussi pour qu'il soit considéré de type  number  – sinon ce sera une chaîne de caractères qui *contient* un nombre !) :

<p>{{ 4346234.36 }}</p>

J'ai choisi un nombre à 7 chiffres avant la virgule et 2 chiffres après, pour vous montrer les différences de format selon les différents pipes. Regardez déjà l'affichage sans pipe :

Un nombre sans formatage est affiché sans espaces et la virgule est en fait un point !Sans formatage

Si vous avez modifié la locale de votre application pour la passer en français, regardez ce qui se passe quand vous ajoutez **DecimalPipe** avec le mot-clé  number  :

<p>{{ 4346234.36 | number }}</p>

Le nombre a des espaces après les millions et après les milliers, et la virgule est maintenant une virgule !DecimalPipe par défaut

C'est déjà beaucoup mieux ! On peut aussi demander à DecimalPipe **d'arrondir** pour l'affichage, par exemple à **l'entier le plus proche** :

<p>{{ 4346234.36 | number: '1.0-0' }}</p>

La configuration signifie :

* au moins un chiffre pour les entiers ;
* minimum 0 chiffres après la virgule – maximum 0 chiffres (Angular arrondira donc à l'entier le plus proche).

Ce qui donne bien :

Le nombre apparaît correctement arrondiArrondi à l'entier le plus proche

On peut aussi spécifier d'arrondir à **un chiffre après la virgule** en spécifiant un chiffre maximum après la virgule :

<p>{{ 4346234.36 | number: '1.0-1' }}</p>

Le ,36 passe bien en ,4 après arrondiArrondi à un chiffre après la virgule

Plus besoin de se prendre la tête avec la fonction  Math.round()  ! Angular peut afficher les nombres arrondis sans toucher à la donnée sous-jacente.

Pour transformer des nombres entre 0 et 1 en **pourcentage** (ex. : 0.4 = 40 %), il suffit d'utiliser **PercentPipe**:

<p>{{ 0.336 | percent }}</p>

Pour 0.336, Angular nous affiche 34 %En pourcentage, déjà arrondi ?

PercentPipe **arrondit par défaut** les pourcentages à l'entier le plus proche. Vous pouvez cependant le configurer de la même manière que DecimalPipe

**CurrencyPipe**

Quand les nombres dans votre application correspondent à des **montants d'argent**, CurrencyPipe facilite leur affichage. Si vous utilisez l'affichage par défaut :

<p>{{ 344.36 | currency }}</p>

Le nombre est affiché en $USEn dollars

Angular présume par défaut que vous parlez de dollars, et donc affiche votre montant en dollars selon les règles de la locale de l'application.

Vous pouvez configurer la **monnaie** à afficher :

<p>{{ 344.36 | currency: 'EUR' }}</p>

Le montant est affiché en euros, €En euros, €

Vous pouvez passer un **deuxième argument** qui permet d'afficher le code de la monnaie plutôt que son symbole. Pour passer un deuxième argument à un pipe, on remet un deux-points  :  et on passe le deuxième argument :

<p>{{ 344.36 | currency: 'EUR' : 'code' }}</p>

Le montant est affiché suivi de EUR plutôt que le symbole €En euros avec le code EUR

### En résumé

* DecimalPipe (mot-clé  number ) vous permet d'afficher un nombre selon les règles de la locale de l'application.
* PercentPipe affiche un nombre entre 0 et 1 sous forme de pourcentage, arrondi par défaut à l'entier le plus proche.
* DecimalPipe et PercentPipe acceptent un argument de configuration sous la forme *'chiffresMinAvantVirgule.chiffresMinAprèsVirgule-chiffresMaxAprèsVirgule'*ex. :  '1.0-1'
* CurrencyPipe facilite l'affichage des montants d'argent, et accepte plusieurs arguments de configuration séparés par des deux-points  :

## Partagez des données avec les Services

Dans une application complète Angular, vous aurez des dizaines, voire des centaines de components différents. Ces components auront très souvent besoin d'accéder aux **mêmes données**. Il y aura des **interactions** courantes avec ces données, comme l'écriture, la modification et la suppression. L'application peut également avoir besoin d'interagir avec un **serveur** concernant ces données, pour la lecture et l'enregistrement.

Pour mieux **organiser** votre code, pour **éviter de répéter** des blocs et pour toujours avoir des données **à jour** partout dans l'application, ce serait intéressant de pouvoir regrouper les données et leurs méthodes ensemble, de les **centraliser**. En Angular, cette centralisation se fait dans des **services**.

### Partagez les données

Vous allez maintenant créer un service qui va centraliser toutes les interactions avec les FaceSnaps, afin que tous les éventuels components de votre application dépendent des mêmes données.

Dans le dossier  app, créez un dossier  services  et dedans un fichier  face-snaps.service.ts.

Un service est une classe, et la façon la plus simple de déclarer une classe comme étant un service est d'utiliser le décorateur  @Injectable()  qui s'importe depuis  @angular/core  :

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

@Injectable({

providedIn: 'root'

})

export class FaceSnapsService {

}

L'objet de configuration qui spécifie  providedIn: 'root'  dit à Angular d'enregistrer ce service à la **racine** de l'application. Ce sera très souvent le cas pour vos services, car ça permet de s'assurer de n'avoir qu'**une seule instance** du service, **partagée** par tous les partis intéressés.

Le premier élément que vous allez déporter dans votre service est le tableau des FaceSnaps. Il est important de savoir qu'un service n'a **pas de méthode  ngOnInit()**, car les services ne sont pas instanciés de la même manière que les components. Il faudra donc déclarer et initialiser le tableau dans la même expression :

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

import { FaceSnap } from './models/face-snap.model';

@Injectable({

providedIn: 'root'

})

export class FaceSnapsService {

*faceSnaps*: *FaceSnap*[] = [

// vos FaceSnap ici

]

}

### Injectez votre service

Pour pouvoir utiliser un service dans un component, il faut utiliser le système d'**injection de dépendances**(dependency injection ou **DI**) que vous fournit Angular. C'est très simple : vous passez un argument du type du service au constructor du component, et Angular vous mettra à disposition la bonne instance du service. Concrètement pour votre application, dans FaceSnapListComponent :

import { FaceSnapsService } from '../services/face-snaps.service';

//...

constructor(private *faceSnapsService*: *FaceSnapsService*) { }

Pour rappel, ajouter un **modificateur d'accès** comme  public  ou  private  à un argument du  constructor  crée une propriété avec ce nom-là dans la classe. Vous aurez donc accès au service via la propriété  faceSnapsService

On préfère généralement  private  pour les injections de service, car ça empêche le template du component d'y accéder directement. Donner au template accès aux injections serait un **anti-pattern** Angular – autrement dit, une approche plus que déconseillée, souvent pour des raisons qui ne sont pas flagrantes au premier abord.

Maintenant que vous avez accès au service, vous pouvez modifier l'initialisation de la propriété  faceSnaps  du component pour y attribuer le tableau du service :

ngOnInit(): void {

*this.*faceSnaps = *this.*faceSnapsService.faceSnaps;

}

### En résumé

* Les services permettent de centraliser les données et la logique pour les différents domaines de votre application.
* Créer un service est aussi simple qu'ajouter le décorateur  @Injectable()  à une classe.
* Pour injecter un service dans un component, ajoutez un argument au constructor du component qui a le type du service, par exemple private userService: UserService

## Centralisez votre logique avec les Services

Cependant, dans une application totalement dynamique, on peut imaginer que ces FaceSnaps viendraient d'un **serveur**, ou d'une autre partie de l'application, et qu'il faudrait appeler une **méthode** pour les récupérer. D'ailleurs, toute modification d'un FaceSnap entraînerait également un appel au serveur.

Il faudra donc **centraliser** toutes les **interactions** avec les FaceSnaps dans FaceSnapsSerice, et c'est exactement ce que vous allez faire maintenant !

### Centralisez les interactions

La première méthode que vous allez créer est  getAllFaceSnaps(). Cette méthode retournera, comme son nom l'indique, tous les FaceSnaps contenus dans le service.

Il s'agit d'une méthode TypeScript, donc il est vivement conseillé de stipuler son **type de retour**– ici, il s'agit d'un tableau de  FaceSnap :

getAllFaceSnaps(): FaceSnap[] {

return *this.*faceSnaps;

}

Même si le comportement n'est pas modifié pour l'instant, cette structure vous permettrait à terme de changer l'implémentation de  getAllFaceSnaps()  sans casser le reste de l'application.

Ce genre d'architecture **modulaire** est l'un des gros points forts d'Angular.

### Précisez les types avec les Literal Types

Afin de limiter les possibilités à des options sémantiques, on peut remplacer le type  string  par un **literal type**:

snapFaceSnapById(*faceSnapId*: *number*, snapType: 'snap' | 'unsnap'): void {

Créer des types "limités" comme ça, sous forme de **literal types**, ce n'est pas limité aux chaînes de caractères ! Pour plus d'informations sur les literal types, n'hésitez pas à plonger dans la [**documentation TypeScript**](https://www.typescriptlang.org/docs/handbook/2/everyday-types.html#literal-types).

### En résumé

* Centraliser les interactions dans un service sous forme de méthodes crée une structure plus modulaire, qui facilite la maintenance et les évolutions de votre application.
* Comme dans toute base de code, refactorisez pour éviter de répéter des blocs de code (le principe DRY : Don't Repeat Yourself).
* Les literal types permettent de créer rapidement des types personnalisés, souvent utilisés pour limiter les choix pour un argument de méthode, par exemple : fileType: 'image' | 'video'

## Passez en SPA avec le routing

Les **Single Page Applications** (ou **SPA**) sont de plus en plus courantes dans le monde du développement web moderne. Elles permettent une **performance** sans égale en enlevant le besoin d'une application de demander, recevoir, puis afficher une nouvelle page HTML à chaque changement d'URL. Angular vous permet avec le **routing** de créer ce genre d'application – à chaque **URL** correspondra un **component**, et Angular remplacera le component actif sans émettre de requête au serveur.

### Ajoutez un module de routing

D'abord, vous allez ajouter un module de routing à votre application. Dans le dossier  app, créez un fichier  app-routing.module.ts. Commencez par y déclarer une classe avec le décorateur  @NgModule()  :

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

import { Routes } from '@angular/router';

*const* routes: Routes = [

];

@NgModule()

export class AppRoutingModule {}

Ce tableau va lier les **routes** de votre application (les différentes URL) aux **components** correspondants. Créez dès maintenant une première route  facesnaps  qui affichera le component FaceSnapListComponent :

@NgModule()

export class AppRoutingModule {}

Au-dessus de la déclaration de classe, vous allez initialiser une constante appelée  routes, de type  Routes  (importé depuis  @angular/router) et qui contiendra un tableau :

*const* routes: Routes = [

{ path: 'facesnaps', *component*: *FaceSnapListComponent* }

];

Pour enregistrer ces routes dans votre application, il faut les passer au routeur en passant un objet de configuration au décorateur  @NgModule()  de AppRoutingModule :

@NgModule({

imports: [

RouterModule.forRoot(routes)

],

exports: [

RouterModule

]

})

RouterModule est aussi importé depuis  @angular/router.

Ici, vous dites à Angular que les routes de ce fichier seront les routes à la **racine** de votre application, et vous réexportez le routeur configuré. Pour l'utiliser, il faut ajouter votre module de routing aux **imports** de AppModule, le module principal de votre application :

@NgModule({

// ...

imports: [

BrowserModule,

AppRoutingModule

],

// ...

})

export class AppModule {

l ne reste plus qu'à dire à Angular **où** il faut afficher le component dicté par la route. Dans le template d'AppComponent, remplacez la balise  <app-face-snap-list>  par une balise  <router-outlet>  :

<app-header></app-header>

<router-outlet></router-outlet>

Comme ça, ce sera le **routeur** qui décidera quel component doit être affiché, et **AppComponent** qui décidera à quel niveau ce component sera ajouté. Dans ce cas, on a décidé que le AppHeader restera présent peu importe la route active, ce qui est plutôt réaliste : on pourrait même envisager d'y placer un menu de navigation pour l'application.

Le but maintenant sera de créer une landing page pour la route "vide", qui permette d'accéder à la route  facesnaps.

🡪 le path d'une route vide est le string vide

Pour créer LandingPageComponent, vous pouvez utiliser le CLI :

ng g c landing-page

Pour l'enregistrer comme étant le component à afficher pour la route vide, il faut ajouter une route à votre tableau  routes  dans AppRoutingModule :

*const* routes: Routes = [

{ path: 'facesnaps', *component*: *FaceSnapListComponent* },

{ path: '', *component*: *LandingPageComponent* }

];

Pour utiliser des fichiers **statiques** comme l'image ci-dessus, mettez-les dans le dossier  **assets**  du projet ! Vous pouvez bien sûr y créer toute une arborescence : images, fonts, PDF…

Le template :

<img src="assets/snapface.png" alt="Snapface logo">

### En résumé

* Un module de routing contient un tableau de type  Routes  qui contient les routes de l'application.
* Une route est un objet de type  { path: 'myPath', component: MyComponent }  qui spécifie le component à afficher pour chaque route.
* On appelle  RouterModule.forRoot()  en passant le tableau de routes pour enregistrer les routes dans le routeur Angular.
* On enregistre le module de routing dans AppModule pour ajouter le routeur configuré à l'application.
* On ajoute une balise  <router-outlet>  pour dire à quel niveau du template le component de la route active doit être inséré.
* Pour ajouter des fichiers statiques à une application (comme des images), on les stocke dans le dossier  assets  .

## Passez d'une route à l'autre

### Créez des liens avec  routerLink

Le routeur d'Angular vous met à disposition une **directive** pour créer ultra facilement des liens vers des routes de vos applications, et cette directive s'appelle  routerLink. Elle est très facile à utiliser : on lui passe simplement la **chaîne de caractères** qui correspond à la **route** vers laquelle on veut créer le lien. Sur votre landing page :

<a routerLink="facesnaps">Continuer vers Snapface</a>

Dans ce cas, on n'a même pas besoin de crochets ! Les crochets sont nécessaires à partir du moment où vous passez autre chose qu'une chaîne de caractères : un nombre, un objet, ou le nom d'une variable ou propriété, par exemple.

Vous avez peut-être remarqué que je vous ai fourni une classe  active  dans les styles. Il y a une extension de la directive  routerLink  qui permet d'attribuer une **classe CSS** au lien quand sa route est la route **active**:

<header>

<h1>snapface</h1>

<nav>

<a routerLink="" routerLinkActive="active">Home</a>

<a routerLink="facesnaps" routerLinkActive="active">FaceSnaps</a>

</nav>

</header>

Vous passez simplement le classe à attribuer à  routerLinkActive.

Problème, on a un **bug** un peu étrange. Lorsque **Home** est la route active, il n'y a que son lien qui est souligné, mais lorsqu'on est sur **FaceSnaps**, les deux liens sont soulignés. Avez-vous une idée d'où ça peut venir ?

En fait, une route est considérée comme étant active lorsqu'elle-même **ou** l'un de ses enfants est la route sélectionnée. On voit bien que  /facesnaps  est un enfant de  /, donc finalement Angular se comporte normalement. Cependant, il existe une configuration qui permet de considérer uniquement la route exacte et non ses enfants. Il suffit d'ajouter :

<a routerLink="" routerLinkActive="active" [routerLinkActiveOptions]="{ exact: true }">Home</a>

### Naviguez avec le Router

Il existe une autre approche pour changer de route dans une application Angular. Vous pouvez injecter le **Router** dans vos components et changer de route **programmatiquement**.

**Programmatiquement** signifie, en gros, **"dans le code TypeScript"**. Ça veut dire que vous pouvez déclencher les changements de route depuis des **méthodes** dans vos components. Dans ce cas précis, vous allez lier une méthode au clic sur un bouton.

<button (click)="onContinue()">Continuer vers Snapface</button>

Ensuite, pour **injecter** le Router, c'est aussi simple qu'avec un service :

import { Router } from '@angular/router';

//...

constructor(private *router*: *Router*) { }

Maintenant il ne reste plus qu'à appeler la méthode  navigateByUrl()  du Router depuis la méthode  onContinue()  :

onContinue() {

*this.*router.navigateByUrl('facesnaps');

}

### En résumé

* Créez des liens qui permettent de passer d'une route à l'autre avec la directive  routerLink.
* Ajoutez des classes CSS aux liens correspondants à la route activée avec  routerLinkActive.
* Ignorez l'activation des routes enfants avec  [routerLinkActiveOptions]="{ exact: true }".
* Injectez le Router dans vos components et utilisez sa méthode  navigateByUrl()  pour de la navigation programmatique.

## Activez les routes avec ActivatedRoute

Il est très souvent utile dans une application dynamique d'avoir accès à la **route active**. Ce que ça va vous permettre de faire dans ce chapitre, c'est de créer une route avec un **paramètre**  id  pour visualiser un seul FaceSnap.

Vous pouvez dès maintenant créer la **route** qui appellera ce component. Il faut que l'on puisse y passer un **paramètre dynamique**, c'est-à-dire le segment de la route qui contiendra l'id  du FaceSnap, et qui sera donc dynamique. Voilà comment faire :

*const* routes: Routes = [

{ path: 'facesnaps/:id', *component*: *SingleFaceSnapComponent* },

Les  :id  spécifient que ce qui se trouvera après  facesnaps/  correspondra à un paramètre qui s'appellera  id

### Récupérez les paramètres de route

Afin de récupérer les informations de la route activée, vous allez injecter**ActivatedRoute**dans SingleFaceSnapComponent

export class SingleFaceSnapComponent implements OnInit {

faceSnap!: FaceSnap;

buttonText!: *string*;

constructor(private *faceSnapsService*: *FaceSnapsService*,

private *route*: *ActivatedRoute*) {}

ActivatedRoute s'importe depuis  @angular/router.

Ensuite, dans  ngOnInit(), vous allez pouvoir récupérer le paramètre id via le **snapshot** de la route (un **snapshot** est un **aperçu instantané** d'une valeur qui change au cours du temps) :

*const* snapId = +*this.*route.snapshot.params['id'];

Tous les paramètres d'une route sont de type  string, mais l'id  des FaceSnaps est de type  number. Ajouter le  +  au début de l'expression permet de **cast** (changer le type d'une variable) une string  de nombres en  number. Par exemple :

*const* stringValue: *string* = '341';

*const* numberValue: *number* = +stringValue;

// numberValue === 341

Avec cet  id, vous pouvez appeler la méthode  getFaceSnapById()  du service pour récupérer le FaceSnap correspondant :

*this.*faceSnap = *this.*faceSnapsService.getFaceSnapById(snapId);

<a routerLink="/facesnaps">Back</a>

<!-- ... -->

Ici, si vous n'ajoutez pas le  /  au début de la route, le Router essaiera d'accéder à une **sous-route**, et vous finirez avec une URL de type  localhost:4200/facesnaps/3/facesnaps, donc pas du tout ce qu'on veut !

### En résumé

* On récupère les paramètres de la route activée en injectant ActivatedRoute, et via son objet  snapshot.params.
* Pour naviguer vers une route absolue (et non relative), n'oubliez pas d'ajouter un  /  au début de la route demandée.

