

Angular

Introduction à Angular

The bottom of the slide features several overlapping, wavy lines in shades of light blue and white, creating a modern, fluid background element.

L'avancée du Web

Dans l'univers du développement web, les premiers sites furent créés en utilisant uniquement de l'HTML. A cela s'est rapidement ajouté le CSS et le Javascript.



Malheureusement, le monde de l'internet s'est rapidement agrandi et de nombreux projets d'envergures ont vu le jour. De nos jours, il est rare de trouver des sites internet sans inscriptions, formulaires, affichage de multiples pages de détails d'une série de données, animations et autres fonctionnalités demandant un rapport étroit entre les données et le navigateur.

Dans le Javascript traditionnel, la gestion des données est aisée, mais est malgré tout liée à une page. Lorsque l'on clique sur un lien nous amenant vers une autre page du site web, on doit de nouveau récupérer les données et les manipuler pour permettre une interaction. Ce processus aurait un coût significatif en cas de l'utilisation de notre site par des dizaines de milliers de personnes, causant facilement une surcharge de notre serveur en back-end.

Pour pallier à ces problèmes, les framework Front-End sont apparus. En 2022, les frameworks les plus répandus dans le monde du web sont **React**, **Angular** et **Vue**. Ces trois frameworks ont un fonctionnement similaire mais possèdent des différences au niveau de la structure de leurs éléments.

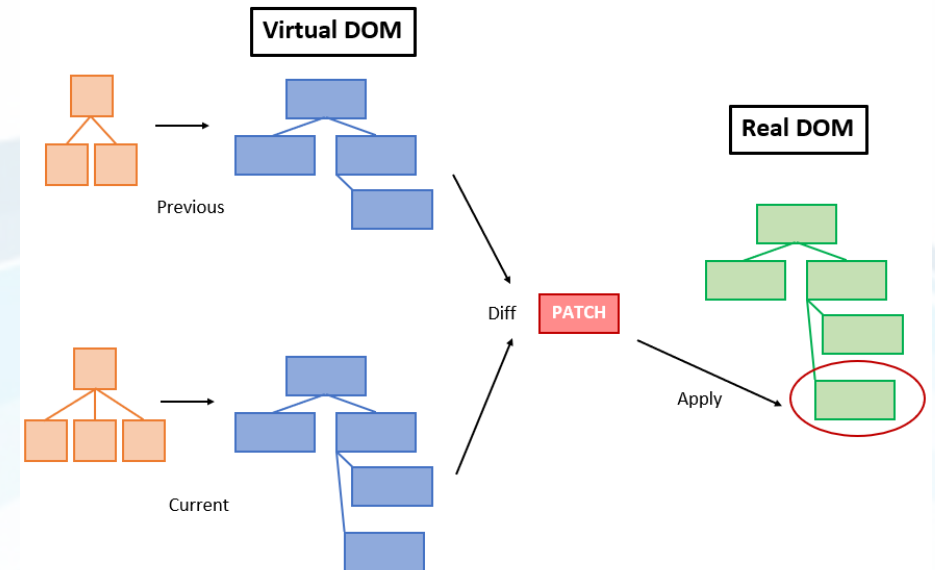
Et Angular ?

@Utopios Consulting

De son côté, Angular est un framework servant à créer des applications clientes web, exécutées sur le navigateur des utilisateurs. Contrairement à ses concurrents, Angular est une plateforme de développement complète qui peut servir à faire des applications web et mobiles. De nombreuses versions d'Angular ont vu le jour, le plus gros changement ayant été entre Angular 1 (Angular JS) et Angular 2.

L'une des particularité majeure d'Angular est de proposer une approche centrée sur les composants, qui sont les briques de notre applications, reliées par le code agissant comme ciment. Le code utilisé lors de la programmation Angular est dérivé de l'ES6, il intègre donc toutes les modifications du Javascript moderne, comme le mot-clé let.

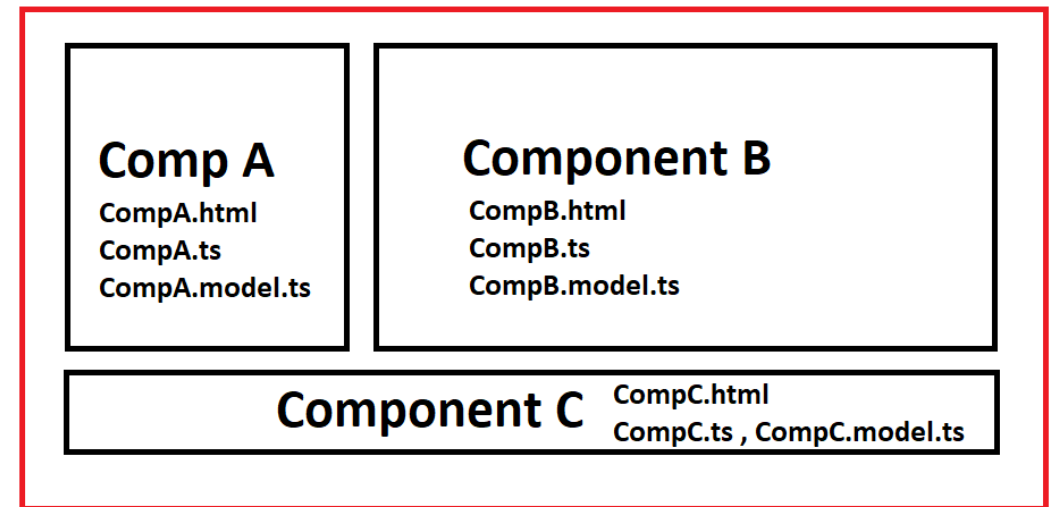
Angular repose également sur le principe de l'utilisation d'un DOM virtuel, qui est une représentation mémoire du DOM. Chaque opération sera donc effectuée sur la mémoire, accélérant grandement notre application, avant de se voir synchronisées vers le DOM physique. Chaque opération peut ainsi être effectuée de façon différentielle en une seule fois, pour un gain de performance majeur.



Une page, des composants

Angular propose nativement une approche basée sur de multiples composants, permettant de reproduire en une ligne d'HTML toute une structure complexe que l'on pourrait remplir différemment ou compléter via le Typescript (une variante typée du Javascript). Cette approche se base sur un fonctionnement en une seule page web, qui sera naviguée via le Typescript et non directement en ouvrant de multiples fichiers HTML. On aura donc une seule et unique page nommée **index.html** qui sera remplie en fonction de nos besoins par le Typescript.

Au niveau de cette page, on pourra naviguer à la façon d'un site classique, et même proposer un système de routing pour simuler le passage d'une page à l'autre. Tout ceci se base néanmoins sur une seule page, ce qui permet à nos données de transiter d'une « page » à l'autre. Via le framework Angular, on peut également créer une seule fois notre barre de navigation et l'injectée en appelant simplement une balise de type `<app-navbar></app-navbar>` dans toutes les pages que l'on voudrait voir pourvue de cette barre de navigation. Ce processus se base sur les composants, l'une des clés de la mécanique de fonctionnement d'Angular.



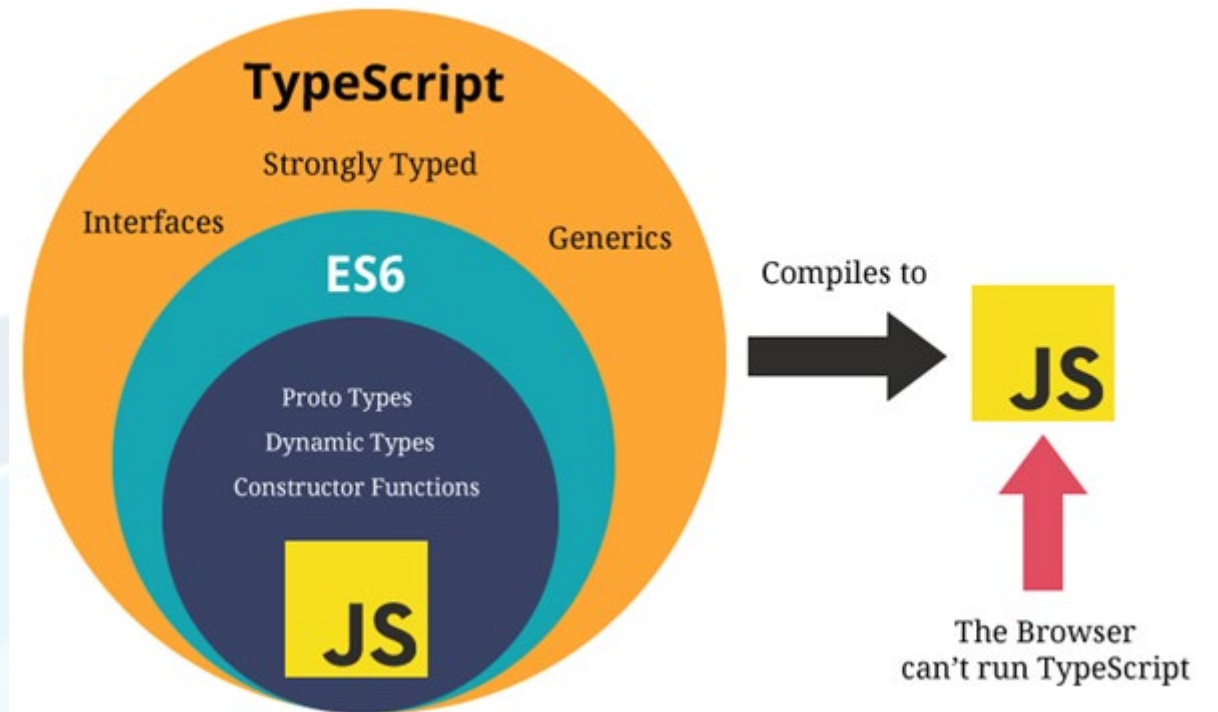
Component X

CompX.html
CompX.ts
??? > CompX.model.ts

Qu'est ce que le Typescript ?

Le Typescript se veut comme une évolution du Javascript. Derrière l'écran, l'ordinateur se charge de lui-même de traduire notre Typescript en Javascript. L'utilisation du TS au lieu du JS a plusieurs avantages :

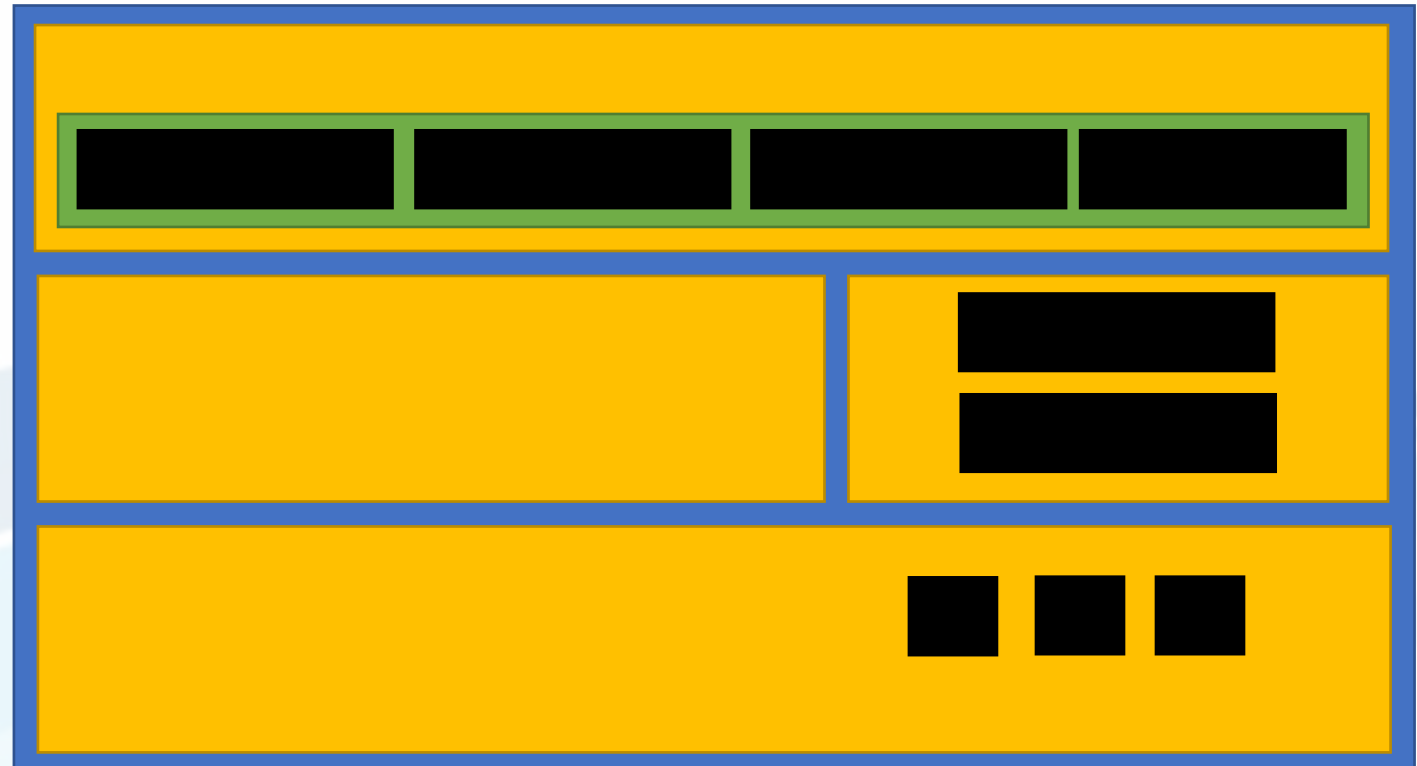
- Le Typescript, comme son nom l'indique, prend en compte le **typage** de nos variables. Il est donc désormais possible, lors de la déclaration d'une variable, de lui donner un type pour sécuriser notre code.
- Le Typescript permet la création de **classes**, nécessaires à la logique de la programmation de type orienté objet. Via les classes, il est possible d'instancier des objets basés sur des moules, comme par exemple une chaise en bois ayant trois pieds ou une chaise en métal ayant 4 pieds.
- Les **interfaces** sont également utilisables. Via l'utilisation d'interfaces, on augmente encore une fois la scalabilité de notre application et on peut sécuriser nos classes en leur forçant à respecter un contrat (par exemple un animal volant doit pouvoir voler et atterrir, doit posséder un moyen de locomotion, etc...) sous peine d'être invalides.



Décomposer notre site Web

Pour concevoir un site réalisé via Angular, il est important de s'imaginer comment briser une page web en plusieurs sous-sections. Par exemple, on peut se dire que notre page **index.html** comporte une barre de navigation dans son header, une section d'informations sur la gauche de notre site, une partie à droit composée de plusieurs images ou publicités et une partie inférieure dans un footer rassemblant l'ensemble des liens vers nos réseaux sociaux et de contact. Notre page peut ainsi être séparée en plusieurs composants. A titre d'exemple, on pourrait avoir un découpage de la sorte (Angular privilégie la séparation de composants, de sorte à pouvoir facilement ré-utiliser ces derniers en cas de besoins futurs) :

- Le composant de Header
 - Un composant de NavBar
 - Plusieurs composants de Boutons
- Le composant de la section gauche
- Le composant de la section droite
 - Les composants de publicités
 - Les composants de vignettes d'images
- Le composant de Footer
 - Plusieurs composants de boutons vers les réseaux sociaux



Notre Premier Projet

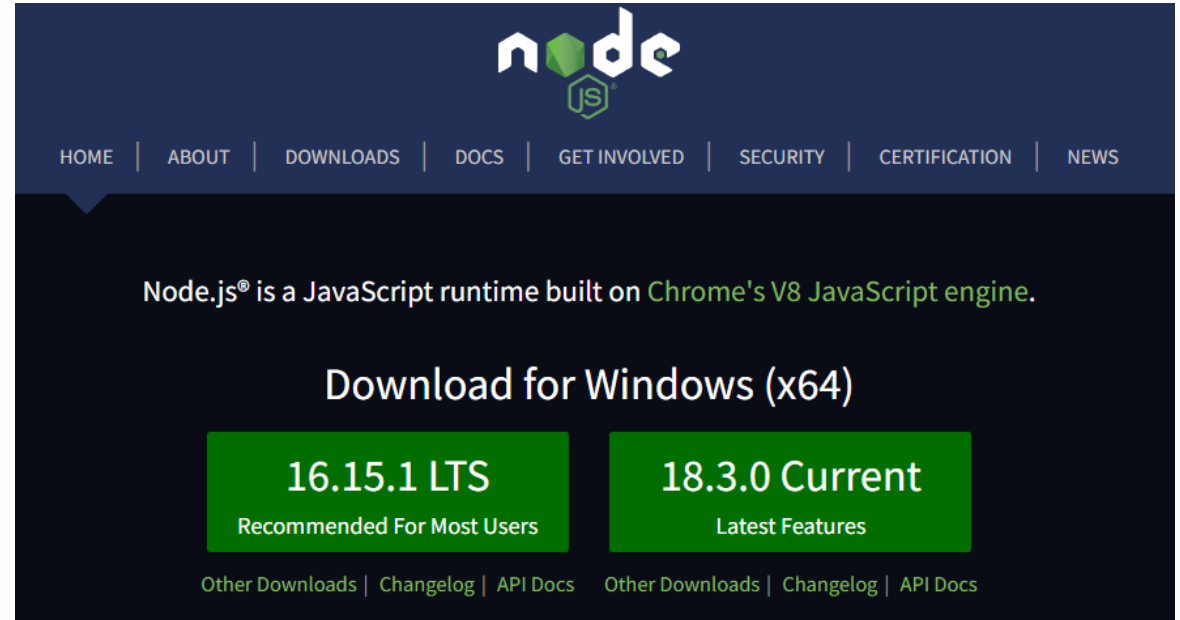
The bottom of the slide features several overlapping, wavy lines in shades of light blue and white, creating a modern, fluid design element.

Installer Angular

Lorsque l'on veut installer et utiliser Angular, il faut déjà être sûr de posséder Node.js, permettant d'obtenir la gestion du Javascript en dehors du navigateur. Pour ce faire, il suffit de se rendre sur le site de Node et de le télécharger. On privilégiera la version LTS (Long Term Support) qui est en général plus stable.

Une fois installé, il nous faudra utiliser notre invité de commande pour entrer la commande d'installation d'Angular via NPM (Node Package Manager). Cette commande se présente de la sorte :

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



Une fois la commande effectuée et Angular correctement installé, on peut créer un nouveau projet via la commande **ng new <nom d'application>** (Attention aux conventions de nommage pour notre application). La commande va créer ainsi un nouveau projet Angular à l'emplacement actuel de notre terminal, nous demandera si l'on souhaite activer le routing et quel type de CSS nous voulons utiliser. Pour commencer, nous créerons des projets sans Routing et usant du CSS classique :

```
PS C:\Users\gharr\source\repos\_codecommit\Formation_M2I_FRONTEND\06_ANGULAR\Examples de Cours> ng new app-01
? Would you like to share anonymous usage data about this project with the Angular Team at
Google under Google's Privacy Policy at https://policies.google.com/privacy. For more
details and how to change this setting, see https://angular.io/analytics. No
```

La Structure d'un projet

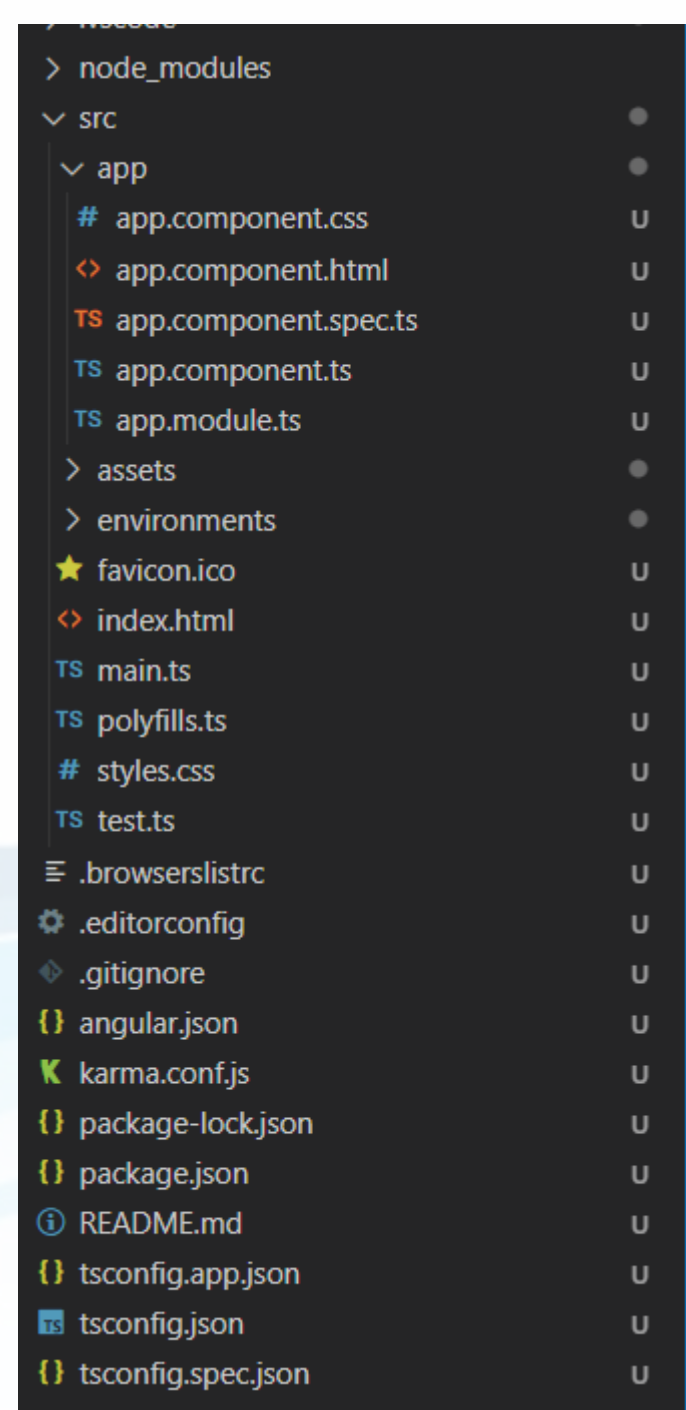
Une fois le projet créé, on peut ouvrir le dossier le contenant avec notre éditeur de code préféré, et en observer la structure.

On remarque tout d'abord que le dossier possède un sous-dossier nommé « **node_modules** ». Ce dossier servira à stocker toutes les dépendances de de code métier que notre application pourrait demander.

Le second dossier, « **src** », contient réellement notre application. Ce dossier possède un fichier **index.html**, qui est la page d'ouverture de notre application. C'est ce fichier HTML qui sera la cible majoritaire de notre Typescript pour la manipulation du site.

Dans ce dossier d'application se trouve un autre sous dossier nommé « **app** ». Ce dossier contiendra les modules, services, composants et autres éléments spécifiques à Angular de notre application. Ainsi, notre application se trouve dans le dossier « racine » app, mais un composant de header pourrait se trouver dans son propre sous-dossier nommé « **header** ». La séparation de notre code dans cette hiérarchie permet une scalabilité et une maintenabilité de notre application plus aisée.

Enfin, le fichier **angular.json** contient la configuration de notre application. C'est ce fichier qu'il faudra modifier en cas d'ajout de dépendance (par exemple Bootstrap).



Ajouter Bootstrap

Pour notre premier projet, nous allons reprendre l'utilisation de Bootstrap dans le but de nous faciliter la création d'une première page web. Pour ce faire, il va nous falloir l'installer tout d'abord.

Vu que nous sommes en train d'utiliser Node.js, il nous suffit d'ajouter le package de Bootstrap puis de créer un lien vers le style de Bootstrap. Pour ce faire, il suffit, dans un terminal ouvert à la racine de notre projet Angular (là où se trouve notre dossier **node_modules**) puis d'entrer la commande **npm install --save bootstrap@5**

Une fois fait, il nous faudra naviguer jusqu'à notre fichier **angular.json** dans le but d'en changer la configuration. L'option à changer se trouve dans **projets : <nom-app> : architect : build : options : styles**

```
27     "src/assets"  
28   ],  
29   "styles": [  
30     "node_modules/bootstrap/dist/css/bootstrap.min.css",  
31     "src/styles.css"  
32   ],  
33   "scripts": []
```

Cette propriété modifiée, il nous suffit désormais de lancer notre site via la commande du terminal **ng serve** pour voir s'afficher sous nos yeux notre site internet.

Notre Premier Composant

Lorsque l'on travaille avec Angular, il nous faut posséder un composant racine servant de point d'entrée à nos autres futurs composants. Par défaut, Angular nous fourni un composant nommé **AppComponent** qui contient dans son dossier des fichiers Typescript, HTML, CSS et potentiellement des fichiers de test.

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'app-01';
}
```

Un composant Angular est une classe décorée de **@Component**. Dans les options de son décorateur se trouvent plusieurs informations :

- **Le sélecteur** : Il spécifie la balise HTML qui servira à injecter notre composant. A chaque fois qu'Angular croisera cette balise dans notre code HTML, un composant y sera injecté
- **La template HTML** : Celle-ci peut soit être écrite en dur dans le fichier du composant, ou alors reliée par une URL vers un fichier HTML, permettant ainsi une séparation des données pour un travail en groupe plus aisé.
- **La template CSS** : Tout comme l'HTML, celle-ci peut être soit de type inline, soit reliée à un fichier CSS placé généralement à côté du fichier de composant.

Par défaut, la CLI va nous donner une template HTML pré-faite afin de nous permettre de lancer notre application et d'éviter d'avoir un écran blanc et notre doute comme seul camarade.

Notre Premier Module

A côté des composants, il existe également des modules. Un module est un élément Angular permettant de rassembler une série de composants, de services et d'autres outils en une entité distincte. Pour fonctionner, Angular a au moins besoin d'un module, et nous en fournis d'ailleurs un par défaut : AppModule.

```
@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Un module se représente par une classe Typescript décorée cette fois-ci de **@NgModule** et possédant plusieurs informations spécifiques au système des modules :

- **Les Imports** : Ils servent à Angular à savoir quels sont les autres modules que notre module pourrait exiger pour fonctionner.
- **La Déclaration** : Cette section sert à notre module pour indiquer quels seront les éléments et les composants qu'il faudra inclure dans le module.
- **Le Bootstrap** : Cette partie sert à indiquer quel sera l'élément de démarrage de l'application.

Notre module de base importe par défaut **BrowserModule**, qui nous permet ainsi de bénéficier de toutes les fonctionnalités de base qu'on pourrait vouloir dans un navigateur internet. De plus, notre module importe par défaut notre composant et le déclare en tant que tel, de sorte à pouvoir l'utiliser par la suite.

Comment se lance notre site ?

Lorsque l'on débute Angular, il est fréquent de se poser les questions de : Comment notre site web va être lancé ? Quels fichiers vont devoir être modifiés pour voir s'opérer un changement ? Quels fichiers ajouter et où pour mettre en avant de nouvelles fonctionnalités ? Toutes ces questions trouvent réponse dans la façon de fonctionner d'Angular et dans le fichier **index.html**. En ouvrant ce fichier, on remarque que notre body ne contient qu'une seule balise :

```
<body>  
  <app-root></app-root>  
</body>
```

Cette balise sert en réalité de point d'entrée dans notre application, comme spécifié dans le fichier **app.component.ts** :

Si l'on explore également le fichier **main.ts**, on peut voir une fonction « **enableProdMode()** » qui sert à Angular pour améliorer l'application en production et nous permettre un débogage plus aisé en mode développement de part son absence.

L'autre fonction « **platformBrowserDynamic()** » sert quant à elle au démarrage du module sur notre navigateur.

Pour lancer notre application, il nous suffit simplement d'utiliser la commande **ng serve** dans notre terminal pour compiler puis lancer notre application dans le navigateur (un lien nous sera proposé pour y accéder). Chaque modification de notre code va automatiquement être détectée à la sauvegarde, et un rafraichissement de notre application aura lieu. Il est donc tout à fait possible de travailler en ayant notre page tournant dans une autre fenêtre de notre ordinateur.

Les Fondamentaux

The bottom of the slide features a series of overlapping, wavy lines in shades of light blue and white, creating a modern, fluid background element.

Créer un composant

Pour faire un composant de façon manuelle, il va nous falloir créer au minimum un fichier se terminant par l'extension **.component.ts**, par exemple, on pourrait avoir un composant nommé « **home.component.ts** » En règle générale, on place les fichiers de notre composant dans un sous-dossier portant son nom.

Une fois fait, il nous faut faire en sorte que ce composant soit exportable afin de pouvoir l'inclure dans un module futur. Pour ce faire, il nous faut ajouter une instruction d'export, par exemple lors de la déclaration de notre classe. Il nous faudra également, comme vu précédemment, ajouter le décorateur **@Component** (ne pas oublier l'import du décorateur) et l'alimenter par un objet contenant des propriétés comme le **template HTML**, le **template de style** et le **sélecteur**. On obtient donc un fichier ressemblant à ceci :

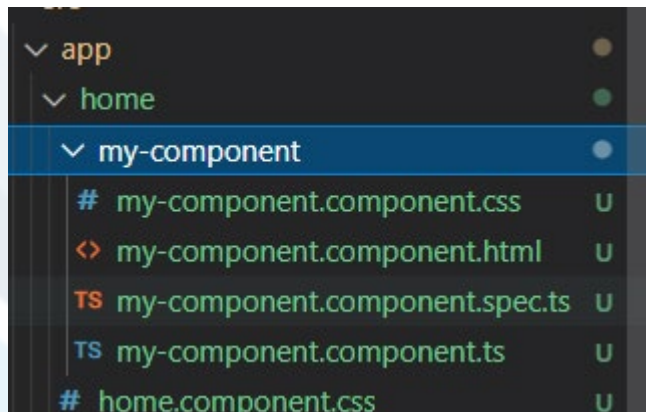
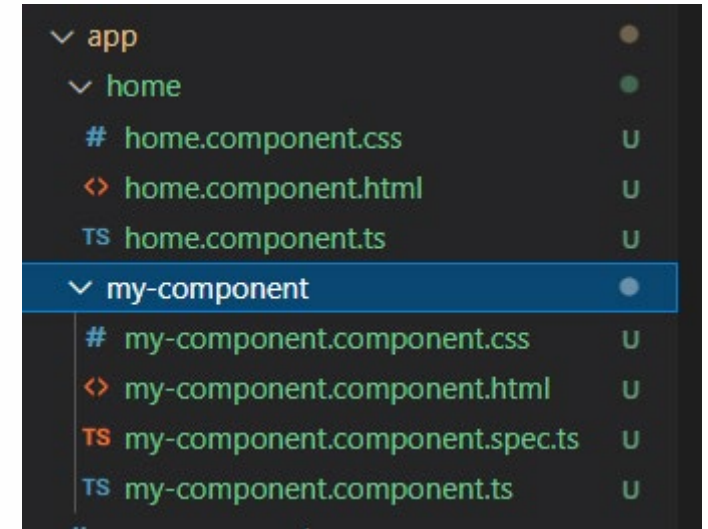
```
import { Component } from "@angular/core";

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

Une fois ceci fait, il va nous falloir le lier à notre module pour pouvoir l'utiliser dans notre application.

Créer un composant avec la CLI

Il est possible et même plus rapide de créer des composants (et d'autres choses) via l'utilisation de la CLI. Pour ce faire, il va nous falloir nous rendre dans notre terminal et utiliser la commande **ng generate component** (ou **ng g c**) puis taper le nom de notre composant. A droite, on utilise par exemple la commande **ng g c my-component**, ce qui nous crée un composant se trouvant à la racine du dossier **app**



ng g c home/my-component : Par défaut, ce composant sera créé dans le dossier **app**, et créera un sous-dossier portant son nom. Si l'on veut placer ce nouveau composant dans une hiérarchie différente, il nous suffit de modifier la commande pour prendre en compte cette hiérarchie, comme dans l'exemple ci-contre.

Si l'on ne souhaite pas disposer à l'issue de la génération d'un fichier de tests portant l'extension **.spec.ts**, il nous suffit d'ajouter comme paramètre de notre commande de génération **--skip-tests**

Relier un Composant au Module

Pour créer cette liaison entre le module et le composant, il nous faut nous rendre dans le module que l'on souhaite utiliser pour faire apparaître notre composant. Par exemple, **AppModule** fera l'affaire. Nous nous rendons donc dans le fichier **app.module.ts** pour y ajouter un import de notre composant ainsi qu'une déclaration du composant. Ainsi, Angular pourra savoir qu'on compte se servir du composant **HomeComponent** dans notre module **AppModule**, et saura où trouver les fichiers Typescript de notre classe.

Ces fichiers reliant la classe à ses templates, nous avons ainsi une série de liens permettant à notre composant de fonctionner pleinement une fois l'application compilée.

```
3
4 import { AppComponent } from './app.component';
5 import { HomeComponent } from './home/home.component';
6
7 @NgModule({
8   declarations: [
9     AppComponent,
10    HomeComponent
11  ],
12  imports: [
13    BrowserModule
```

Une fois la configuration de nos liaisons effectuées, il ne nous reste plus qu'à ajouter notre balise portant le nom de notre sélecteur dans l'un de nos composants parents. Par exemple, dans le fichier **app.component.html**

```
Go to component
<h1>Bienvenue sur Angular</h1>

<app-home></app-home>
```

Le Sélecteur du composant

Dans notre fichier Typescript du composant, nous avons défini un sélecteur de type balise. Il est intéressant de savoir qu'il est également possible d'avoir recourt à un sélecteur de type attribut ou de type classe. Pour cela, il nous suffit de changer la syntaxe de notre sélecteur dans le fichier Typescript :

- **selector** : Va sélectionner une balise de type : `<selector> </selector>`

```
selector: "app-home",
```

```
<app-home></app-home>
```

- **[selector]** : Va sélectionner un élément HTML possédant comme attribut notre sélecteur : `<div selector></div>`

```
// selector: "app-home",  
selector: "[app-home]",  
// selector: "app-home"
```

```
<div app-home></div>
```

- **.selector** : Va sélectionner un élément HTML possédant comme classe notre sélecteur : `<div class="selector"></div>`

```
// selector: "[app-home]",  
selector: ".app-home",  
templateUrl: "/home.com"
```

```
<div class="app-home"></div>
```

Malgré la possibilité pour Angular d'injecter des éléments via les sélecteurs de type attributs ou de type classe, il est fréquent de voir les composants utiliser des sélecteurs de type balise.

Exercice : Création de Composants

Objectif

Appréhender la manipulation et la création de composants dans le framework Angular

Sujet

En vous servant du framework Angular, vous devrez, au sein d'une même page internet, posséder plusieurs élément possédant un style d'alert (Bootstrap). Ces éléments seront différents et posséderont un style et un message adapté à leur cas.

Par exemple, vous pourrez avoir une alerte possédant un message de confirmation, une alerte d'information et une alerte d'erreur, qui posséderont chacune leur composants, template et style à part.

Chacune de ses alertes devra se situer l'une au dessus de l'autre dans une même page.

Exercice : Création d'une page d'Accueil

Objectif

Appréhender la manipulation et la création de composants dans le framework Angular

Sujet

En vous servant du framework Angular, vous devrez, au sein d'une même page internet, réaliser une page d'accueil en vous servant de plusieurs composants.

Le premier composant sera votre header, qui contiendra le haut commun de vos futurs pages Internet (pensez à la barre de navigation).

Viendra ensuite le composant de votre corps de contenu, permettant l'affichage de votre page et de son contenu à proprement parlé. Par exemple, vous pourriez y placer un message d'accueil ainsi que certaines informations sur le contenu de votre site.

Le troisième composant sera votre footer, qui contiendra les liens vers les réseaux sociaux de votre site ainsi que de potentiels crédits.

Qu'est-ce que le Databinding ?

Lorsque l'on veut pouvoir produire rapidement un site, il nous faut pour cela pouvoir rentrer des données préexistantes facilement en les adaptant à une mise en forme pré-faite. Il nous faut également être capable d'altérer nos données lorsque l'utilisateur manipule notre site internet. Pour cela, nous avons recourt au Data Binding (Liaison des données).

Ce databinding se présente de plusieurs façon :

- Lorsque l'on veut remplir les données de notre HTML à partir de ce qui se trouve dans notre logique métier et dans notre Typescript : Pour ce faire, on peut avoir recourt aux Strings Interpolés (**String Interpolation**) via une syntaxe de type `{{données}}` ou à la liaison des propriétés de balises (**Property Binding**) par la syntaxe du `[propriété]="données"`.
- Si l'on veut naviguer dans l'autre sens, comme par exemple réagir à la modification d'un input par l'utilisateur ou le clic d'un bouton de notre page, il nous faut nous lier cette fois ci aux évènements du DOM (**Event Binding**) avec cette syntaxe : `(évènement)="fonction()"`.
- Il est également possible d'avoir recourt à une liaison bi-directionnelle, qui permet un rafraichissement des données présentées sur notre page en temps réel et une modification de notre code en réaction à l'utilisateur de notre application. On parle alors de **Two-Way Data Binding**. La syntaxe pour ce mode de liaison est par exemple `[(ngModel)]="données"`

Les Strings Interpolés

Le fonctionnement des strings interpolés est assez simple. Il suffit de récupérer les noms de variables que l'on aurait pu mettre dans notre code Typescript et de les placer à des endroits de notre template HTML. Ce qui est placé entre les deux doubles accolades de la syntaxe des strings interpolés doit cependant absolument se finir sur la forme d'une chaîne de caractère.

```
}  
export class HomeComponent {  
  myDogName: string = "Bernie";  
  myDogAge: number = 4;  
  myDogBreed: string = "German Sherpard";  
}
```

Le fichier Typescript

```
<p>  
J'aime les chiens. Je possède un chien qui s'appelle {{ myDogName }},  
est de race {{ myDogBreed }} et a {{ myDogAge }} ans !  
</p>
```

La template HTML

Il n'est pas possible de placer de la logique métier, comme par exemple des conditions de type IF / ELSE dans une string interpolée. Il est cependant possible d'avoir recours à l'opérateur ternaire :

```
<p>Mon chien est {{ myDogAge > 2 ? "Majeur" : "Mineur" }}</p>
```

Le Property Binding

Si l'on veut contrôler le contenu de notre template via le Typescript, il est également possible de le faire via le Property Binding. Le property binding est très utile par exemple pour remplir des état de formulaires, comme par exemple cocher ou décocher une checkbox, rendre un input écrivable ou le passer en lecture seule, etc...

```
export class HomeComponent {  
  canAddDog: boolean = false;  
}
```

Le fichier Typescript

```
<button class="btn btn-primary" [disabled]="!canAddDog">  
  <i class="bi bi-plus-circle"></i> Ajouter un chien !  
</button>
```

La template HTML

Pour réaliser une liaison sur une propriété, il suffit de placer le nom de la propriété que l'on veut lier entre crochet, puis de la lier à une variable présente dans notre fichier Typescript du composant. Ainsi, on peut remplir dynamiquement chacune des propriétés HTML de nos balises via le Typescript. Angular a comme particularité le fait de rafraichir automatiquement ce genre de propriétés si jamais les variables étaient modifiées dans notre code métier. Ainsi, on s'assure d'être toujours à jour dans l'affichage, peu importe ce qu'il peut arriver.

L'Event Binding @Utopios Consulting

Dans le cas où l'on veut que ce soit la template qui modifie notre code logique, il nous faut nous lier au niveau des évènements. Pour ce faire, il faut déjà posséder une méthode à laquelle lier notre évènement, celle-ci devant se trouver dans notre code métier. Grâce à l'évènement, nous allons pouvoir déclencher la méthode, et ainsi effectuer des modification de nos données dans les instruction Typescript. Par exemple, si l'on veut modifier une variable en appuyant sur un bouton, on peut le faire de la sorte :

```
dogDeletedMessage: string = "";

onDeleteDog() {
  this.dogDeletedMessage = "A dog was deleted...Poor Soul....";
}
```

Fichier Typescript

```
<button class="btn btn-danger" (click)="onDeleteDog()">
  <i class="bi bi-trash"></i> Supprimer un chien
</button>
```

Template HTML

On pourrait également placer une série d'instruction de façon inline dans notre liaison d'évènement, mais ce processus est peut recommandé car il alourdi grandement les template HTML. De plus, il nuit au principe du « **Separation of Concern** », en rassemblant au même endroit la structure et la logique de notre application.

Récupérer les valeurs de l'Event

Bien entendu, notre event binding ne nous permet pas que de lancer l'exécution d'une méthode sans paramètres. Il est possible par exemple de modifier à la volée la valeur d'une variable via l'input de notre template. Pour cela, il va falloir se servir d'un paramètre noté **\$event**. Ce paramètre peut être récupéré au niveau de notre logique Typescript et décortiqué pour en extraire les propriétés qui nous intéressent (il contient de base toutes les données émises par notre événement). Une fois fait, leur traitement est désormais possible, et il est ainsi aisé de manipuler réactivement nos données depuis les fichiers de templates de nos composants.

```
<input
  type="text"
  class="form-control"
  name="dog-name"
  id="dog-name" |
  placeholder="Bernie"
  (input)="onChangeDogName($event)">
```

Template HTML

```
onChangeDogName(event: Event) {
  this.myDogName = (<HTMLInputElement> event.target).value;
}
```

Logique Typescript

Le Two-Way Data Binding

Pour utiliser le Two-Way Data Binding, il faut déjà s'assurer que l'on possède l'import **FormsModule** dans notre module de base (AppModule). Cet import va nous permettre d'utiliser la directive **[(ngModel)]** dans notre template HTML. C'est grâce à cette directive que l'on peut procéder à ce type de liaison de données. Pour cela, il ne faudra pas oublier de l'importer depuis **@angular/forms**.

```
<input type="text" class="form-control"  
  name="dog-name" id="dog-name" placeholder="Bernie"  
  [(ngModel)]="myDogName">
```

Template HTML

```
myDogName: string = "Bernie";
```

Fichier Typescript

Le Two-Way Data Binding nous offre la possibilité d'avoir une actualisation de nos données depuis les éléments HTML, mais également l'inverse. Il combine ainsi les deux sens de transfert de l'information, et est au cœur de la création d'une application moderne ou de formulaires plus poussés.

Exercice : Les Début du Binding

Objectif

Appréhender la manipulation et l'utilisation du Data Binding dans le framework Angular

Sujet

En vous servant du framework Angular, vous devrez réaliser un formulaire permettant à un utilisateur d'entrer un nom d'utilisateur, et de le remettre à zéro via le clic sur un bouton.

Ce bouton ne devra être cliquable que lorsque le nom entré par l'utilisateur ne sera pas vide.

De plus, ce nom devra être inscrit quelque part sur votre page internet et se mettre à jour à chaque modification de l'input.

Qu'est-ce qu'une Directive ?

Les Directives sont des instructions dans le DOM. Par exemple, notre directive **[(ngModel)]** va servir à Angular pour comprendre que l'on veut lier à un emplacement du DOM des données de notre logique métier.

Les composants sont également des directives, mais celles-ci possèdent une template. Il existe bien entendu d'autres directives ne se servant pas de templates, et il est possible de créer nos propres directives tout comme on peut créer nos propres composants. Contrairement aux composants, les directives ont pour habitude de se fixer au niveau des attributs des classes, et non des noms d'éléments HTML.

Nous allons commencer par voir les directives les plus utilisées en Angular, celles-ci étant présentes de base dans un projet d'application :

- **ngIf** : Permet de rendre conditionnel l'affichage d'un composant en se basant sur une variable booléenne.
- **ngFor** : Permet de multiplier l'apparition d'un composant en se basant sur une variable donnée.
- **ngStyle** : Permet de styliser dynamiquement les composants de notre application.
- **ngClass** : Permet de modifier les classes de notre élément de façon dynamique.

Conditionner avec ngIf

Si l'on veut rendre la présence d'un composant ou d'un élément de l'HTML conditionnée à une variable de notre code, le moyen le plus simple est de se servir de la directive ***ngIf**. Les directives possédant un astérisque en premier caractère de leur nom sont des directives structurelles, c'est-à-dire qu'elles altèrent le DOM et en modifient la structure. C'est le cas de la directive ngIf, qui ajoute ou retire un élément du DOM en fonction d'une variable.

```
<p *ngIf="myDogBorn">Un chien est né !</p>
```

Dans les paramètres de la directive, il nous faut placer n'importe quelle expression renvoyant un booléen. Cette expression peut également être une méthode ou une fonction qui va renvoyer une valeur de ce type. Il est également possible d'ajouter un « sinon » à notre directive, pour proposer un affichage différent en cas de valeur booléenne fausse. Pour ce faire, il nous faut altérer légèrement notre directive et ajouter un élément de type ng-template qui porte une référence locale (symbolisée par le hashtag placé devant son nom) :

```
<p *ngIf="myDogBorn; else noDog">Un chien est né !</p>
```

```
<ng-template #noDog>
```

```
  <p>Aucun chien n'est né...</p>
```

```
</ng-template>
```

Styliser avec ngStyle

Si l'on souhaite ajouter des propriétés de style de façon dynamique à nos éléments, il est possible de le faire via l'utilisation de la directive ngStyle. N'étant pas une directive de structure, cette directive ne possède pas d'astérisque au début de son nom. Seule, cette directive n'est pas très utile, mais il est possible de créer un property binding sur elle afin de pouvoir modifier ses caractéristiques en adéquation avec un objet portant les styles en tant que série de clés et de valeurs. Cette série de clés-valeurs peut se faire à la façon du CSS ou à la façon du Javascript On obtient donc l'une des deux syntaxes de ce type :

```
<p [ngStyle]='{'background-color': getColor()}'>  
  Mon chien est {{ myDogAge > 2 ? "Majeur" : "Mineur" }}  
</p>
```

Version CSS

```
<p [ngStyle]='{backgroundColor: getColor()}'>  
  Mon chien est {{ myDogAge > 2 ? "Majeur" : "Mineur" }}  
</p>
```

Version Javascript

On voit également qu'il est possible de récupérer les valeurs de nos propriétés via une méthode, celle-ci devant bien entendu nous renvoyer les valeurs du type nécessaire à notre CSS. La méthode utilisée ici est disponible ci-dessous :

```
getColor() {  
  return this.myDogAge >= 2 ? "orange" : "dodgerblue";  
}
```


Modifier les classes avec ngClass

Dans une application web, il se peut que l'on ait besoin de modifier rapidement et dynamiquement les classes possédées par nos éléments HTML. Pour ce faire, via Angular, il nous est proposé d'utiliser la directive **ngClass**. On pourrait ainsi imaginer une classe regroupant plusieurs propriétés CSS pour nos chiens adultes et qui ressemblerait à peu près à ça :

```
.adult-dog {  
  font-family: Impact, Haettenschweiler, 'Arial Narrow Bold', sans-serif;  
  font-size: 1.1rem;  
  color: white;  
  background-color: orange;  
}
```

Pour appliquer ou non cette classe (et ainsi paramétrer la présence ou non des propriétés CSS qui lui sont associées), on peut se servir de notre directive, qui se présente aussi sous la forme d'une liaison de propriété. Cette directive doit prendre en paramètre un autre objet Javascript, dont chaque clé correspond à une classe (on peut également placer le nom sous forme d'une string en cas de nommage spécifique) et chaque valeur représente les condition d'attachement de ces classes à notre élément HTML.

```
<p [ngClass]="{'adult-dog': myDogAge >= 2}">  
  Mon chien est {{ myDogAge > 2 ? "Majeur" : "Mineur" }}  
</p>
```


Lister et Itérer avec ngFor

Lorsque l'on produit une application, il est fréquent que l'on ait besoin de manipuler un nombre suffisamment grand de données pour nécessiter l'utilisation d'itérateur, de listes, de tableaux et autre systèmes relatives aux variables de type conteneur. Angular nous fournit une directive permettant de rapidement produire des éléments du DOM en fonction d'une variable de ce type : ***ngFor**. Cette directive de structure fonctionne en se basant sur une variable de type tableau dans notre code métier, et rafraichira notre affichage en cas de modification du contenu de cette variable.

```
myDogToys: string[] = ["Ball", "Plastic Bone", "Bear Plush", "Mashed Rope"]
```

La structure dans notre template HTML, comblée avec une utilisation des chaînes de caractères interpolées, se traduirait de la sorte :

```
<h2>Liste des jouets de mon chien :</h2>
<ul>
  <li *ngFor="let toy of myDogToys">{{ toy }}</li>
</ul>
```

Améliorer l'utilisation de ngFor

Si l'on souhaite utiliser ***ngFor** pour lister nos objets, il peut être intéressant d'avoir accès à l'index de l'élément de notre tableau. Pour se faire, il va falloir altérer légèrement notre directive pour lui faire créer une variable représentant l'index. Pour ce faire, il faut donner une syntaxe de la sorte à notre instruction précédente :

```
<tbody>
  <tr *ngFor="let toy of myDogToys; let i = index;">
    <td>{{ i }}</td>
    <td>{{ toy.category }}</td>
    <td>{{ toy.name }}</td>
    <td>{{ toy.price }}</td>
    <td>{{ toy.quantity }}</td>
  </tr>
</tbody>
```

Plusieurs éléments peuvent ainsi être récupérés en plus du contenu de notre liste :

- **index** : Permet l'accès à l'index de l'élément actuellement itéré
- **first / last** : Permet de savoir si l'élément actuellement itéré est le premier / le dernier de la liste
- **even / odd** : Permet de savoir si l'élément actuellement itéré possède un index pair / impair

On peut également voir ici que de part l'utilisation d'un tableau d'objets, il nous est possible de facilement créer des cellules pour notre tableau se basant sur les propriétés de chacun des objets de notre liste.

Exercice : Les Débuts du Binding #2

Objectif

Appréhender la manipulation et l'utilisation du Data Binding dans le framework Angular

Sujet

En vous servant du framework Angular, vous devrez réaliser une page offrant un bouton qui, une fois cliqué, vous permet d'accéder ou non à une partie secrète de la page.

Cette partie secrète devra contenir à la fois un texte la décrivant comme un secret, et un compteur de nombres de clics que le bouton. Le compteur des clics devra offrir sous la forme d'une liste leur date et heure de clic.

De plus, cette liste devra donner un fond coloré en bleu au descriptif du clic à partir du 5^{ème} clic en se servant de la directive **ngStyle**. Elle devra également, en se servant de **ngClass**, donner une couleur blanche au texte.

Objectif

Maîtriser la manipulation et l'utilisation du Data Binding dans le framework Angular

Sujet

En vous servant du framework Angular, vous devrez réaliser une application de compteur qui permettra d'expérimenter les règles du jeu de FizzBuzz. Voici les règles :

- Lorsque le nombre est **divisible par 3**, on doit afficher **Fizz**
- Lorsque le nombre est **divisible par 5**, on doit afficher **Buzz**
- Lorsque le nombre est **divisible par 3 et par 5**, on doit afficher **FizzBuzz**
- **Dans le cas contraire**, on affiche seulement la valeur du **nombre**

En plus de cela, vous devrez, via l'utilisation de la directive **ngClass**, modifier la police de caractère afin de donner plus d'importance aux mots Fizz, Buzz et FizzBuzz. Vous devrez également, via l'utilisation de la directive **ngStyle**, donner une couleur différente à votre affichage en fonction de s'il s'agit d'un nombre, de Fizz, de Buzz ou de FizzBuzz.

@Utopios Consulting

Les Pipes

Qu'est-ce qu'un Pipe ?

Lorsque l'on manipule des données, il n'est pas rare de devoir les modifier à la volée pour leur donner un aspect plus adapté à notre front-end. Pour cela, Angular nous fournit ce qui s'appelle les pipes. Par exemple, une date stockée sous la forme de **Sun Jul 10 2022 17:00:00 GMT+0200 (Paris, Madrid)** pourrait, via l'utilisation d'un pipe, se présenter à nous sous la forme de **10/07/2022**. De même, des valeurs numériques pourraient se présenter sous la forme d'une valeur monétaire, ou des nombres se présenter sous la forme de chaînes de caractères.

Les pipes s'utilisent directement dans le template du composant, via l'utilisation de l'opérateur pipe « | » entre la valeur à transformer et le pipe que l'on souhaite lui appliquer. Ainsi, la transformation d'une date sous un format plus adapté à nos besoins pourrait se présenter de la sorte :

```
<p>Bonjour à tous, nous sommes le {{ dateOfDay | date }}</p>
```

Il est également possible d'y ajouter des paramètres, de sorte à offrir encore plus de personnalisation à notre pipe. Pour cela, il suffit d'ajouter le caractère « : » suivi des paramètres (ici une chaîne de caractères) du pipe. Ainsi, si l'on ne veut que la date au format américain, on peut demander un pipe de la sorte :

```
<p>Bonjour à tous, nous sommes le {{ dateOfDay | date:"MM/dd/yyyy" }}</p>
```

Les Différents Pipes du Framework

Angular met à notre disposition de nombreux pipes de base dans le framework :

- **date** : Ce pipe sert à mettre en forme une date et il est possible d'en personnaliser le format de sortie grâce à ses paramètres.
- **async** : Ce pipe prend en entrée une promesse ou un observable et en affiche le résultat.
- **uppercase** : Ce pipe permet de transformer un texte en une version composée uniquement de lettres capitales.
- **lowercase** : Ce pipe fait l'inverse du précédent, et donne un résultat fait uniquement de lettres minuscules.
- **json** : Il transforme l'objet que l'on lui envoie en une représentation sous la forme d'une chaîne de caractères.
- **decimal** : Il permet de mettre en forme une variable numérique. Il possède un paramètre permettant de spécifier la partie entière et la partie décimale (nombre de chiffres après la virgule).
- **percent** : Va afficher la valeur numérique sous la forme d'un pourcentage.
- **currency** : Il va afficher les nombres dans un format monétaire (on peut spécifier la monnaie via le paramètre)
- **i18nSelect** : Permet l'affichage d'un texte en fonction d'une clé (Il faut un paramètre de tableau d'équivalence clé-valeur)
- **i18nPlural** : Permet la même chose que le précédent mais sur plusieurs éléments
- **replace** : Permet le remplacement de parties de chaînes de caractères par d'autres
- **slice** : Permet de couper les chaînes de caractères à des index de lettres données.

Les Pipes de Transformation

Il est également possible de créer nos propres pipes pour transformer les données en fonction de nos besoins. Pour ce faire, il nous faut créer une classe implémentant l'interface `PipeTransform`, qui ne possède qu'une seule méthode :

`transform(value: any, ...args: any) : any`

Le premier paramètre est la valeur sur laquelle sera appliqué l'opérateur de pipe, tandis que les suivants seront ceux passés en paramètre lors de l'appel du pipe. Il sera nécessaire de les séparer par des doubles points. Le retour de cette méthode sera utilisé lors du binding du pipe.

Il nous faudra également ajouter le décorateur **@Pipe** à notre classe, et y spécifier le nom de notre pipe. Par exemple, on pourrait avoir ce pipe :

```
@Pipe({ name: "monPipe"})
export class MonPipe implements PipeTransform {
  transform(value: string, isMale: boolean = true) : string {
    if (isMale) return "Bonjour monsieur " + value;
    return "Bonjour madame " + value;
  }
}
```

Pour utiliser ensuite ce pipe dans un composant, il est nécessaire de l'ajouter aux déclarations de notre module :

```
@NgModule({
  declarations: [
    AppComponent,
    HomeComponent,
    MonPipe
  ],
})
```


Les Pipes de Filtre

La signature de la méthode **transform()** nous permet de retourner n'importe quelle valeur. Les pipes peuvent ainsi être utilisées dans le but de filtrer des valeurs, ce que l'on pourrait imaginer être associé à la directive structurante ***ngFor**. Pour ce faire, il suffit que notre pipe prenne en paramètre un tableau et en retourne un nouveau (filtré cette fois-ci donc).

```
@Pipe({ name: "monPipe"})
export class MonPipe implements PipeTransform {
  transform(value: Personne[]) : Personne [] {
    return value.filter(x => x.isMale);
  }
}
```

On se sert ici de la méthode **.filter()** disponible en Javascript de par la classe **Array**. Une fois le pipe construit, il ne nous reste plus qu'à l'utiliser dans un élément HTML. On peut ainsi obtenir facilement des tris différents de nos données en appliquant un pipe ou un autre en fonction des choix utilisateurs :

```
<tbody>
  <tr *ngFor="let owner of owners | monPipe; let i = index;">
    <td>{{ i }}</td>
    <td>{{ owner.name }}</td>
  </tr>
</tbody>
```

Les Pipes Purs et Impurs

Angular différencie deux grands types de pipes : Les pipes dits « purs » et les pipes dits « impurs ». La différence entre les deux se situe dans leur façon de rafraichir les données, c'est-à-dire lorsqu'Angular relance le pipe dans le but de modifier l'interface utilisateur.

Un pipe pur ne se réexécute que lorsque Angular détecte un changement sur une valeur de type primitive ou lors d'un changement de référence pour les types objets. Il ne se réexécutera donc pas lorsqu'un élément sera ajouté à un tableau (la référence de ce dernier ne changeant alors pas). A contrario, un pipe impur se réexécutera à chaque changement.

Un pipe pur est plus performant par nature, et il vaut mieux si possible éviter d'utiliser des pipes impurs. Dans le cas d'un filtre, le pipe est facilement remplaçable par une propriété dans le composant, celle-ci se voyant être mise à jour manuellement lorsque les filtres ou les données changent.

Si l'on veut créer malgré tout un pipe impur, il nous faudra ajouter la métadonnée **pure** et lui donner comme valeur **false**. On remarque donc que par défaut, tous les pipes de notre application sont des pipes purs. Angular fonctionne ainsi plus rapidement, en évitant les rafraichissements inutiles.

```
@Pipe({
  name: "monPipe",
  pure: false
})
export class MonPipe implements PipeTransform {
  transform(value: Personne[]) : Personne [] {
```

Exercice : camelPipe et trimPipe

Objectif

Appréhender la manipulation et l'utilisation des pipes dans le framework Angular

Sujet

En vous servant du framework Angular, vous devrez réaliser deux pipes :

- Le **premier pipe** aura pour but la transformation du texte en une version camelCase
- Le **second pipe** fera en sorte que le texte ne pourra pas dépasser une longueur maximale de 10 caractères sous peine d'être coupé et suivi d'un ensemble de points de suspension.

Une fois fait, vos pipe devront être **testés** sur un ensemble de données qui seront affichées, par exemple sous la forme d'un tableau ou d'une liste, afin d'en vérifier le fonctionnement. Vous pourrez par exemple tenter d'utiliser l'un des deux pipes sur une ligne du tableau, ou les **deux à la fois** sur une autre ligne.

Exercice : enumPipe et sortingPipe

Objectif

Appréhender la manipulation et l'utilisation des pipes dans le framework Angular

Sujet

En vous servant du framework Angular, vous devrez réaliser deux pipes : un pipe de transformation de valeur et un pipe de tri d'un tableau.

Dans une application vous permettant d'avoir un descriptif de plusieurs serveurs (composés d'un **nom**, d'une **taille**, d'un **statut**, et d'une **date de mise en service**). Ces descriptifs de serveur devront avoir un style personnalisé pour avoir un visuel harmonieux et plaisant.

Le pipe de transformation aura pour but d'offrir à l'utilisateur la description en toute lettre de la taille du serveur en fonction de sa taille en TB. Par exemple, un serveur de 1TB sera considéré comme petit, un serveur de 2TB comme moyen, etc...

Le pipe de tri aura pour objectif de permettre l'affichage des serveurs triés par leur ancienneté, de manière à avoir le plus ancien en premier et le plus récent en dernier.

Les Composants

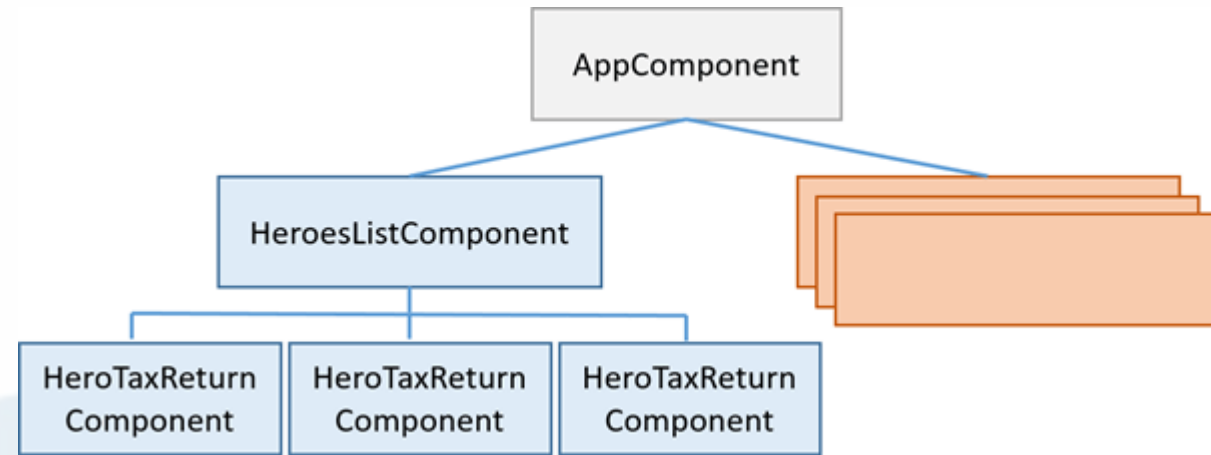
The bottom of the slide features several overlapping, wavy lines in shades of light blue and grey, creating a modern, fluid background element.

L'Importance des Composants

Les composants sont l'une des composantes essentielles d'Angular. L'utilisation de multiples composants au lieu d'un composant contenant énormément de code est l'une des qualités que doit posséder un développeur Front-End Angular. De plus, l'utilisation de plusieurs composants, le développeur fournit à d'autres potentiels collaborateurs le moyen de travailler en symbiose et de ne récupérer que les fonctionnalités qu'ils leur faut.

Lorsque l'on conçoit une application web avec Angular, il ne faut ainsi pas hésiter à multiplier les composants, et à en créer des sous-ensembles lorsque l'on voit que notre code se répète ou qu'il commence à être trop long.

Bien entendu, il faut également faire attention à nos données (nos variables), car la séparation en de multiples composants va nous pousser à réfléchir à où doivent se trouver des types tableaux par exemple. De même, les entrées utilisateurs devront se trouver à l'endroit adéquat pour accéder facilement à la capacité d'ajouter ou de modifier ces tableaux.



Angular nous fournit bien évidemment la capacité de faire communiquer nos composants, permettant ainsi de placer un bouton de suppression sur chacun de nos composants enfants afin de les faire disparaître au besoin en cas de modification des tableaux du parent.

Rappels sur le Binding

Précédemment, nous avons vu comment nous servir de la liaison de donnée pour permettre à notre code et à notre template de communiquer. On a ainsi pu observer qu'il existait plusieurs types de bindings, tels que :

- Le One-Way Data Binding
 - String Interpolation : La capacité d'afficher du texte en fonction d'une valeur de notre code Typescript
 - Property Binding : La capacité d'altérer une propriété d'élément HTML à partir de notre code Typescript
 - L'Event Binding : La capacité de nos éléments HTML à déclencher des méthodes de notre code Typescript
- Le Two-Way Data Binding : La capacité de nos éléments HTML à rafraichir les données TS et à suivre les modification de ces valeurs afin de se rafraichir automatiquement

Les liaisons de données sont disponibles lors de la création d'une application Angular à plusieurs niveaux (nous en avons déjà explorés certains) :

- **HTML** : Au niveau de l'HTML, avec par exemple la capacité de modifier l'apparence d'un bouton en fonction de nos données, ou la capacité à remplir les cellules d'un tableau par les propriétés d'un objet Typescript.
- **Directives** : Au niveau de nos directives, comme par exemple le fait de modifier les classes de notre élément HTML en fonction de notre code, ou la capacité de synchroniser les formulaires avec nos données par utilisation d'un lien sur la directive **[(ngModel)]**
- **Composants** : Il est également possible de nous lier directement à nos composants afin de les alimenter en données.

Se lier à nos propriétés

Pour commencer, voyons comment faire en sorte d'alimenter en données les enfants de notre composant. Pour ce faire, il va nous falloir exposer à l'extérieur du composant des propriétés personnalisées qui pourront être liées au niveau du parent. Par défaut, les propriétés déclarées dans un composant ne sont accessibles qu'à ce composant, mais il est possible d'altérer ce processus afin de pouvoir, au niveau d'un parent, profiter d'une syntaxe de ce type :

Pour ce faire, il existe deux variantes, toutes deux reposant sur l'utilisation du décorateur **@Input()** au niveau de notre propriété :

```
@Input()  
element: Dog | undefined;
```

```
<app-dog *ngFor="let dog of dogs;" [element]="dog"></app-dog>
```

Ce décorateur peut être altéré pour modifier le nom de la propriété extériorisée (l'**alias**) de notre composant (on pourrait vouloir garder le nom de « **element** » à l'intérieur du composant, mais vouloir utiliser un nom différent pour le binding, par exemple « **dog** »). Pour ce faire, il suffit d'ajouter une chaîne de caractère dans les paramètres du décorateur :

```
@Input('dog')  
element: Dog | undefined;
```

```
<app-dog *ngFor="let dog of dogs;" [dog]="dog"></app-dog>
```

Se lier à nos événements

Bien entendu, la réciproque de la méthode vue précédemment existe également. Il nous est possible de nous lier aux événements déclenchés par nos composants enfants dans le but d'altérer les composants parents. Pour ce faire, nous allons devoir écrire cependant un peu plus de code, et créer des méthodes personnalisés. Via l'utilisation de ce processus, il nous sera possible d'obtenir un lien sur l'évènement au niveau du parent comme dans l'exemple ci-dessous :

```
<app-dog *ngFor="let dog of dogs;" (dogCreated)="onDogCreated($event)"></app-dog>
```

```
onDogCreated(dogData: Dog) {  
  this.dogs.push(dogData);  
}
```

Au niveau de l'enfant, c'est cette fois-ci le décorateur **@Output()** (qui peut également porter un alias dans ses paramètres) qu'il va nous falloir utiliser. Ce décorateur devra se trouver devant des propriétés contenant des **EventEmitter<T>** (EventEmitter étant un générique, il nous faut lui donner un type, par exemple **EventEmitter<Dog>**). Une fois fait, il nous faudra créer dans le composant enfant une méthode permettant d'émettre cet évènement. Cette méthode, ainsi que la propriété, se présentent de la sorte :

Il faut bien faire attention à émettre le type spécifié entre les chevrons dans notre EventEmitter (la généricité doit être respectée pour éviter les problèmes lors de la compilation).

```
@Output('gogCreated')  
newDog = new EventEmitter<Dog>();  
  
onDogCreated() {  
  this.newDog.emit({  
    name: this.dogName,  
    breed: this.dogBreed,  
    age: this.dogAge,  
    isMale: this.dogIsMale  
  });  
}
```

TP : TodoList App

@Utopios Consulting

Objectif

Maîtriser les bases du Databinding, de la création et de la communication des composants

Sujet

En vous servant du framework Angular, vous devrez réaliser une application permettant à un utilisateur de bénéficier d'une liste de tâches.

L'utilisateur doit pouvoir ajouter, modifier et supprimer les tâches et se voir présenter une liste des tâches comprenant leur titre, leur description, leur état de complétion (par exemple « En cours » et « A faire ») ainsi que des boutons pour interagir avec chacune des tâches.

Cette application se servira de plusieurs composants : un composant servant à l'ajout des tâches, et un composant servant au display des tâches et à leur manipulation.

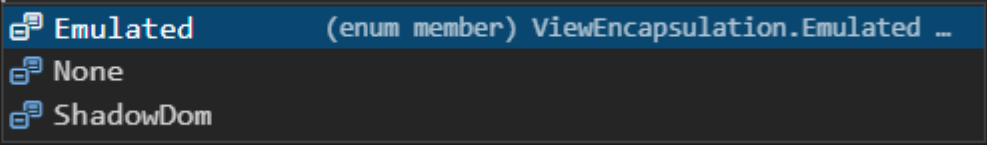
La View Encapsulation

La **View Encapsulation** est la capacité d'Angular d'isoler nos différents composants pour leur faire ou non posséder des propriétés personnelles. Par exemple, grâce à la View Encapsulation, il nous est possible de déclarer des styles pour nos composants et être sûr de ne pas les voir s'appliquer dans d'autres composants, et ce malgré que l'on ait ciblé des éléments HTML communs aux deux.

Ce comportement provient du fait que par défaut, Angular ajoute à nos éléments HTML compris dans un composant des attributs de type **ng-content-xxxx**. Ces attributs étant différent pour chaque composant, on a ainsi un ciblage CSS dépendant du composant dans lequel les valeurs de notre style sont écrites.

Il est bien entendu possible de modifier cette encapsulation, via la modification d'une propriété dans notre décorateur **@Component** :

```
encapsulation: ViewEncapsulation.
})
export class DogComponent implements
@Input()
```



- **Emulated** : C'est la valeur par défaut de notre encapsulation
- **None** : Retire tout simplement l'encapsulation (ce qui permet par exemple de redonner sa globalité au CSS)
- **ShadowDom** : Possède la même fonctionnement qu'Emulated, mais uniquement dans les navigateur supportant cette fonctionnalité

Les Références Locales

Pour récupérer les données de notre template, il est possible d'utiliser le Data Binding. Malgré tout, si notre but est de ne récupérer des valeurs qu'à un moment donné (le clic d'un bouton d'envoi par exemple), alors la liaison de donnée est trop poussée pour nos besoins. Pour combler à ce problème, il est possible de se servir de références locales. Il est possible de placer des références locales sur n'importe quel élément HTML de nos composants. Pour ce faire, il suffit d'utiliser la syntaxe **#nomRéférence**. Cette référence est en lien avec l'élément, et non sa valeur comme cela pouvait être le cas lors de l'utilisation de [(ngModel)].

```
<input type="text" class="form-control"
name="dog-name" id="dog-name" placeholder="Bernie"
#dogName>
```

Cette référence locale peut être directement utilisée dans notre HTML, comme par exemple pour envoyer des données via un bouton :

```
<button class="btn btn-danger" (click)="onDeleteDog(dogName)">
  <i class="bi bi-trash"></i> Supprimer un chien
</button>
```

Il faudra bien entendu traiter les composantes de notre élément HTML dans notre méthode au niveau du fichier de code Typescript.

```
onDeleteDog(input: HTMLInputElement) {
  this.dogDeletedMessage = input.value + " was deleted...Poor Soul...";
}
```

Il est également possible d'avoir accès à toute la template HTML directement depuis notre fichier de code Typescript. Pour ce faire, il va nous falloir utiliser le décorateur **@ViewChild()**. Ce décorateur doit être importé de **@angular/core** pour fonctionner.

Pour nous servir de ce décorateur, on peut créer une propriété dans notre classe de composant que l'on précède du décorateur. Dans les paramètres du décorateur, il nous faudra mettre l'élément que l'on souhaite sélectionner, et un paramètre déterminant si cette propriété est statique ou non. La syntaxe pourrait ainsi ressembler à quelque chose comme ceci :

```
@ViewChild("dogName", {static: true}) myDogName! : ElementRef;
```

Contrairement aux références locales vues précédemment, nous ne passons ici plus directement l'élément HTML mais une référence vers ce dernier. Ce changement nous force ainsi à modifier légèrement notre code métier si l'on souhaite récupérer les valeurs de l'élément HTML :

```
this.myDogName = this.myDogNameElement.nativeElement.value;
```

Attention, quand bien même il est possible d'accéder au DOM dans le but de le manipuler (changer par exemple la valeur de notre input dans cet exemple), il est fortement déconseillé de le faire de la sorte. Pour cela, on aurait tendance à privilégier l'utilisation du Data Binding (String Interpolation / Property Binding), par exemple.

Ajouter du contenu avec ng-content

Par défaut, Angular ne se soucie pas de ce que l'on pourrait avoir placé entre les deux balises délimitant notre sélecteur. Ainsi, n'importe quel code HTML se trouvant par exemple entre `<app-dog>` et `</app-dog>` sera ignoré. Il y a néanmoins une possibilité que l'on veuille y placer du code, et ainsi remplir notre composant dynamiquement depuis le parent. Pour ce faire, il va nous falloir nous servir de **ng-content**

```
<app-dog *ngFor="let dog of dogs; let i=index;">
  <tr>
    <td scope="row"> {{ i }}</td>
    <td> {{ dog.name }}</td>
    <td> {{ dog.breed }}</td>
    <td> {{ dog.isMale }}</td>
    <td> {{ dog.age }}</td>
  </tr>
</app-dog>
```

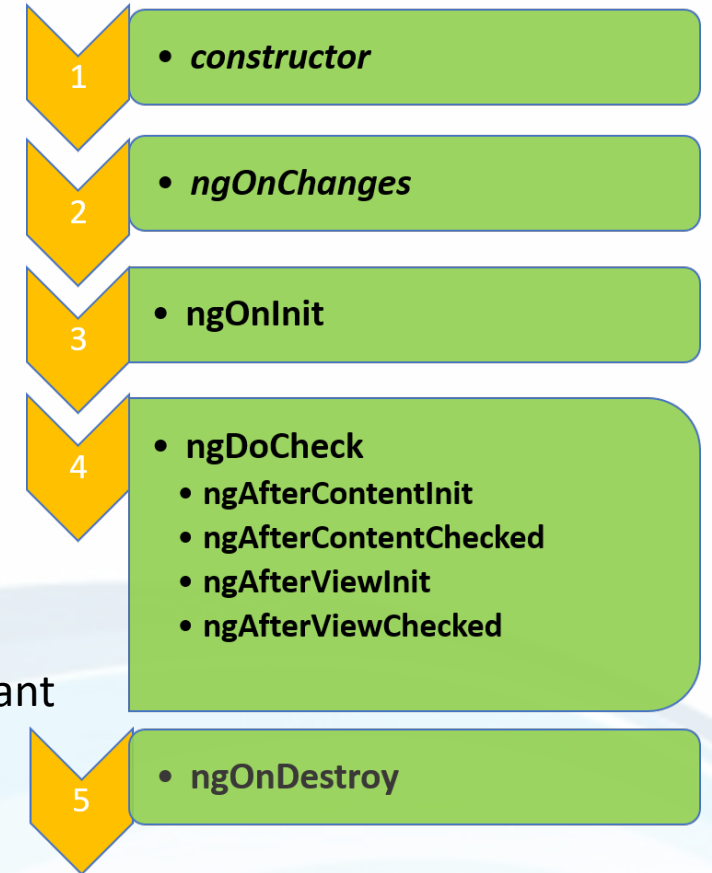
Son utilisation est très simple, il suffit simplement, au niveau de notre enfant, de placer une balise `<ng-content>` `</ng-content>`, comme dans l'exemple ci-dessous :

```
Go to component
<hr>
<ng-content></ng-content>
<hr>
```


Les Cycles de Vie des Composants

Les composants d'Angular passent par plusieurs étapes au court de leur existence. La connaissance de ces différents moment nommés « cycle de vie ». De plus, la compréhension de ces cycles de vie permet d'éviter des erreurs potentielles, comme par exemple l'affichage d'une valeur alors qu'elle n'est pas encore présente au niveau de notre composant, causant par exemple l'apparition en console d'une valeur **undefined**. En nous liant à ces cycles de vie, via l'utilisation des méthodes ci-dessous, nous pouvons exécuter du code au moment désiré lors de la modification des composants par Angular :

- **ngOnChanges** : Sera exécutée à chaque changement de propriété dans le composant
- **ngOnInit** : Sera exécutée lors de l'initialisation du composant
- **ngDoCheck** : Sera exécutée à chaque vérification du contenu par Angular
- **ngAfterContentInit** : Sera exécutée après la projection de **ng-content** dans le composant
- **ngAfterContentChecked** : Sera exécutée à chaque vérification de **ng-content**
- **ngAfterViewInit** : Sera exécutée après l'initialisation de la vue du composant
- **ngAfterViewChecked** : Sera exécutée après la vérification de la vue par Angular
- **ngOnDestroy** : Sera exécutée si jamais on supprime le composant du DOM



La connaissance des cycles de vie nous apprend par exemple que le contenu de notre DOM (par exemple le texte contenu dans un ****) n'est pas disponible à la méthode **ngOnInit()**, mais qu'il l'est à partir de **ngAfterViewInit()**

@ContentChild()

Utopios Consulting

Si l'on souhaite accéder à des références locales depuis un enfant pour des éléments projeté par le parent via l'utilisation de **ng-content**, **@ViewChild()** ne fonctionnera pas. Cela fonctionnerait pour notre parent, mais pas l'enfant. En effet, le contenu projeté dans un enfant par l'utilisation des balises **<ng-content> </ng-content>** ne fait pas partie de la vue, mais du contenu. La différence se fait également sentir lorsque l'on cherche à accéder à ces valeurs dans les méthodes d'accroche des cycles de vie de notre composant (**ngAfterViewInit** est différent de **ngAfterContentInit**).

```
<app-dog *ngFor="let dog of dogs; let i=index;">
  <tr>
    <td scope="row"> {{ i }}</td>
    <td #dogName> {{ dog.name }}</td>
    <td> {{ dog.breed }}</td>
    <td> {{ dog.isMale }}</td>
    <td> {{ dog.age }}</td>
  </tr>
</app-dog>
```

Pour remédier à notre problème, Angular a introduit le décorateur **@ContentChild()**. Sa syntaxe est similaire au décorateur précédent, et dans le cas d'une référence locale, on aurait ainsi comme décorateur :

```
@ContentChild("dogName", {static: true}) dogNameTableData!: ElementRef;
```

Le fonctionnement est globalement le même que pour **@ViewChild()**, à ceci près que la valeur contenu dans un élément ne le sera cette fois ci pas avant la méthode **ngAfterContentInit** (au lieu de **ngAfterViewInit** comme c'était le cas pour **@ViewChild()**)

Exercice : Compteur pairs / impairs

Objectif

Appréhender la manipulation et l'utilisation du Data Binding entre composants dans le framework Angular

Sujet

En vous servant du framework Angular, vous devrez réaliser plusieurs composants :

- Un composant servant aux **contrôles** du compteur de nombres
- Un composant servant à afficher les **nombres pairs**
- Un composant servant à afficher les **nombres impairs**

En regroupant des 3 composants dans votre composant racine (**AppComponent**), vous devrez réaliser une application permettant de **démarrer**, **mettre en pause** et **redémarrer** un compteur de nombres. Les boutons permettant le contrôle de l'application se trouveront dans le premier des trois composants listés ci-dessus.

Ces nombres seront générés **chaque seconde**, et s'afficheront, en fonction de leur type, dans le composant 2 ou le composant 3, par exemple sous la forme « **EVEN - 24** ».

Lors de la mise en **pause** de l'application, plus aucun nombre ne se verra être ajouté à l'interface. La remise en route de l'application par le bouton de **démarrage** redémarrera le comptage. Le bouton permettant de **relancer** le compteur réinitialisera l'interface en supprimant les nombres déjà affichés et **demandera le clic sur le bouton de lancement pour recommencer**.

Les Directives

The bottom of the slide features several overlapping, wavy lines in shades of light blue and grey, creating a modern, fluid background element.

Rappels sur les Directives

Les directives sont utilisées par Angular pour modifier nos templates HTML plus facilement. Il en existe deux types majeurs : les directives d'attributs, et les directives structurelles.

Nous avons déjà pu nous essayer à la manipulation des directives d'attributs suivante :

- **ngClass** : Il nous est possible de conditionner la présence de classes en passant un objet contenant en clé les classes et en valeur les condition pour chacune d'entre elles
- **ngStyle** : Il est possible de modifier les valeurs de propriétés CSS en se servant d'un objet possédant des clés rappelant les propriétés CSS (sous la forme de CSS ou de JS) et en valeur des ternaire remplissant ces propriétés par des valeurs.

Pour rappel, nous avons également vu deux directives structurelles extrêmement utilisées dans une application web réalisée avec Angular : ***ngIf** et ***ngFor**.

- Grâce à ***ngIf**, il nous est possible d'afficher ou de masquer des éléments de notre DOM ou même de nos propres composants en fonction des variables présentent dans notre code métier Typescript
- Grâce à ***ngFor**, il nous est possible de générer l'affichage dynamique de plusieurs composants ou de plusieurs éléments en fonction de nombre d'éléments présents dans une variable de type conteneur (un tableau en Javascript).

Dans les deux cas l'affichage de nos éléments sera automatiquement mis à jour en cas de modification de nos valeurs dans le code métier. Il est important de savoir également qu'il n'est pas possible de cumuler les directives de type structurelles dans un même élément. On ne peut ainsi pas faire à la fois une itération et une condition pour générer l'affichage de nos composants. Si l'on souhaite par exemple filtrer un tableau, la solution la plus évidente serait l'utilisation d'un pipe.

Créer une directive d'attribut

Pour créer une directive d'attribut, il nous suffit encore une fois de simplement créer une classe destinée à l'export et de cette fois-ci la faire précéder du décorateur **@Directive()**. Ce décorateur sera alimenté par un objet qui contiendra un sélecteur de type attribut. Pour ce faire, il nous suffit de respecter la syntaxe suivante : **[nomDeNotreAttribut]**

Pour modifier l'élément sur lequel est appliqué la directive, il nous faut y accéder. Pour cela, Angular nous permet facilement de profiter de son injection. Pour l'injecter, il nous faut déclarer une propriété dans notre classe et lui placer l'injection depuis notre constructeur. Cette injection sera une variable de type **ElementRef**, et pourrait ainsi se présenter comme ci-dessous :

```
@Directive({
  selector: '[appDirectiveDemo]'
})
export class DirectiveDemoDirective implements OnInit {

  constructor(private elementRef: ElementRef) { }

  ngOnInit() {
    this.elementRef.nativeElement.style.fontSize = "2rem";
  }
}
```

Pour la faire fonctionner et la tester, il ne nous reste plus qu'à l'ajouter à la liste des déclarations de notre module (Attention, lors de l'ajout aux déclarations, il ne faut pas oublier l'import !) et d'ajouter l'attribut à un élément HTML pouvant être la cible d'un tel changement de style.

Utiliser le Renderer

Comme spécifié précédemment, l'utilisation et la modification des éléments de notre HTML en passant par l'ElementRef de la sorte n'est pas conseillé. Pour modifier notre DOM de la sorte, une meilleure approche est celle de l'utilisation du Renderer.

Pour ce faire, il nous faut modifier notre injection, et donc obtenir un constructeur n'injectant plus uniquement une variable de type **ElementRef**, mais également **Renderer2**. On obtient donc une injection ressemblant à ceci :

```
constructor(private element: ElementRef, private renderer: Renderer2) { }
```

Via l'utilisation du Renderer, il est possible de modifier notre élément par ses méthodes, comme par exemple la méthode **.setStyle()**, qui prend en paramètre l'élément à modifier, la propriété CSS à modifier, la valeur que l'on souhaite lui donner, et de façon optionnelle des flags comme par exemple **!important**) Par exemple, pour reprendre notre exemple de directive précédent, on pourrait avoir une méthode exécutée lors de l'initialisation via l'instruction suivante :

```
ngOnInit() {  
  this.renderer.setStyle(this.element.nativeElement, "font-size", "2rem");  
}
```

Cette approche est supérieure à la précédente car le Renderer permet la modification des propriétés malgré l'absence de DOM (comme c'est le cas des applications ne tournant pas dans un navigateur). Par l'utilisation du Renderer, on s'assure donc que nos modifications d'élément peut s'effectuer peu importe l'environnement de lancement de notre application.

@HostListener()

@Utopios Consulting

A l'heure actuelle, il n'est pas possible de modifier les éléments de notre HTML en fonction des événements émis par notre DOM. Ceci est amené à changer avec l'utilisation de **HostListener**.

Pour utiliser HostListener, il suffit de placer le décorateur **@HostListener** devant une méthode que l'on souhaite voir s'exécuter lors du déclenchement d'un événement donné (cet événement étant donné au décorateur en paramètre sous la forme d'une chaîne de caractère). Il est également possible de récupérer les données soulevées par l'événement en ajoutant un paramètre de type Event à la méthode décorée.

Par exemple, pour obtenir une directive réactive au passage de la souris sur un élément du DOM, on pourrait avoir deux méthodes comme ci-dessous :

```
@HostListener("mouseenter") mouseHover (event: Event) {  
    this.renderer.setStyle(this.element.nativeElement, "font-size", "2rem");  
}  
  
@HostListener("mouseleave") mouseLeave (event: Event) {  
    this.renderer.setStyle(this.element.nativeElement, "font-size", "1rem");  
}
```

@HostBinding

Utopios Consulting

Si l'on veut se lier à notre template dans le but de l'altérer sans avoir à utiliser le `Renderer`, il existe une autre possibilité. Celle-ci consiste en l'utilisation d'un autre décorateur : **@HostBinding()** que l'on va lier à une propriété, par exemple **fontSize**, qui sera de type **string**. Dans le paramètre du décorateur, il va nous falloir spécifier à quelle propriété de notre élément nous souhaitons relier cette propriété TS. Pour continuer dans l'exemple précédent, on aurait ici un décorateur ressemblant à ceci :

```
@HostBinding("style.fontSize") fontSize: string = "1rem";
```

Il nous faut désormais faire attention à posséder une valeur par défaut, sous peine d'avoir un problème de logique au moment de la construction de notre élément HTML. Une fois fait, il ne nous reste plus qu'à modifier nos méthodes précédentes pour ne plus utiliser le `Renderer` mais désormais utiliser notre binding. Une fois fait, nous pourrions avoir par exemple les deux méthodes suivantes :

```
@HostListener("mouseenter") mouseHover (event: Event) {  
    this.fontSize = "2rem";  
}  
  
@HostListener("mouseleave") mouseLeave (event: Event) {  
    this.fontSize = "1rem";  
}
```

Paramètres de Directive

Si l'on décide désormais de permettre à notre directive de prendre en paramètre les valeurs qui seront modifiées par elle, alors il nous faut modifier encore une fois notre code. On pourrait ainsi imaginer qu'un développeur utilisant notre directive ne veuille pas augmenter la taille du texte à 2rem, mais à une valeur qu'il aurait lui-même choisie.

Pour ce faire, on pourrait par exemple avoir deux propriétés auxquelles nous lier. La première serait la valeur par défaut, et l'autre la valeur en cas de survol de notre élément par le curseur. Ces deux propriétés seront décorées par un décorateur

@Input()

```
@Input() defaultSize: string = "1rem";  
@Input() changedSize: string = "2rem";
```

directive.ts

```
<p appDirectiveDemo [defaultSize]='0.5rem' [changedSize]='1.5rem'></p>
```

Template HTML

Si l'on travaille avec une seule propriété, il est possible d'avoir la même syntaxe que les directives **ngClass** et **ngStyle** en ajoutant le nom du sélecteur de la directive en tant qu'alias de notre propriété :

```
@Input('appDirectiveDemo') changedSize: string = "2rem";
```

```
<p [appDirectiveDemo]='2rem' [defaultSize]='0.5rem'></p>
```

Il est intéressant de savoir que lors de l'utilisation de valeurs de type **string**, il est possible d'omettre les crochets et les guillemets simple, de sorte à avoir une syntaxe allégée :

```
<p [appDirectiveDemo]='2rem' defaultSize=0.5rem></p>
```

Directive structurelle

Il est intéressant de savoir que l'utilisation de l'astérisque dans les directives structurelle n'est qu'un moyen simplifié de les écrire qui nous est offert par Angular. De base, les propriétés structurelles de type ***ngIf** sont transformées en une structure entouré des balises **<ng-template [ngIf]="condition"> </ng-template>**. On peut ainsi observer que ces directives se servent du **Property Binding** classique pour fonctionner une fois mises sur un élément de type ng-template.

```
<div *ngIf="myDogBorn">
  <div class="col-12 col-md-4">
    
  </div>
  <div class="col-12 col-md-8">
    <p>Félicitation ! Un chien est né !</p>
  </div>
</div>
```

La syntaxe ci-dessus est par exemple transformée par Angular en la syntaxe suivante et lui est équivalente :

```
<ng-template [ngIf]="myDogBorn">
  <div>
    <div class="col-12 col-md-4">
      
    </div>
    <div class="col-12 col-md-8">
      <p>Félicitation ! Un chien est né !</p>
    </div>
  </div>
</ng-template>
```

Créer une Directive structurelle

Si l'on souhaite créer une directive structurelle, il nous faut accéder par l'injection dans le constructeur à la référence du template ainsi qu'à la référence du conteneur de vue. Une fois fait, il va nous falloir lier une propriété à notre directive, de sorte à profiter du Property Binding vu précédemment. Ici, la propriété pourrait posséder le même nom que la directive, ce qui nous permettra de nous passer d'un alias en paramètre du décorateur **@Input()**.

```
export class UnlessDirective {
  @Input() set appUnless (condition: boolean) {
    if (!condition) {
      this.vcRef.createEmbeddedView(this.templateRef);
    } else {
      this.vcRef.clear();
    }
  }

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

Dans cet exemple, nous cherchons à reproduire une directive nous offrant le fonctionnement inverse de la directive ***ngIf**. Il nous faut donc afficher le contenu de notre template seulement si la condition est fausse. On voit donc que lorsque c'est le cas, on affiche notre template dans le conteneur de vue via sa méthode **.createEmbeddedView()** dans laquelle on passe le template. Dans le cas où la condition est vraie, il nous suffit d'utiliser la méthode **.clear()** pour vider le DOM de notre conteneur de vue.

La Directive [ngSwitch]

Si l'on souhaite afficher des éléments ou des composants en fonction d'une valeur donnée, il nous est possible de le faire via la directive **[ngSwitch]** (il est intéressant de rappeler que ***ngIf** peut gérer un « else », mais ne nous propose pas de « else if »). Pour s'en servir, il nous faudra nous lier à une propriété qui contiendra la valeur que l'on souhaite évaluer, et mettre des ***ngSwitchCase** pour chacune de nos valeurs, ainsi qu'un ***ngSwitchDefault** pour permettre la génération d'un composant par défaut. On obtient donc une structure pouvant ressembler à celle-ci :

```
<div [ngSwitch]="value">
  <p *ngSwitchCase="1">La valeur vaut 1</p>
  <p *ngSwitchCase="2">La valeur vaut 2</p>
  <p *ngSwitchCase="3">La valeur vaut 3</p>
  <p *ngSwitchDefault>La valeur vaut quelque chose...</p>
</div>
```

Exercice : Sélecteur de contenu

Objectif

Appréhender la manipulation et l'utilisation des directives dans le framework Angular

Sujet

En vous servant du framework Angular, vous devrez réaliser plusieurs sections d'une page :

- La première section contiendra un **message d'accueil** sur la page
- La seconde offrira à l'utilisateur un **descriptif sommaire de produits** d'un magasin
- La troisième permettra d'avoir un bref **aperçu des locaux** de l'échoppe

Via l'utilisation de directives, vous ferez en sorte que seule l'une de ces trois sections soit affichée en fonction d'un choix réalisé par l'utilisateur. Ce choix pourra se faire par exemple via l'utilisation d'une entrée de type `<select></select>`. De plus, chacune de ces sections sera modifiée visuellement ou structurellement par une directive, permettant par exemple :

- Pour la **première section**, d'avoir le message d'accueil. Parmi celui-ci se cachera un mot-clé qui passera en gras et en couleur en cas de survol par la souris de la section
- Pour la **seconde section**, chaque produit devra être ajouté dans le tableau via l'utilisation d'une directive structurelle branchée sur un tableau de produits
- Pour la **troisième section**, les locaux ne seront affichées que si l'utilisateur a au préalable cliqué sur un élément se cachant parmi le texte descriptif de la première section.

Services et Injection de Dépendances

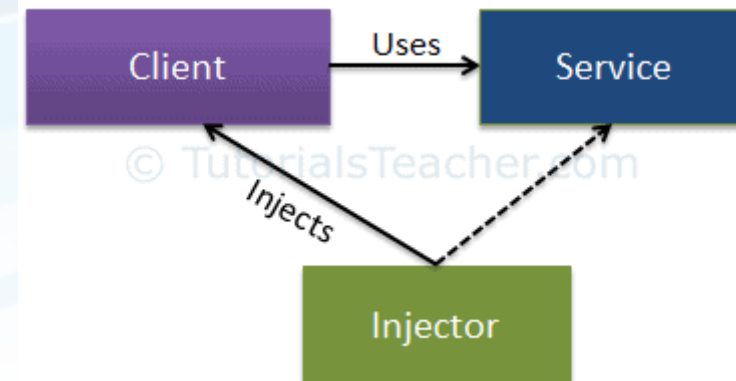


Qu'est-ce qu'un Service ?

Les services sont des éléments d'Angular nous permettant de traduire des fonctionnalités en un composants distinct : un service. Les cas typiques de l'utilisation des services sont les fonctionnalités liées au stockage des données ou les ensemble de fonctions qui sont utilisées à plusieurs endroits de notre application.

On pourrait par exemple imaginer l'utilisation d'un service de Logging dans le but d'obtenir toutes nos informations en console au fur et à mesure de notre développement et de l'avancement de notre application. Il est également possible d'avoir un service relié à une API pour recevoir et envoyer des données et pouvoir y accéder depuis chacun de nos composants usant ce service dans leurs classes. De cette façon, on peut donc centraliser notre code.

Bien d'autres cas d'utilisation des services existent, et il est nécessaire, lors de l'utilisation de services, de connaître le mécanisme de l'injection de dépendances pour bien comprendre le fonctionnement global de la chose. L'injection de dépendance provient du fait que ce n'est pas le développeur qui contrôle la création des composants, directives et autres éléments de notre application, mais bel et bien Angular. C'est au framework d'instancier nos classes pour former notre application. De par ce constat, il est nécessaire de réaliser ce qui s'appelle une « injection » lorsque l'on demande à une classe de se servir d'une autre classe. Pour ce faire, il nous faut paramétrer le « scope » de notre injection (est-elle globale à toute notre application ? Doit-elle être partagé par un composant et ses enfants ? Seulement par un composant en particulier ?). Grâce à ce paramétrage, il est possible d'avoir des données en commun dans notre application, ou de créer une nouvelle instance du service pour gérer une nouvelle paire de données si l'on le désire.



Créer un service de Logging

Le Logging consiste en l'envoi en console d'information utile pour un développeur lors de l'utilisation de son application. Grâce au logging, on peut récupérer des informations essentielles au débogage tout en conservant une interface homogène proche du client, sans perturber cet affichage par des **alert** ou des **** disgracieux lui montrant des valeurs qu'il ne pourrait pas décrypter. Pour suivre les conventions d'Angular, la création de service passe encore une fois par la création d'une classe, suffixée cette fois ci de l'extension **.service.ts**. Cette fois ci cependant, on ne se sert pas d'un décorateur comme par exemple **@Service()**. Une classe TS peut dès le début être utilisée en tant que service.

Pour un service de log, notre classe devrait par exemple posséder une méthode présentant la série d'instruction suivantes :

```
logToConsole(value: string) {  
  console.log(value);  
}
```

Une fois notre service rédigé, il ne nous reste plus qu'à l'injecter dans notre composant pour pouvoir accéder à toutes ses fonctionnalités. Attention, il est effectivement possible de créer une nouvelle instance de notre service et de nous servir de ses fonctionnalités. Malgré tout, ce mode de fonctionnement est à proscrire car il ne tire pas partie de l'injection des dépendances qu'Angular nous fournit. Pour nous en servir, il faut donc l'injecter depuis le constructeur :

```
constructor(private logService: LoggingService) {}  
  
onDogCreated(dogData: Dog) {  
  this.dogs.push(dogData);  
  this.logService.logToConsole(dogData.name + " was added to the dogs !")  
}
```

L'Injection de Dépendances

Comme expliqué précédemment, il est plus utile et plus optimal de déclarer les services en passant par l'injection des dépendances. Cette injection de dépendance nous est fournie par le framework Angular, et c'est le framework qui se chargera de l'ajouter à notre composant.

Grâce à l'injecteur de dépendance, il est possible de partager des services dans notre application, et de ne pas devoir à chaque composant instancier une nouvelle fois ce composant (ce procédé, en plus d'être coûteux, créerait à chaque fois un composant vierge de toute modifications, et donc, une pseudo-persistance des données serait impossible).

L'injecteur de dépendance d'Angular fonctionne donc premièrement par l'ajout de cette dépendance dans le constructeur, soit par la syntaxe de base, soit via l'utilisation de la syntaxe raccourcie :

```
private logService: LoggingService;

constructor(logService: LoggingService) {
  this.logService = logService;
}
```

```
constructor(private logService: LoggingService) {}
```

Ensuite, il nous faut inscrire dans notre décorateur de composant que l'on souhaite lui pourvoir le service, via l'ajout d'une propriété nommé **providers** et qui prend en valeur un tableau de classes que l'on souhaite voir injectées par Angular :

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

Créer un service de Données

L'autre grand type de service que l'on voit fréquemment dans une application web est sans nul doute le service de centralisation des données. Via l'utilisation d'un tel service, il ne sera plus nécessaire de faire transiter nos données de X parents à X enfants, il nous suffira de relier chaque composants à cet unique service pour que toutes les données se voient être commune à l'application.

Pour ce faire, il nous suffit de procéder encore une fois à l'injection via le constructeur et à l'utilisation de la propriété **providers** dans le décorateur de notre composant. Si l'on veut récupérer les données dans notre composant, il nous faut une propriété que l'on va alimenter via le Hook **ngOnInit()**, de sorte à obtenir une classe de ce type :

```
dogs: Dog[] = [];  
  
constructor(private logService: LoggingService, private dogsService: DogsService) { }  
  
ngOnInit(): void {  
  this.dogs = this.dogsService.dogs;  
}
```

Nous allons cependant rapidement nous rendre compte que cette façon de procéder, en injectant le même service via les **providers** dans tous les composants, ne va pas fonctionner. Nos données ne seront pas synchroniser dans toute l'application.

```
export class DogsService {  
  dogs: Dog[] = [  
    {  
      name: "Bernie",  
      breed: "German Shepard",  
      age: 2,  
      isMale: false  
    },  
    {  
      name: "Rex",  
      breed: "Doberman",  
      age: 7,  
      isMale: true  
    },  
    {  
      name: "Caramel",  
      breed: "Teckel",  
      age: 1,  
      isMale: true  
    }  
  ];  
  
  addDog(newDog: Dog) { ...  
  }  
  
  updateDog(id: number, newValues: Dog) { ...  
  }  
}
```

Gérer le Scope de nos Services

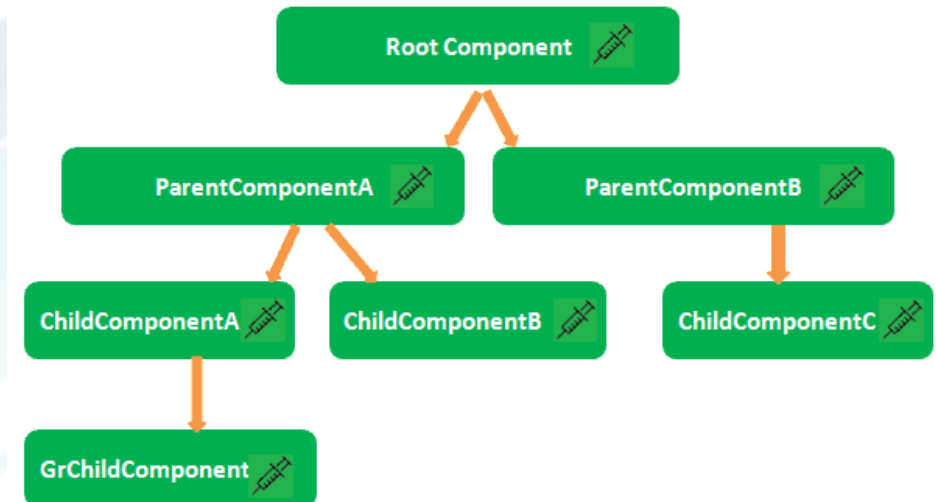
Nos services n'ont pas besoin d'être recréés à chaque fois que l'on souhaite les injecter. En les ajoutant dans la propriété **providers** de chaque composants parents et enfant, on informe en réalité Angular de procéder à cette création pour chaque élément de notre application. Le niveau hiérarchique de notre injection est important, car le fonctionnement de l'injecteur de dépendance du framework Angular fonction de façon à respecter une hiérarchie, de sorte à obtenir, de plus global à plus spécifique :

- **AppModule** : En injectant notre service à ce niveau, nos services seront disponibles dans toute l'application.
- **AppComponent** : En injectant dans ce composant, notre service est disponible dans tous les autres composants (mais pas les autres services)
- **Un composant** : L'instance de ce service sera disponible dans ce composant et dans tous ses enfants.

Il est donc intéressant de réfléchir à combien d'instance de service nous avons besoin lors de l'injection de nos services. Dans le cas d'un service de logging, ce détail ne modifie pas grandement son fonctionnement.

Par contre, si l'on a besoin d'utiliser un service pour transiter des données d'un composant à un autre, alors il nous faudra généralement l'injecter de façon plus globale, pour rendre les données communes à tous nos composants qui en dépendent.

ElementInjector Hierarchy



Des Services dans les Services

Il est également bien entendu possible d'utiliser des services dans d'autres services. On pourrait par exemple imaginer que lors de la modification de nos données, nous loguons les informations dans la console pour nous informer que les ajout ou les modifications ont bien été effectués. Ce **LoggingService** devra donc être injecté dans notre service de données, en passant notamment par l'utilisation du constructeur :

```
constructor (private logService: LoggingService) {}
```

Comme vu précédemment, il est également nécessaire pour réaliser une injection DANS un service d'avoir déclaré le service à injecté au niveau de notre module, ce qui le rend disponible dans toute l'application. Il nous faut donc ajouter, au niveau de notre module, le composant sans sa propriété de décorateur **providers** :

Une fois fait, on va cependant se heurter à un problème. Pour définir qu'un service va recevoir un autre service, il faut le déclarer comme étant injectable. Pour ce faire, il nous suffit simplement de lui ajouter le décorateur **@Injectable()**. Ce décorateur peut accepter comme paramètre un objet permettant de spécifier où sera injecté notre service. Via l'utilisation des services, il est donc possible de communiquer entre les composants. Un service pourrait par exemple posséder une propriété de type **EventEmitter<T>** qui serait émise par l'un des composants via la méthode **.emit()** et également souscrite au niveau de l'autre composant via la méthode **.subscribe()**. Dans les paramètres de la méthode se trouverait la fonction à réaliser.

```
UnlessDirective
],
imports: [
  BrowserModule,
  FormsModule
],
providers: [LoggingService],
bootstrap: [AppComponent]
})
export class AppModule { }
```

```
@Injectable()
export class DogsService {
  dogs: Dog[] = [
    {

```


Exercice : serviceClients

Objectif

Appréhender la manipulation et l'utilisation des services de Données dans le framework Angular

Sujet

En vous servant du framework Angular, vous devrez réaliser un service permettant d'effectuer un pseudo-CRUD en rapport avec des clients (ces clients auront comme composantes un **id**, un **nom**, un **service** et une **activité**). Ce service proposera les fonctionnalités suivantes :

- La récupération des clients actifs
- La récupération des clients inactifs
- La modification de l'activité d'un client
- La récupération d'un client par son ID

En branchant vos composants à ce service, vous devrez permettre à une application composées d'au moins deux composants (un composant pour les clients actifs et un pour les clients inactives) de permettre l'affichage sous la forme de deux listes stylisées les clients en fonction de leur activité.

En pressant sur un bouton contenu dans le composant correspondant, ce client « actif » devra alors passer d'actif à inactif, se déplacer dans la section correspondante, ou l'inverse en cas de statut « inactif » de base.

Le Routage

Qu'est-ce que le Routage ?

Le routage est un principe bien connu du web. Il est le processus permettant, dans une barre d'adresse, de spécifier une route nous amenant à une page de notre choix. Cette route peut également accepter des paramètres de requêtes, nous offrant ainsi un dynamisme et une communication entre un serveur et un navigateur.

Angular fonctionne par un système de page unique, et par relation de cause à effet, un routage n'est par conséquent pas possible. Cependant, il est possible de le simuler, via l'ajout d'une série de lignes de code et en général d'un autre module qui se consacrera à la gestion de ce routage.

Lorsque l'on crée une application Angular, il nous est d'ailleurs demandé si l'on veut se servir du routing. Dans le cas où l'on confirme ce choix, alors Angular nous fournira de lui-même un autre module que **AppModule** qui sera spécialisé dans la gestion du routage de notre application.

Créer du routing

Atopios Consulting

La création d'un routing en Angular passe par l'utilisation d'une variable de type **Routes**. Cette variable va prendre en valeur un tableau d'objets qui seront composés des propriétés **path** dans laquelle on renseignera une chaîne de caractère pour indiquer la route à suivre (cette route débutant sur notre machine à <http://localhost:4200/>) et d'une propriété **component** pour lui indiquer le composant à afficher à cette route.

```
const appRoutes: Routes = [  
  { path: "", component: HomeComponent },  
  { path: "dogs", component: DogComponent },  
];
```

Pour faire fonctionner nos routes, il nous faut également importer le module **RouterModule** et s'en servir pour attaquer sa méthode **.forRoot()** dans laquelle on passera notre constante de type **Routes**. Enfin, il nous faut donner un emplacement de sortie pour nos routes en utilisant la directive **router-outlet**, qui possède un sélecteur de type élément. La syntaxe sera donc **<router-outlet> </router-outlet>**

```
},  
imports: [  
  BrowserModule,  
  FormsModule,  
  RouterModule.forRoot(appRoutes),  
],  
providers: [LoggingService]
```

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

Accéder à nos routes

Pour l'accès à nos routes, il pourrait être tentant de se dire qu'il suffit de passer par l'utilisation de l'attribut **href** d'une balise **<a>**. Ce choix pourrait encore plus nous induire en erreur de par le fait que l'on atteindrait effectivement les routes que l'on souhaite. Malheureusement, même si nos routes sont ainsi atteintes, ce procédé nous impose un problème majeur : notre application rafraichit la page. Or, dans une application Angular, nous n'avons qu'une seule page, qui doit être modifiée par notre script Typescript. Le rafraichissement de la page cause ainsi une perte de tout état actuel de notre application, et donc, des données modifiées, ce qui n'est pas souhaitable.

Pour réaliser une redirection correcte avec Angular, il faut se passer de **href** et privilégier à la place l'utilisation de **routerLink**. L'utilisation de cet attribut de balise va permettre la création par Angular d'un lien qui sera géré différemment de notre attribut href. L'utilisation des chemins débutant avec un caractère « / » au début du chemin permet un chemin absolu, et son absence un chemin relatif au chemin actuel.

```
<ul>
  <li><a routerLink="/">Accueil</a></li>
  <li><a routerLink="/dogs">Chiens</a></li>
  <li><a [routerLink]="['/owners']">Propriétaires</a></li>
</ul>
```

Cette propriété peut être liée à une chaîne de caractère ou à un tableau, et nous offrir ainsi les deux syntaxes ci-dessus. L'utilisation de la chaîne de caractère est plus facile et est utilisée pour les chemins simples, mais offrira moins de personnalisation en cas de passage de multiples paramètres par la suite comme ça sera le cas dans des chemins complexes.

Le CSS et la classe « active »

Lorsque l'on veut styliser une application web, et en particulier les liens menant aux différentes pages, il nous faut utiliser une classe qui se nomme « **active** ». Cette classe, si l'on l'ajoute via l'attribut **class** de façon classique, ne marchera que difficilement dans une application Angular. A la place, le framework nous offre un moyen plus simple de gérer ce détail : la directive **routerLinkActive**, qui prend en paramètre les classes que l'on veut passer à l'élément HTML en cas de lien actif.

Ainsi, notre syntaxe précédente combiné à l'utilisation de **routerLinkActive**, dans le cas de l'utilisation de Bootstrap (qui nous demande d'ajouter cette classe au niveau de nos balises ****), donnerait le résultat suivant :

```
<ul>
  <li routerLinkActive="active"><a routerLink="/">Accueil</a></li>
  <li routerLinkActive="active"><a routerLink="/dogs">Chiens</a></li>
  <li routerLinkActive="active"><a [routerLink]="['/owners']">Propriétaires</a></li>
</ul>
```

Attention cependant. Par défaut, la directive **routerLinkActive** va regarder si notre chemin contient la variable ciblée, et en l'état actuel des choses, chacun de nos chemins contiennent le caractère « **/** », ce qui va rendre la présence de la classe « **active** » pour notre lien vers l'accueil commune pour toutes nos routes. Pour remédier à cela, il est possible de lier la propriété **routerLinkActiveOptions** et de lui passer un objet ayant comme propriété **exact** dans laquelle on place un **booléen** permettant de spécifier si l'on veut que le chemin soit égal à la variable au lieu de simplement la contenir.

```
<ul>
  <li routerLinkActive="active" [routerLinkActiveOptions]="{ exact: true }"><a routerLink="/">Accueil</a></li>
```

Naviguer depuis le code

On peut également forcer la navigation dans notre application depuis notre code, de sorte à permettre l'ajout de fonctionnalités implémentant cette mécanique dans leur fonctionnement. Pour ce faire, il nous suffit d'accéder au Router, que l'on peut injecter dans notre constructeur. Ce Router possède des méthodes, dont la méthode **.navigate()** prenant en paramètre un tableau représentant le chemin où chaque élément sera séparé d'un / à la création de la route.

```
onGetDogs() {  
  this.router.navigate(["/dogs"]);  
}
```

Cette méthode n'a pas conscience, contrairement aux directives **routerLink**, où l'on se trouve, ce qui fait que la création d'une route relative se passe autrement. Pour se faire, il faut ajouter en paramètre à la méthode un objet possédant la propriété **relativeTo**. Si l'on veut rendre notre route relative à notre emplacement, il est possible d'injecter dans le constructeur de notre classe **ActivatedRoute** pour obtenir cette information :

```
private router: Router, private activeRoute: ActivatedRoute) { }
```

Il nous suffit donc désormais de modifier notre méthode navigate pour prendre en compte les changements

```
onGetDogs() {  
  this.router.navigate(["dogs"], { relativeTo: this.activeRoute });  
}
```


Paramètres d'URL

Scalios Consulting

Pour créer des routes pour une série de données, on pourrait, dans un excès de zèle, être tentés de créer une route pour chacun de nos éléments, ce qui nous amènerait à l'écriture d'une quantité astronomique de lignes de code. Heureusement pour nous, il existe une méthode plus simple, celle de passage de paramètres dans notre route pour créer des routes dynamiques. Pour ce faire, il nous faut éditer nos propriétés **path** dans les objets contenus dans notre constante de type **Routes**. L'ajout d'un paramètre dynamique se fait par la syntaxe « **:nomParamètre** », de sorte à obtenir par exemple :

```
{ path: "dog/:id", component: DogComponent },
```

Le nom de notre paramètre devra être respecté pour la suite des étapes, et il est alors important de lui donner un nom clair. L'accès à ce paramètre passe par l'injection dans le composant cible de **ActivatedRoute**. L'une des informations contenues dans cet objet est l'ensemble des paramètres de l'URL. On peut donc facilement ajouter au hook **ngOnInit()** la récupération de l'élément que l'on souhaite afficher. Il suffit de passer par la propriété **.snapshot.params[<nomParamètre>]**, tel que :

```
ngOnInit(): void {  
  this.dogId = this.route.snapshot.params['id'];  
}
```

Angular, par défaut, ne va pas recharger un composant s'il est déjà affiché, ce qui fait que la modification d'une route n'entraînera pas forcément le rafraichissement des variables. Pour pallier à ce problème, il va nous falloir utiliser la propriété **params** de l'**ActiveRoute**, qui est un « **observable** » (permettant la gestion plus aisée des informations asynchrone) auquel on peut s'inscrire pour en suivre l'évolution :

```
this.dogId = this.route.snapshot.params['id'],  
this.route.params.subscribe((params: Params) => { this.dogId = params['id']; });
```

Se désabonner de l'Observable

Il est recommandé de stocker la souscription aux informations d'un observable dans une variable afin de pouvoir se désabonner en cas de destruction du composant. Cette mécanique évite ainsi la récupération perpétuelle des informations malgré la suppression de notre composant ainsi que l'ajout d'une seconde souscription en cas de relance du composant. Cette variable sera évidemment de type **Subscription** et devra être importée du package **rxjs/Subscription** (rxjs est un package qui nous fournit un nombre conséquent d'outils pour le Javascript et dont dépend Angular)

```
paramsSub: Subscription;
```

Ainsi, on peut facilement créer l'abonnement au moment de la création de notre composant dans le but d'observer les changements des paramètres de l'URL, et se désabonner à chaque fois que ce composant est détruit dans le but de libérer la mémoire et du temps processeur.

```
ngOnInit(): void {  
  this.dogId = this.route.snapshot.params['id'];  
  this.paramsSub = this.route.params.subscribe((params: Params) => { this.dogId = params['id']; });  
}  
  
ngOnDestroy(): void {  
  this.paramsSub.unsubscribe();  
}
```

Paramètres de Requêtes

En plus des paramètres de l'URL, il est également possible de passer des paramètres via la requête HTTP. Les paramètres de requêtes se présentent généralement sous la forme **?g=blabla&test=result** Pour ce faire, il nous faut altérer le fonctionnement de notre navigation. En effet, en plus de l'utilisation de **routerLink**, il est possible de se lier à la propriété **queryParams** pour lui adjoindre nos fameux paramètres. Ces paramètres se présenteront sous la forme d'un objet composé de clés et de valeurs représentant les paramètres et leurs valeurs. A côté de cela, il est possible de passer via la propriété **fragment** pour ajouter par exemple une ancre d'arrivée, afin d'atteindre une section particulière de notre route dès le début.

```
li routerLinkActive="active">  
  <a [routerLink]="['/owners']" [queryParams]="{g: 'blabla', test: 'result'}" fragment="section-one">Propriétaires</a>  
li>
```

De façon dynamique, l'utilisation des paramètres de requête se présente de façon différente. Pour se faire, il faut utiliser l'objet dans lequel nous pouvons ajouter la propriété **relativeTo**, mais cette fois-ci lui passer la propriété **queryParams** et **fragment**, que l'on peut alimenter de la même façon que dans notre template HTML via un objet et une chaîne de caractère.

```
onGetDogs() {  
  this.router.navigate(["dogs"], { queryParams: {d: 'blabla', test: 'result'}, fragment: 'section-one' });  
}
```

Récupérer les queryParams

Encore une fois, dans un soucis d'implémentation de nos fonctionnalités, il nous faut accéder à nos paramètres de requêtes dans nos composants pour pouvoir réaliser des appels à nos services dans le but de récolter les données nécessaires à notre template. Pour réaliser cela, il va nous falloir injecter notre **ActivatedRoute** encore une fois et parcourir ses propriétés.

On peut par exemple commencer par l'utilisation de sa propriété **.snapshot.queryParams** ou **.snapshot.fragment** en fonction de nos besoins. Cette solution va amener aux mêmes soucis que lors de l'utilisation des paramètres d'URL, c'est-à-dire que notre composant ne va pas offrir de rafraichissement de nos valeurs en cas de modification des paramètres de requêtes.

```
this.result = this.route.snapshot.queryParams['test'];  
this.fragment = this.route.snapshot.fragment;
```

L'autre solution est donc de passer par la version **observable**, qui est tout simplement les propriétés **.queryParams** et **.fragment**. En passant par ces deux observables (dont on stockerait idéalement l'abonnement dans une variable en vue de réaliser un désabonnement une fois le composant en destruction), il nous sera possible de profiter d'un rafraichissement de nos valeurs.

```
this.qParamsSub = this.route.queryParams.subscribe();  
this.fragmentSub = this.route.fragment.subscribe();  
}
```

Récupérer les queryParams

Encore une fois, dans un soucis d'implémentation de nos fonctionnalités, il nous faut accéder à nos paramètres de requêtes dans nos composants pour pouvoir réaliser des appels à nos services dans le but de récolter les données nécessaires à notre template. Pour réaliser cela, il va nous falloir injecter notre **ActivatedRoute** encore une fois et parcourir ses propriétés.

On peut par exemple commencer par l'utilisation de sa propriété **.snapshot.queryParams** ou **.snapshot.fragment** en fonction de nos besoins. Cette solution va amener aux mêmes soucis que lors de l'utilisation des paramètres d'URL, c'est-à-dire que notre composant ne va pas offrir de rafraichissement de nos valeurs en cas de modification des paramètres de requêtes.

```
this.result = this.route.snapshot.queryParams['test'];  
this.fragment = this.route.snapshot.fragment;
```

L'autre solution est donc de passer par la version **observable**, qui est tout simplement les propriétés **.queryParams** et **.fragment**. En passant par ces deux observables (dont on stockerait idéalement l'abonnement dans une variable en vue de réaliser un désabonnement une fois le composant en destruction), il nous sera possible de profiter d'un rafraichissement de nos valeurs.

```
this.qParamsSub = this.route.queryParams.subscribe();  
this.fragmentSub = this.route.fragment.subscribe();  
}
```

Créer des Routes Enfants

Il est tout à fait possible de créer des routes dans des routes. Par exemple, on pourrait imaginer que dans notre page de gestion de clients, il se trouverait un menu permettant d'accéder aux détails de chaque client que l'on pourrait sélectionner. Pour ce faire, l'idéal est d'avoir recours à des routes enfants permettant à notre composant présentant la liste des clients d'afficher un composant de détail différent pour chaque client, à proximité de la liste.

Au niveau de nos routes, à proximité des propriétés **path** et **component**, il est également possible d'ajouter une propriété **children** qui prendra en valeur un nouveau tableau de routes.

```
const appRoutes: Routes = [  
  { path: "", component: HomeComponent },  
  { path: "dogs", component: DogComponent, children: [  
    { path: "dogs/:id", component: DogComponent },  
    { path: "dogs/:id/edit", component: DogEditComponent },  
  ]},  
];
```

Pour fonctionner, il faudra cependant à notre composant parent un nouvel élément **<router-outlet>** **</router-outlet>** dans son HTML afin de permettre aux routes enfants de s'afficher.

Il est possible que, lors de l'utilisation de ces routes imbriquées, nous perdions nos paramètres de requête. Pour pallier à ce problème, il nous suffit de modifier la propriété **queryParamsHandling** en « **preserve** » afin de conserver ces derniers. D'autres options, tels que le **merge**, sont également disponibles.

Redirection et Wildcards

Si l'on veut créer une application web, il nous faut penser au niveau de nos routes à l'utilisation de redirections et de routes par défaut. Par exemple, veut-on qu'une personne puisse accéder à toute notre application ? Que se passe-t-il si la personne demande les détails d'un article non existant ?

Pour répondre à ces questions en précisant un peu, il est aisé de se représenter une page 404. Cette page, souvent personnalisée sur les sites internet et porteuses d'easter eggs, sert de point d'arrivée à une route impossible à résoudre. Pour en créer une, il nous suffit de faire un composant, que l'on pourrait appeler par convention un composant de « page not found », et de lui attribuer une route au niveau de notre constante des routes déclarée dans le module de Routing.

Dans nos routes, on peut donc ajouter une route de chemin /not-found qui mènera à notre composant dédié. Une fois fait, il nous est possible d'ajouter des redirections afin de ne pas avoir à réécrire les composants, mais de rediriger une route vers une autre. Pour pousser encore un peu la chose, il nous est possible de placer dans le chemin de la route « ** », qui prendra en compte toutes les valeurs possibles, de sorte à avoir à la fin de toutes nos routes prévues une redirection vers la route menant à notre composant **NotFoundComponent**. Attention, l'emplacement de cette route est crucial afin de lui fournir le fonctionnement adéquat sous peine de bloquer toutes les routes écrites après elle.

```
    ]},  
    { path: "not-found", component: NotFoundComponent },  
    { path: '**', redirectTo: "/not-found"}  
  ];
```


Exporter les Routes

L'écriture de nos routes et de nos configurations prend de plus en plus de place à mesure de l'aggrandissement de notre application, et il devient de plus en plus difficile de nous repérer dans notre fichier **AppModule**. Pour éviter d'entraver la bonne lecture et la compréhension de nos application, il est commun de réaliser un autre module que l'on dédiera à nos routes. Pour ce faire, il nous faut créer une nouvelle classe, celle-ci se voyant être décorée de **@NgModule()**.

Ce module va simplement servir à contenir nos routes, et l'on peut dès lors déplacer à cet emplacement notre constante de routes. Il ne faudra pas oublier d'ajouter dans le décorateur de notre nouvelle classe une propriété **imports** dans laquelle se trouvera désormais l'import de notre **RouterModule** et la configuration de sa méthode **.forRoot()**. Afin de pouvoir faire sortir cette configuration, il nous faudra également la propriété **exports** dans laquelle on exportera notre **RouterModule** désormais configuré. Désormais, ce sera donc notre propre module de routing qu'il nous faudra importer dans **AppModule** afin de faire fonctionner nos routes.

```
const appRoutes: Routes = [
  { path: "", component: HomeComponent },
  { path: "dogs", component: DogComponent, children: [
    { path: "dogs/:id", component: DogComponent },
    { path: "dogs/:id/edit", component: DogEditComponent },
  ] },
  { path: "not-found", component: ErrorPageComponent },
  { path: '**', redirectTo: "/not-found" }
];

@NgModule({
  imports: [
    RouterModule.forRoot(appRoutes, { useHash: true }),
  ],
  exports: [
    RouterModule
  ]
})
```

CanActivate

@Utopios Consulting

Il peut être important de réaliser des instructions à chaque fois qu'une route est demandée ou à chaque fois qu'elle est quittée, par exemple la vérification que l'utilisateur est autorisé à accéder à une route donnée pour en protéger le contenu face à des utilisateurs malveillants ou à des bots. Pour ce faire, Angular se sert d'un système baptisé les Guards.

La protection de nos routes passe par l'utilisation de l'interface **CanActivate**, qui se trouvera dans un service que l'on aurait dédié à la gestion de nos routes et de leur sécurité. L'implémentation de cette interface nous amène à la déclaration de la méthode **canActivate()**, qui prend en paramètre une **ActivatedRouteSnapshot** et un **RouterStateSnapshot**. Cette fonction peut de son côté renvoyer soit un observable de booléen, soit une promesse de booléen ou directement un booléen.

Une fois notre Guard configuré, il ne nous reste plus qu'à l'ajouter à la liste des routes que l'on veut protéger après l'avoir ajouté à la propriété **providers** de notre AppModule.

Pour ce faire, il nous suffit d'ajouter la propriété **canActivate** qui prendra en valeur l'ensemble des Guards que l'on veut tester avant d'accéder à cette route. Si chacun d'entre eux renvoie **true**, alors la route sera atteinte.

```
{ path: "", component: HomeComponent },  
{ path: "dogs", component: DogComponent, canActivate: [AuthGuard], children: [  
  { path: "dogs/:id", component: DogComponent },  
]
```

```
export class AuthGuard implements CanActivate {  
  constructor(private authService: AuthService, private router: Router) {}  
  
  canActivate(route: ActivatedRouteSnapshot,  
    state: RouterStateSnapshot): boolean | Observable<boolean> | Promise<boolean> {  
    return this.authService.isAuthenticated()  
      .then(  
        (authenticated: boolean) => {  
          if (authenticated) {  
            return true;  
          } else {  
            this.router.navigate(['/']);  
            return false;  
          }  
        })  
      );  
  }  
}
```

CanActivateChild

Quipios Consulting

Si ce que l'on cherche à protéger n'est pas forcément tout un composant et ses enfants, mais uniquement les routes de ses enfants, il nous faut alors soit ajouter la propriété `canActivate` sur chacune de nos routes enfants, soit remplacer l'utilisation de l'interface précédente par **CanActivateChild**. Son fonctionnement est similaire, demande les mêmes paramètres et offre le même retour.

```
canActivateChild(route: ActivatedRouteSnapshot,  
state: RouterStateSnapshot): boolean | Observable<boolean> | Promise<boolean> {  
    return this.canActivate(route, state);  
}
```

Une fois fait, ce sera la propriété **canActivateChild** qu'il faudra remplir de notre Guard afin de protéger les routes enfants de notre route parent.

```
{ path: "", component: HomeComponent },  
{ path: "dogs", component: DogComponent, canActivateChild: [AuthGuard], children: [  
    { path: "dogs/:id", component: DogComponent },  
    { path: "dogs/:id/edit", component: DogEditComponent },  
]},
```

CanDeactivate @Utopios Consulting

Il est également possible de contrôler la capacité de notre utilisateur à quitter la page. Par exemple, dans une page d'édition, il ne faudrait pas que l'utilisateur puisse par mégarde faire un retour en arrière et perdre toutes les données qu'il aurait modifié. Ce genre de comportement devra être appelé depuis le composant via l'injection d'un service, notre Guard implémentant **CanDeactivate<T>**. Cette interface est de type générique, et prend en type une **interface** qu'il nous faudra créer pour relier notre service à notre composant.

```
export interface CanComponentDeactivate {  
  canDeactivate: () => Observable<boolean> | Promise<boolean> | boolean;  
}  
  
export class CanDeactivateGuard implements CanDeactivate<CanComponentDeactivate> {  
  canDeactivate(component: CanComponentDeactivate,  
    currentRoute: ActivatedRouteSnapshot,  
    currentState: RouterStateSnapshot,  
    nextState?: RouterStateSnapshot): Observable<boolean> | Promise<boolean> | boolean {  
    return component.canDeactivate();  
  }  
}
```

Une fois l'interface implémentée, notre service devra donc contenir la méthode `canDeactivate()` qui prendra en paramètre un composant (d'où l'ajout de l'interface précédente), la route actuelle, l'état actuel et l'état visé en tant que paramètre optionnel. Une fois nos paramètres donnés, il nous suffit de lui faire retourner le résultat de la méthode que l'on a créé dans notre interface, de façon à permettre au Guard de communiquer avec notre composant (il va à terme appeler sa propre méthode **canDeactivate()**, que l'on doit donc implémenter après l'ajout de notre interface personnalisée parmi ses implémentations). Il ne faudra pas non plus oublier l'ajout de la propriété **canDeactivate** dans notre route.

Passer des données aux Routes

Si l'on veut, il est également possible d'envoyer des données à nos routes, de sorte à pouvoir alimenter nos composants en données statiques immédiatement. Ce genre de processus pourrait se voir être utilisé lors de la création de pages génériques, par exemple une page d'erreur, qui pourrait être alimentée par un message d'erreur personnalisé en fonction de la route que l'utilisateur aurait emprunté.

```
},  
{ path: "not-found", component: ErrorPageComponent, data: { message : "Page not Found" } },  
{ path: "*", redirectText: "(not found)" }
```

Pour ce faire, il nous faut bien sûr relier une propriété via par exemple les chaînes de caractères interpolées à notre template HTML. Une fois fait, il va falloir alimenter cette propriété. Pour cela, il faut passer par la propriété **data** de notre route, dans la liste des **Routes**. Cette propriété prend en paramètre un objet qui sera un ensemble de clés et de valeurs, par exemple la valeur de notre message d'erreur dans une propriété **message**.

Encore une fois, pour récupérer les valeurs, il nous suffit de passer par l'injection dans notre composant de notre **ActivatedRoute** et de ses paramètres en version **snapshot.data** ou via l'abonnement à l'observable **data**.

```
//  
export class ErrorPageComponent implements OnInit {  
  errorMessage!: string;  
  
  constructor(private route: ActivatedRoute) { }  
  
  ngOnInit(): void {  
    this.errorMessage = this.route.snapshot.data['message'];  
  
    this.route.data.subscribe(  
      (data: Data) => {  
        this.errorMessage = data['message'];  
      }  
    );  
  }  
}
```

Passer des Données Dynamiques

Si désormais nous voulons non plus récupérer des données statiques mais des données dynamiques (par exemple stockées dans un serveur), il va nous falloir passer par un Guard résolveur. Ce Guard va nous permettre de récupérer des données qui pourraient être nécessaires à notre composant avant de le faire s'afficher.

```
export class DogResolver implements Resolve<Dog> {  
  constructor(private dogService: DogsService) {}  
  
  resolve(route: ActivatedRouteSnapshot,  
    state: RouterStateSnapshot): Observable<Dog> | Promise<Dog> | Dog {  
    return this.dogService.getDog(+route.params['id']);  
  }  
}
```

Une fois ajouté à la liste des **providers** de notre **AppModule**, on doit désormais l'ajouter dans la route via l'ajout d'une propriété **resolve** qui prendra en valeur un objet. Cet objet sera composé de clés ayant comme valeurs nos Guards Résolveurs. Une fois le composant chargé, il faudra accéder aux données qui se trouveront dans l'observable **data** vu précédemment (Le nom de la propriété dans data sera la même que celle définie dans l'objet resolve).

```
{ path: "dogs/:id", component: DogComponent, resolve: { dog: DogResolver } },
```

Les Stratégies de Localisation

Dans notre environnement de développement, nos routes fonctionnent actuellement à merveille. Malheureusement, ces chemins pourraient ne plus fonctionner lorsque l'on aura déployé notre application sur un serveur distant. Pour configurer correctement nos chemins, il est nécessaire de bien comprendre les stratégies de localisation d'Angular.

Il est possible de se servir de vieilles techniques de nommage pour notre chemin via l'ajout du symbole #. En effet, certains vieux serveur doivent posséder cette séparation afin de pouvoir séparer la route au niveau local et permettre à Angular d'avoir le contrôle de son système de route placé après ce symbole.

```
@NgModule({  
  imports: [  
    RouterModule.forRoot(appRoutes, { useHash: true }),  
  ],  
})
```

Pour ce faire, il nous suffit d'ajouter un objet contenant la propriété **useHash** avec comme valeur **true**. dans les paramètres de la méthode **.forRoot()** de notre RouterModule.

TP : Movin'Out App

@Utopios Consulting

Objectif

Maîtriser les bases du routing, des services et de la communication entre pages Angular.

Sujet

En vous servant du framework Angular, vous devrez réaliser une application permettant à un employé de service de déménagement de consulter sa liste de clients ainsi que la liste des adresses. Les clients auront comme champs leur **nom**, **prénom**, **date de naissance**, **numéro de téléphone**, et **adresse mail** alors que les adresses seront constituées de leur **numéro de rue**, **intitulé de rue**, **code postal** et **commune**.

Ces deux liste pourront être modifiées et possèderont un pseudo-**CRUD** (Create Read Update Delete) basé sur l'utilisation d'un **JSON** partagé via l'utilisation d'un service.

Via l'appel au **service**, votre page de clients ou d'adresse possèdera un display permettant d'accéder aux pages de modification ou de visionnage des détails de cet élément, en plus de permettre sa suppression. Modifier ou visionner les détails de l'élément se fera via un **routing** amenant sur une page dédié à cet effet.

Il sera possible de voir le nombre d'habitants par adresse et le nombre de logements habités par le client dans les listes d'adresses ou de clients. De plus, il sera possible de faire déménager ou emménager un client depuis les page d'éditions des clients ou des adresses.

Les Observables

The bottom of the slide features several overlapping, wavy, horizontal bands in shades of light blue and white, creating a modern, fluid background element.

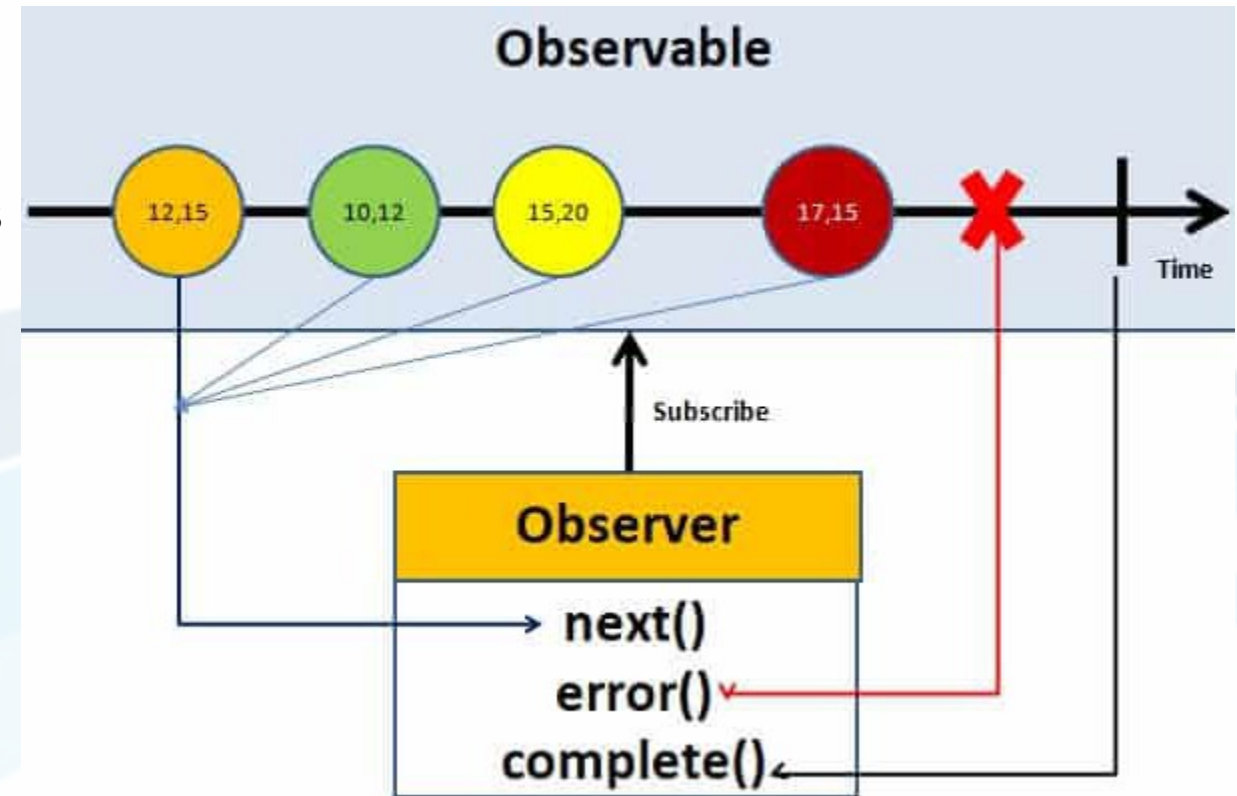
Qu'est-ce qu'un Observable ?

Un observable peut être perçu comme un fournisseur de données. Dans le cadre d'Angular, qui repose sur l'utilisation du package **rxjs**, les observables possèdent des observers. Entre les deux, des données sont émises et peuvent être récoltées ou non par l'**observer** en vue de leur traitement.

Via l'utilisation d'un observable, on peut gérer trois cas possibles :

- Gérer les **données** envoyées par l'observable
- Gérer les **erreurs** envoyées par l'observable
- Gérer la **complétion** de la tâche de l'observable

Pour chacun de ses trois cas, il nous est possible d'écrire bien entendu du code en fonction de nos objectifs. Il n'est d'ailleurs pas obligatoire de gérer les trois cas de figure pour se servir d'un observable. Il est possible pour un observable de ne jamais se compléter (comme ça serait le cas d'un observer attendant le clic d'un bouton part l'utilisateur). Un observable est en quelque sorte une promesse qui va pouvoir envoyer plusieurs fois des informations et également offrir une complétion potentielle.



Notre premier Observable

Dans la section sur le Routing, nous avons utilisé des observables afin de suivre l'évolution de paramètres de nos routes. Que ce soit l'observable **data**, **queryParams**, **fragment** ou **params**, tous les quatre fonctionnent sur le principe d'un abonnement que l'on peut ou non stocker dans une variable. Cet abonnement nous permet de suivre l'état de notre observable, qui nous renverra automatiquement des changements en cas de modification de nos paramètres d'URL ou de requête. Via ce fonctionnement, on peut avoir un rafraîchissement de nos données affichées sur la page web malgré l'utilisation du même composant (ce composant n'étant pas recréé en cas de simple modification des paramètres de sa route).

```
interval().subscribe((count) => {  
  console.log(count)  
});
```

Les observables ne sont pas originaires d'Angular, mais bien du package rxjs, dont Angular se sert (il y a notion de dépendance). Via ce package, nous pouvons créer un observable. Par exemple, via l'import de sa méthode **interval()**, on peut rapidement créer un observable nous renvoyant une série de nombres croissant à un intervalle régulier. On peut donc s'y abonner via la méthode **subscribe()** et y donner en paramètre une méthode, par exemple, une fonction fléchée permettant de logger les nombres renvoyés par notre observable.

```
this.intervalSub = interval().subscribe((count) => { console.log(count) });
```

```
this.intervalSub.unsubscribe();
```

Les observables ne stoppent pas d'émettre des données malgré le fait que l'on quitte notre composant, et il est donc nécessaire de stocker l'abonnement dans une variable afin de pouvoir par la suite s'en désabonner. Les observables gérés par Angular sont automatiquement désabonnés par Angular, mais c'est une bonne pratique de s'habituer à le faire nous-même.

Créer un Observable

Pour créer un observable, il va cette fois-ci falloir importer **Observable** depuis **rxjs**. Via son utilisation, nous disposons de sa méthode **.create()** ou de son **constructeur** qui va prendre en paramètre une fonction devant gérer l'observer. Dans cette fonction, il nous sera possible de faire appel à :

- La méthode **.next()** pour transmettre des données à l'observer
- La méthode **.error()** pour lui transmettre une erreur et stopper l'observable
- La méthode **.complete()** pour terminer l'observable et en informer l'observer.

Par exemple, si l'on voulait recréer l'observable **interval()** vu précédemment, on pourrait s'y prendre de la sorte :

```
const customIntervalObservable = new Observable(observer => {  
  let count = 0;  
  setInterval(() => {  
    observer.next(count);  
    count++;  
  }, 1000);  
});
```

Exercice : Mon premier Observable

Objectif

Appréhender les notions d'Observable dans le framework Angular

Sujet

En vous servant du framework Angular, vous devrez réaliser un observable qui permettra à l'utilisateur de bénéficier chaque seconde d'une nouvelle face d'un dés en comportant 6 (le dé classique). Cet observable sera affiché et rafraîchi chaque seconde sous la forme d'une image de face de dés dans le composant dédié à son affichage.

Les Erreurs et la Complétion

Il est également possible de déclencher des erreurs ou la complétion de notre observable via les méthodes de l'observateur **.error()** et **.complete()**. La première peut prendre un paramètre qui sera renvoyé en tant que message d'erreur à l'observateur une fois l'erreur levée, tandis que l'autre ne fait que signaler la fin de l'observable à l'observateur.

```
const customIntervalObservable = new Observable(observer => {  
  let count = 0;  
  setInterval(() => {  
    if (count > 10) observer.complete();  
    if (count > 5) observer.error("Plus grand que 5, attention !");  
    observer.next(count);  
    count++;  
  }, 1000);  
});
```

Il va désormais nous falloir gérer la réception de ces erreurs ou de cette complétion. Pour se faire, il est possible de passer un objet à la méthode **.subscribe()**. Dans cet objet, des propriétés représentant la récolte des données, la réception d'une erreur ou la réception de la complétion pourront être alimentées par des fonctions exécutées s'il y a réception d'une telle donnée.

```
customIntervalObservable.subscribe({  
  next: (data) => { console.log(data) },  
  error: (error) => console.log(error),  
  complete: () => console.log("Observable is complete !")  
});
```

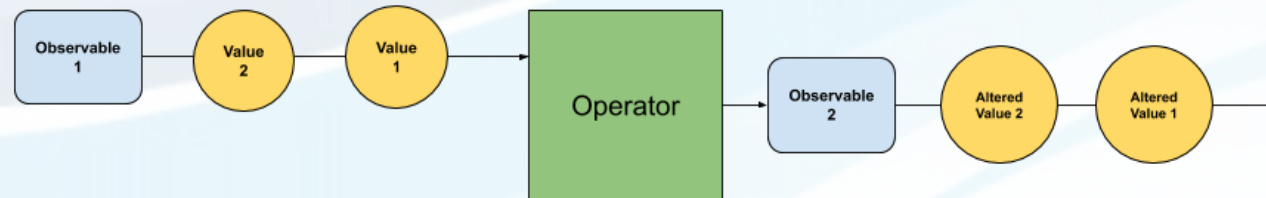

Les Operators

@Utopios Consulting

Les **Operators** sont une fonctionnalité fournie par **rxjs** permettant aux données d'être réceptionnées avant tout par l'opérateur avant la transmission à l'abonnement puis à l'observateur. Via l'ajout d'un opérateur, il est possible d'altérer les données avant leur réception. Par exemple, on pourrait vouloir l'ajout d'un texte avant nos chiffres renvoyés par notre observable maison créé précédemment. On pourrait s'amuser à ajouter cela dans la fonction résolvant la réception des données, mais il est également possible de le faire via un opérateur :

```
customIntervalObservable.pipe(map((data: any) => {  
  return "Round n°" + (data + 1);  
}));
```

Les opérateurs devront se placer dans un pipe (il est possible de les enchaîner en plaçant plusieurs opérateurs séparés par des virgules en paramètre), comme c'est le cas ici de notre opérateur **map** qui va effectuer une modification pour chacune de nos valeurs avant de les renvoyer. Cet opérateur prend lui-même en paramètre la fonction de modification des données. Via cette méthode, nous n'altérons pas les données envoyées par l'observable, ce qui est pratique si l'on n'a tout simplement pas accès à cette classe.



Exercice : Mon premier Operator

Objectif

Appréhender les notions d'Observable dans le framework Angular

Sujet

En vous servant du framework Angular, vous devrez réaliser un opérateur qui permettra à l'utilisateur de cliquer sur un bouton et de se voir proposé une série de lettre aléatoire d'une taille elle aussi aléatoire chaque minute. Ces lettres devront être présentées sous la forme de lignes afin de ressembler à un test de vue classique que l'on pourrait avoir chez l'ophtalmologiste.

L'utilisation de l'opérateur aura pour rôle de transformer les chiffres reçus par l'observable en une série de lettres, via l'utilisation de la table ASCII.

Les Subjects

@Utopios Consulting

Un **Subject** est un élément de notre code qui est capable de remplacer le fonctionnement par **Service** associé à un **EventEmitter<T>** en simplifiant l'émission d'évènement par nos composants de façon globale. Pour se faire, il nous faut, à la place d'un **EventEmitter<T>**, utiliser un **Subject<T>**. A la place de la méthode **.emit()**, nous allons utiliser la méthode **.next()**, comme on l'aurait fait sur un Observable (contrairement à ces derniers, les Subject déclenchent cette méthode de l'extérieur, et non dans leur code). L'utilisation des Subjects est recommandée par rapport à celle des EventEmitter, car elle est plus optimisée et permet la résolution de problèmes plus facilement.

```
didActivate = new Subject<boolean>();
```

Dans le service

```
this.logService.didActivate.next(true);
```

Dans le composant émetteur.

```
this.activateSub = this.logService.didActivate.subscribe({  
  next: (d) => console.log(d)  
});
```

Dans le composant récepteur

Les Formulaires



Angular et les Formulaires

Lorsque l'on travaille avec des formulaires, Angular nous permet d'en obtenir rapidement une représentation sous la forme d'un objet composé d'une propriété **value** rassemblant sous la forme d'un nouvel objet remplis de clés et de valeurs nos différents inputs de formulaire. A côté de la propriété value se trouve également une propriété **valid** permettant de rapidement savoir si notre formulaire est valide ou non.

Il est important de savoir qu'Angular offre deux grandes approches lors de l'utilisation de formulaires :

- Une approche basée sur notre **template (Template-Driven)** : Via cette approche, Angular va inférer l'objet Form depuis notre DOM.
- Une approche basée sur le **code**, dite **Reactive** : Plus complexe, elle offre plus de contrôle et se base sur l'ajout via le code de notre formulaire qui se verra être synchroniser avec le DOM par la suite.

Créer un formulaire (TD)

Nous allons commencer par l'approche basée sur un template. Pour ce faire, il va falloir commencer par créer un formulaire et y inscrire les différents éléments réceptionnant les informations utilisateurs.

Il nous suffit déjà de nous assurer qu'un a bien importé **FormsModule** dans notre **AppModule**. Grâce à ce module, Angular va automatiquement créer un objet correspondant à chaque fois qu'un élément `<form> </form>` est créé. Les inputs ne seront cependant pas ajoutés automatiquement. Pour les inscrire, il nous faut utiliser **ngModel**, sans les crochets et les parenthèses dans un premier temps. Il va nous falloir passer le nom de notre input (et donc bien s'assurer que l'on a ajouté l'attribut **name** à notre input). Une fois tout cela fait, nous avons les bases nécessaires à la réalisation d'un formulaire via Angular.

```
<div class="row">
  <form (ngSubmit)="onSubmit(f)" #f="ngForm">
    <div class="col-12 col-md-6">
      <input type="text" id="dog-name" name="dog-name" ngModel>
    </div>
  </form>
</div>
```

Pour analyser les données envoyées, il va nous falloir ajouter un lien sur l'événement **ngSubmit** dans notre formulaire. Si l'on veut voir les données, il est possible d'utiliser des références locales pour accéder à l'élément HTML, mais il est également possible de donner une valeur à cette référence locale : **ngForm**. Grâce à cela, nous n'aurons plus un **HTMLFormElement**, mais un **ngForm** en paramètre de notre méthode **onSubmit()**. Dans cet objet, il est possible de trouver une propriété **value** qui contient les valeurs envoyées par l'utilisateur. Il est également possible d'accéder au formulaire via l'utilisation du décorateur **@ViewChild()** pour récupérer l'élément html dans notre vue via la référence locale .

```
@ViewChild("f") addDogForm!: NgForm;
```

D'autres propriétés, comme la validation, l'édition, la désactivation ou autre se trouvent également dans cet objet. Ces propriétés participent ensemble à l'**état** de notre formulaire.

La Validation d'un formulaire

Afin d'améliorer l'expérience utilisateur, il va nous falloir utiliser des éléments de code permettant de valider par exemple nos inputs pour les entourer de bordures rouges en cas de non validité. Pour ajouter la validité, on pourrait par exemple ajouter l'attribut **required** sur nos inputs pour qu'Angular les détecte et s'en serve. La directive **email** existe également si l'on veut rendre un input valide en cas de présence d'une adresse email. Une liste des validateurs est disponible à l'adresse suivante : <https://angular.io/api/forms/Validators>

Le caractère de validité se trouve à la fois au niveau du formulaire et également au niveau des différents inputs. Pour ce faire, Angular ajoute des classes telles que **ng-dirty**, **ng-valid**, etc... pour donner l'état des inputs.

```
</div class="text-end mb-3">  
<button class="btn btn-primary" [disabled]="!f.valid" type="submit">  
  <i class="bi bi-send"></i> Envoyer  
</button>
```

On peut ensuite par exemple lier la propriété **disabled** d'un bouton d'envoi des données à la propriété **valid** de notre formulaire via sa référence locale. Une autre chose que l'on peut faire est de modifier notre CSS pour se lier à la classe **ng-invalid** créée par Angular pour nos inputs dans le but de fournir une bordure rouge à nos inputs. Il va nous falloir également faire attention à la présence de la classe **ng-touched** afin de ne pas avoir de bordures rouges dès le début.

Si l'on veut ajouter un texte d'alerte en dessous de notre contrôle, il va nous falloir lui ajouter une référence locale, cette fois-ci liée à **ngModel** afin d'avoir accès à sa validité. Via cette méthode, nous pourrions ainsi rendre visible ou non notre texte.

```
<div class="col-8">  
  <input type="text" id="dog-name" name="dog-name" ngModel #dogName="ngModel">  
</div>  
<div class="col-6" *ngIf="!dogName.valid && dogName.touched">  
  <span class="text-danger">Veuillez entrer un nom valide !</span>  
</div>
```

Améliorer notre formulaire

Si l'on veut, il est possible d'altérer notre ajout de **ngModel** en ajoutant des crochets autour. De la sorte, on peut le lier à une propriété, celle-ci possédant la valeur par défaut de notre input. Dans le cas où l'on souhaite réagir également aux changements, alors l'utilisation du **Two-Way Databinding** (et donc des parenthèses) sera nécessaire. Via cela, on peut offrir une réactivité plus avancée à notre formulaire et afficher dans l'interface utilisateur des informations à mesure qu'il nous offre des données. Il existe ainsi trois façons de se servir de ngModel :

- **Directive simple** : Permet à Angular de récolter les informations contenues dans notre input et de les ajouter à notre objet NgForm
- **Property Binding** : Permet de saisir des valeurs par défaut alimentées par une propriété dans notre code Typescript.
- **Two-Way Data Binding** : Permet d'offrir une réactivité et une modification des valeurs affichées dans notre interface lors de la modification des valeurs contenues dans les inputs.

Il est également possible de regrouper nos inputs dans le but de les rassembler au niveau de notre objet JS. Pour se faire, il nous faut ajouter la directive **ngModelGroup** à un contenant HTML regroupant nos éléments enfants en lui donnant une valeur représentant le nom du groupe. Ce groupe possèdera également toutes les propriétés de validation, de modification, etc... d'un input ou du formulaire. Ce groupe pourra également être référencé et cette référence contenir cette fois-ci ngModelGroup pour offrir des possibilités au niveau de notre code ou de notre style.

```
form (ngSubmit)="onSubmit()" #f="ngForm">  
  <div class="col-12 col-md-6" ngModelGroup #dogInfos="ngModelGroup">  
    <div class="row mb-3">  
      <div class="col-4">
```

Les Valeurs du Formulaire

Si l'on souhaite modifier les valeurs de notre formulaire, il est possible de le faire via l'utilisation du Data Binding. Il existe également une autre méthode, se servant des deux méthodes **setValue()** ou se trouvant sur notre objet de type NgForm récupéré via l'utilisation du décorateur **@ViewChild()** :

- **setValue()** : Cette méthode prendra en paramètre un objet servant de lien vers notre objet final. Il sera important donc de respecter la hiérarchie des propriétés et de leurs valeurs pour obtenir la même structure que notre objet final envoyé par le formulaire. De plus, via cette méthode, l'absence d'une valeur pour une des propriétés va remettre à zéro l'élément non setté, et également remplacer les valeurs des autres inputs.
- **patchValue()** : Se trouve sur l'objet form de notre NgForm, et permet, en plus de setter les valeurs de certains de nos inputs, de conserver les valeurs précédemment entrées par l'utilisateur.

Pour afficher les valeurs lors de l'envoi, il nous suffirait de lier des propriétés implémentées dans l'interface via des **String Interpolations**, et les cacher derrière une **<div>** comprenant la directive ***ngIf** reliée à un **booléen** modifié lors de la méthode déclenchée par **NgSubmit**.

Si l'on veut ensuite effacer le formulaire, il est possible de le faire via la méthode **.reset()** disponible sur l'objet de type NgForm récupéré par notre décorateur **@ViewChild()**. En plus de resetter les valeurs, cette méthode va également remettre à zéro l'état de notre formulaire.

Exercice : Formulaire TD

@topios Consulting

Objectif

Appréhender la manipulation et l'utilisation du module FormsModule dans le framework Angular

Sujet

En vous servant du framework Angular, vous devrez réaliser un formulaire possédant ces informations :

- Une **adresse mail**
- Un **mot de passe**
- Un sélecteur de **type d'abonnement** (Basique, Avancé, Pro) avec **Avancé** comme valeur par défaut.
- Un bouton pour **envoyer le formulaire**

Ce formulaire devra se servir de la création de formulaire de façon « **Template driven** ».

En fonction de la modification du formulaire, vous devrez afficher en dessous de chaque input s'il est correct ou non en avertissant des valeurs incorrectes d'un message.

De plus, le bouton d'envoi du formulaire ne devra être disponible que suite à l'ajout de toutes les valeurs requises. Cet envoi provoquera son affichage dans la console.

L'Approche Reactive

L'autre façon de faire des formulaires en Angular, l'approche dite Reactive, repose sur la création de notre formulaire par la création d'une propriété de type **FormGroup**. Cette propriété repose sur l'utilisation du module **ReactiveFormsModule** à la place de **FormsModule** au niveau de notre **AppModule**.

Notre propriété **FormGroup** devra être initialisée avant le rendu du formulaire, par exemple dans le hook **NgOnInit()**. Dans la propriété, il nous faudra passer un nouvel objet **FormGroup**, via son constructeur qui prendra en paramètre l'objet représentant notre formulaire. Il ne nous reste plus qu'à remplir chacune des propriétés par des éléments de type **FormControl**. Dans leur constructeur, il nous faudra passer l'état de base, des validateurs synchrones et des validateurs asynchrones.

Pour synchroniser l'ensemble avec notre HTML, il va falloir ajouter des directives à notre élément **<form> </form>**. Pour commencer, il va nous falloir la directive **formGroup** à laquelle on va lier la propriété que l'on a créé pour stocker le formulaire **FormGroup**.

Pour chaque input, il va nous falloir ajouter la directive **formControlName** et la lier au nom que l'on a donné à nos **FormControl**.

```
export class DogAddComponent implements OnInit {  
  genders = ['male', 'female'];  
  addDogForm!: FormGroup;  
  
  constructor() { }  
  
  ngOnInit(): void {  
    this.addDogForm = new FormGroup({  
      'dog-name': new FormControl(null),  
      'owner-name': new FormControl(null),  
      'gender': new FormControl('male')  
    });  
  }  
}
```

```
<div class="row">  
  <form [formGroup]="addDogForm">  
    <div class="col-12">  
      <input type="text" id="dog-name" name="dog-name" formControlName="dog-name" />  
    </div>  
  </form>  
</div>
```

```
<div class="col-8">  
  <input type="text" id="dog-name" name="dog-name" formControlName="dog-name" />  
</div>
```


Manipuler le Formulaire

Pour envoyer les données, on va également pouvoir se servir de la liaison sur l'évènement `ngSubmit`. La différence est que l'on n'a plus besoin de récupérer l'objet créé pour nous. Il nous suffit d'analyser les valeurs stockées dans notre propriété de formulaire (la propriété de type **FormGroup**).

Pour la validation, il va falloir ici configurer cette dernière dans notre code, et non dans notre template. Il va falloir passer par les validateurs que l'on peut ajouter sous la forme d'un élément unique ou d'un tableau dans nos constructeurs de **FormControl**.

```
ngOnInit(): void {  
  this.addDogForm = new FormGroup({  
    'dog-name': new FormControl(null, Validators.required),  
    'owner-name': new FormControl(null, Validators.required),  
    'owner-email': new FormControl(null, [Validators.required, Validators.email]),  
  });  
}
```

Pour accéder aux contrôles et afficher des messages en cas de valeurs incorrectes, il nous suffira ici de passer par les méthodes **.get()** qui permettra d'accéder au contrôle via la hiérarchie menant à lui ou à son nom.

```
<div class="col-6" *ngIf="!addDogForm.get('dog-name')?.valid && addDogForm.get('dog-name')?.touched">
```

Pour grouper des éléments de notre formulaire, il nous faut dans l'approche réactive passer des **FormGroup** dans des **FormGroup**. Au niveau de notre template, les éléments HTML devront se trouver dans un élément portant la directive **formGroupName** alimentée par le nom écrit dans le **FormGroup** parent.

```
ngOnInit(): void {  
  this.addDogForm = new FormGroup({  
    'dog-name': new FormControl(null, Validators.required),  
    'owner-data': new FormGroup({  
      'owner-name': new FormControl(null, Validators.required),  
      'owner-email': new FormControl(null, [Validators.required, Validators.email]),  
    }),  
    'gender': new FormControl('male')  
  });  
}
```


FormArray

@Utopios Consulting

Imaginons désormais que l'on veuille permettre à l'utilisateur d'ajouter des contrôles à notre formulaire, comme par exemple des inputs pour ajouter des descriptifs à l'un de nos objets. Pour ce faire, il va falloir donner comme valeur à l'une de nos propriétés de FormGroup un **FormArray**.

```
'gender': new FormControl('male')
'toys': new FormArray([])
);
```

Une fois fait, il est possible d'ajouter une méthode dans notre code Typescript qui se chargera d'effectuer la méthode **.push()** sur notre FormArray, en lui ajoutant des élément de type FormControl par exemple.

```
onAddToy() {
  const control = new FormControl(null, Validators.required);
  (<FormArray>this.addDogForm.get('toys')).push(control);
}
```

Pour le visionner dans notre template, on peut désormais se servir de la directive **formArrayName**, qui prendra en paramètre le nom de notre FormArray., et qui contiendra, à terme, la boucle permettant de générer nos élément. Pour générer les éléments, il nous faudra une méthode renvoyant la propriété **controls** après un casting du résultat de notre **.get()** en FormArray :

```
getToyControls() {
  return (<FormArray>this.addDogForm.get('toys')).controls;
}
```

```
<div class="row mb-3" formArrayName="toys">
  <h4>Ses Jouets</h4>
  <button class="btn btn-primary" (click)="onAddToy()" type="button">Ajouter un jouet</button>
  <div class="mb-3" *ngFor="let toys of getToyControls(); let i=index;">
    <input type="text" [formControlName]="i">
  </div>
</div>
```

Créer un Validator personnalisé

Il est tout à fait possible de réaliser des validateurs personnalisés, au cas où, par hasard, les validateurs disponibles ne nous suffisent pas. Un validateur n'est ni plus ni moins qu'une fonction qui se verra être exécutée lors de la modification des données dans notre formulaire. Cette fonction prendra en paramètre notre **FormControl** et renverra un objet qui aura comme propriété une clé de type string et en valeur un booléen ou une valeur nulle.

```
forbiddenNames(control: FormControl): {[s:string]: boolean} | null {  
    if (this.forbiddenNamesValues.indexOf(control.value) !== -1) {  
        return {'nameForbidden': true};  
    }  
    return null;  
}
```

Il ne faudra pas oublier, lors de l'ajout de notre Validateur, de binder **this** à notre composant sous peine d'avoir des erreurs de référence.

```
this.addDogForm = new FormGroup({  
    'dog-name': new FormControl(null, [Validators.required, this.forbiddenNames.bind(this)]),  
    'owner-data': new FormGroup({
```

La combinaison de clé et de valeur produites par notre validateur se retrouvera dans un tableau contenu dans la propriété **errors** de notre FormControl. Il est ainsi possible de faire des instructions pour afficher des messages d'erreurs personnalisés en dessous de nos différents contrôles.

Gérer les valeurs en Reactive

Lorsque l'on veut suivre, modifier ou vérifier les valeurs de notre formulaire en approche reactive, il est facile de le faire via l'abonnement aux observables **valueChanges** et **statusChanges**. Via leur utilisation, il est facile de loguer les différences de valeurs ou de statut de notre formulaire et de nos contrôles.

```
this.addDogForm.valueChanges.subscribe((data) => {  
  console.log(data);  
});  
  
this.addDogForm.statusChanges.subscribe((data) => {  
  console.log(data);  
});  
}
```

Pour setter les valeurs, il existe la méthode **.setValues()** sur l'objet de type FormGroup. Fonctionnant en passant en paramètre un objet représentant la hiérarchie de clés et de valeurs de notre formulaire JSON, il existe également la méthode **.patchValues()**, ainsi que **.reset()**, qui rapellons le, peut prendre également un objet pour avoir des valeurs de base après remise à zéro de notre formulaire.

```
this.addDogForm.reset({  
  'owner-name': "Bernard",  
  'dog-name': "Rex"  
});
```

Exercice : Formulaire Reactive

Objectif

Appréhender la manipulation et l'utilisation du module ReactiveFormsModule dans le framework Angular

Sujet

En vous servant du framework Angular, vous devrez réaliser un formulaire possédant ces informations :


- Une **adresse mail** pour le créateur
- Un **nom de projet** qui ne pourra pas être « Test »
- Un sélecteur de **statut du projet** (Stable, Critique, Finis) avec comme valeur par défaut **Critique**
- Un bouton pour **envoyer le formulaire**

Ce formulaire devra se servir de la création de formulaire de façon « **Reactive** ». En fonction de la modification du formulaire, le bouton d'envoi du formulaire devra ou non être cliquable.

Cette vérification passera par l'utilisation des **vérificateurs**. Pour vérifier le nom du projet, vous devrez réaliser votre vérificateur personnalisé.

Une fois cliqué, le bouton devra **renvoyer en console le contenu du formulaire**.

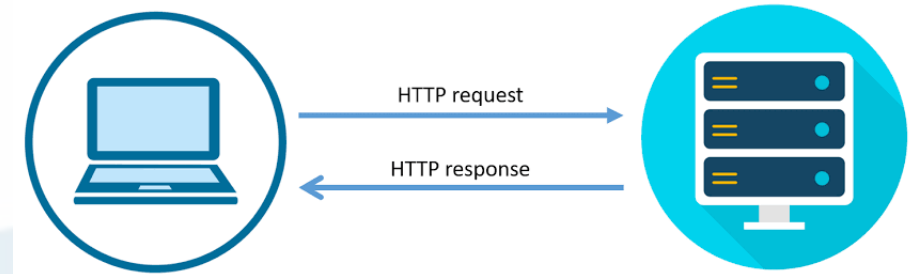
Les Requêtes HTTP

The bottom of the slide features several overlapping, wavy lines in shades of light blue and white, creating a modern, fluid background element.

A quoi ça sert ? @Utopios Consulting

Jusqu'à présent, notre application fonctionnait d'elle-même, sans avoir recours à l'aide d'un serveur ou d'une base de donnée. Ce fonctionnement, bien que réactive et pratique, offre un problème de taille. Tout rafraichissement de notre page entraine une perte de toutes les données. A côté de cela, il nous est également impossible de faire appel à des services externes tels qu'une **API**. Pour ce faire, nous avons besoins d'effectuer des **requêtes HTTP**. Ces requêtes contiendront l'ensemble de l'adresse du service API que l'on voudrait atteindre, en plus de stocker dans son header des potentielles informations d'authentification et dans son corps des paramètres qui pourraient être nécessaire à l'API pour fonctionner. L'utilisation d'une API est cruciale lors de l'accès à une base de données, notre application Front-end n'offrant pas de sécurité suffisante pour stocker les informations de connexion à une base de données.

Une fois la requête envoyée, il va nous falloir attendre la réponse du serveur, qui peut être de plusieurs type : Elle peut être favorable et nous offrir les informations demandées (code **200**), la réponse pourrait ne jamais revenir à cause d'une adresse fausse ou d'un serveur éteint (code **404**), nous pourrions ne pas être autorisé à accéder à cette fonctionnalité (code **401**), etc...



Il va donc nous falloir idéalement traiter tous les cas potentiels de sorte à offrir à notre utilisateur un affichage adapté. De plus, afin de ne pas perturber le fonctionnement de l'application Angular, ces requêtes devront se faire de façon **asynchrone** pour ne pas bloquer le déroulement de notre interface.

Firestore

@Utopios Consulting

Firestore est un service fourni par Google afin de stocker rapidement et simplement des données en ligne. Cet outil nous offre, en plus d'une base de données NoSQL, une API permettant de la manipuler. Elle est ainsi idéale lorsque l'on veut rapidement tester une application Angular. Pour accéder à Firestore, il nous faut disposer d'un compte Google, puis nous connecter à Firestore. Une fois fait, il va nous falloir créer une nouvelle application :

Une fois fait, il va nous falloir créer une base de données Realtime, que l'on paramètrera en mode Test :

🔵 Démarrer en **mode test**

Par défaut, vos données sont publiques pour permettre une configuration rapide. Toutefois, vous devez modifier vos règles de sécurité dans les 30 jours pour autoriser l'accès client en lecture/écriture sur le long terme.

Commençons par donner un nom à votre projet®

Nom du projet

test-demo-app

test-demo-app-fda13

Une fois la base de données créée, elle nous offrira directement une adresse HTTP nous permettant de manipuler les données, en envoyant par exemple une requête **POST** (Ajout de données).

🔗 <https://test-demo-app-fda13-default-rtdb.europe-west1.firebaseio.com>

```
https://test-demo-app-fda13-default-rtdb.europe-west1.firebaseio.com/: null
```

Les Verbes REST

Utopios Consulting

Lorsque l'on manipule une API permettant l'accès à des données, il existe une série de verbes qu'il nous est recommandé de connaître. Ces verbes sont les verbes nécessaires à une API pour être qualifiée d'API REST. Il y en a plusieurs, mais les plus courants sont sans nul doute les 5 verbes suivants :

- **GET** : Permet l'accès à une ou plusieurs données afin de les lire dans une interface externe (ici, notre application Front-End)
- **POST** : Permet l'ajout de données à l'API, qui va se charger de les relier à la base de données et de les y stocker. Une requête de type POST contient généralement un corps rassemblant sous la forme d'un JSON ou de données de formulaire lesdites données.
- **DELETE** : Permet la suppression de données dans la base de données. L'API va généralement demander comme paramètre l'ID de l'élément à supprimer, et offrir une réponse adaptée pour confirmer ou infirmer la suppression (si par exemple l'ID n'existe pas).
- **PATCH** : Permet la modification des données. Ce verbe va souvent demander un ID pour aller chercher l'élément, et un corps contenant les informations à modifier sur l'objet. Il n'est pas nécessaire de renseigner toutes les valeurs (ce choix est néanmoins au dépend de l'API), on peut simplement donner celles que l'on souhaite modifier via un ensemble de clés et de valeurs.
- **PUT** : Permet l'ajout ou la modification des données. Contrairement à PATCH, ce verbe est une sorte de passe-partout, qui va ajouter des données en cas d'ID manquante, ou modifier les données si l'ID est existant.

Envoyer des données : POST

Pour faire fonctionner les requêtes HTTP sur Angular, il va nous falloir faire appel au module **HttpClientModule** (provenant de **@angular/common/http**), qu'il ne faudra pas oublier d'ajouter aux imports au niveau de notre **AppModule**. Pour nous en servir par la suite dans notre composant, il ne nous reste plus qu'à injecter dans une propriété le **HttpClient**.

```
, private http: HttpClient) { }
```

Une fois le client HTTP injecté, il est possible de se servir de ses méthodes, dont la méthode **.post()**, pour manipuler une API. Dans notre cas, il va nous falloir utiliser notre URL d'API Firebase, qui permet de lui concaténer une propriété dans laquelle seront rangés les valeurs. Pour ce faire, il suffit de se servir de l'adresse de base et de lui ajouter un nom de propriété séparé via un caractère **/** et suffixé de **.json** :

```
const baseUrl = "https://test-demo-app-fda13-default-rtdb.europe-west1.firebaseio.com";  
const finalUrl = baseUrl + "/dogs.json";
```

Le second paramètre obligatoire de la méthode **.post()** est un corps (le **body**). Ce corps doit contenir un objet, qui ici sera notre nouveau chien :

```
const newDog = new Dog("Bernie", "German Shepard", 2, true);  
  
this.http.post(finalUrl, newDog);
```

Se basant sur le principe des observables, il va nous falloir nous inscrire à cet observable pour récupérer des informations:

```
this.http.post(finalUrl, newDog).subscribe(postData => { console.log(postData) });
```

Récupérer des données : GET

Maintenant que nous avons pu ajouter des données dans notre base de données distante, il nous est possible, en cas de besoin, de les récupérer pour nous en servir dans notre application. Pour ce faire, il va nous falloir utiliser un autre verbe des requêtes : le verbe **GET**. Notre client HTTP nous permet, comme pour la méthode **.post()**, d'avoir une méthode pour réaliser cette tâche : la méthode **.get()**.

```
onFetchDogs() {  
  const baseUrl = "https://test-demo-app-fda13-default-rtdb.europe-west1.firebaseio.com";  
  const finalUrl = baseUrl + "/dogs.json";  
  
  this.http.get(finalUrl).subscribe(postData => { console.log(postData) });  
}
```

Notre méthode se présentera de la même façon que la méthode précédente, mais ne demandera pas de corps obligatoire pour être envoyée (en effet, le corps qui nous intéresse ici est plutôt celui de la réponse). La réponse, dans notre cas, sera un ou plusieurs objets Javascript encapsulés derrière des clés générées automatiquement par Firebase. Il ne nous restera plus qu'à les transformer en nous servant d'un opérateur.

```
this.http.get<{ [fireKey: string] : Dog }>(apiURL)  
  .pipe(map(getData => {  
    const finalArray = [];  
    for (const k in getData) {  
      finalArray.push( { ...getData[k], firebaseKey: k } )  
    }  
    return finalArray;  
  } ) )  
  .subscribe(getData => {  
    console.log(getData);  
  } )
```

Utiliser un service pour l'HTTP

Bien entendu, toutes ces manipulations de données sont un scénario typique où l'on souhaiterait utiliser un service. Via ce service, nous pourrions facilement décentraliser tout notre code dans une classe qui contiendrait nos méthodes d'appel à l'API. Grâce à cela, il nous serait du coup plus aisé d'avoir des composants lisibles, dont le code ne concerne que l'interface et non les fonctionnalités de manipulation de nos données.

Une fois le service fait, il va nous falloir récupérer la connexion et la mise à jour de notre interface par le service. Pour ce faire, il va déjà falloir injecter notre service et nous en servir.

Ensuite, deux possibilités s'offrent à nous. La première serait l'utilisation d'un **Subject** sur lequel nous appellerions la méthode **.next()** et qui serait réceptionné par notre composant. L'autre solution est tout simplement de retourner l'observable généré par notre méthode de GET et de le traiter au niveau du composant.

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

  constructor(private http: HttpClient) {}

  createAndStoreDog(name: string, breed: string, age: number, isMale: boolean) {
    const newDog = new Dog(name, breed, age, isMale);

    this.http.post(apiURL, newDog).subscribe(postData => {
      console.log(postData);
    });
  }
}
```

```
this.isFetching = true;
this.dogService.fetchDogs().subscribe(dogs => {
  this.isFetching = false;
  this.dogs = dogs;
});
```

Supprimer des données : DELETE

Imaginons désormais que l'on veuille supprimer des données se trouvant dans notre base de données Firebase. Pour ce faire, il va nous falloir envoyer une nouvelle requête HTTP, mais cette fois-ci portant le verbe DELETE. Dans le cas où l'on souhaite simplement supprimer tous les éléments de notre base de données, il nous suffit de cibler le JSON dans son intégralité (comme nous le faisons pour les requêtes POST et GET) et de demander à notre client HTTP sa méthode **.delete()**

```
deleteDogs() {  
  return this.http.delete(apiURL);  
}
```

Cette méthode devra être retournée afin que l'observable qu'elle retourne puisse être traité dans notre application au niveau du composant. De la sorte, on pourra retirer tous les éléments de notre interface une fois ceux-ci supprimés au niveau du serveur.

```
this.dogService.fetchDogs().subscribe( {  
  next: dogs => {  
    this.isFetching = false;  
    this.dogs = dogs;  
  },  
  error: error => {  
    this.error = error.message;  
  }  
});
```


Gérer les erreurs

apios Consulting

Dans le cas désormais d'une erreur (comme c'est souvent le cas lorsque l'on traite avec des données ou fonctionnalités externes à notre application), il va nous falloir gérer les erreurs dans le but de fournir une expérience utilisateur de meilleure qualité. Pour ce faire, il nous suffit d'altérer notre méthode **.subscribe()** pour prendre en compte l'erreur, que l'on pourrait imaginer être affichée dans une **<div>** en cas de besoin.

Une autre solution en cas d'utilisation du service dans plusieurs composants serait de passer par l'utilisation d'un **Subject<T>** que l'on réceptionnerait dans chacun de nos composants, par exemple pour afficher le message ou le code d'erreur.

```
this.dogService.fetchDogs().subscribe( {  
  next: dogs => {  
    this.isFetching = false;  
    this.dogs = dogs;  
  },  
  error: error => {  
    this.error = error.message;  
  }  
});
```

```
fetchDogs() {  
  return this.http.get<{ [fireKey: string] : Dog }>(apiURL)  
    .pipe(map(getData => {  
      const finalArray = [];  
      for (const k in getData) {  
        finalArray.push( { ...getData[k], firebaseKey: k} )  
      }  
      return finalArray;  
    })),  
    catchError(errorResponse => {  
      return throwError(errorResponse.message);  
    }));  
}
```

Enfin, la troisième solution serait de nous servir de l'opérateur **catchError** fourni par **rxjs**. Via cet opérateur, il nous est possible de réceptionner dans le pipe une erreur, et de potentiellement la renvoyer via l'utilisation de l'observable **throwError**. De cette façon, nous n'avons pas forcément besoin d'altérer le fonctionnement de nos ou de notre composant comme pour l'utilisation d'un **Subject<T>**.

Configurer notre Requête

Il est important de savoir que l'on peut bien entendu ajouter des Headers à notre requête. Ces headers s'avèrent être essentiels lorsque l'on travaille avec une API de niveau professionnel car ils nous permettent par exemple de gérer nos **Credentials**. Les credentials se trouvent être souvent des clés d'API ou des informations permettant à l'API de nous autoriser à utiliser son ou ses services (une API peut facilement verrouiller des services à une catégorie de personnes mais pas à d'autres).

Pour ajouter des Headers à notre requête, il suffit, en paramètre supplémentaire de nos méthodes **.post()**, **.delete()**, **.get()**, etc.. d'ajouter un objet qui va contenir comme propriété **headers**. Cette propriété prendra en paramètre un objet de type **HttpHeaders**, qu'il nous faudra construire via le passage en paramètre de son constructeur d'un objet composé des clés et des valeurs que l'on veut retrouver dans les headers de notre requêtes.

```
return this.http.get<{ [fireKey: string] : Dog }>(apiURL, {
  headers: new HttpHeaders({
    "mon-header": "blabla",
    "API-KEY": "dqsdqddq-gfgd55fd84-sfsfsf5s6d1s"
  })
})
.pipe(map(getData => {
```

Il est également possible d'attacher à notre requête des paramètres. Dans ce cas, il va nous falloir passer par l'utilisation de la propriété **params** qui se trouve au même niveau que headers. Dans cette propriété, nous pourrons passer un objet de type **HttpParams**.

Cet objet, une fois créé, possèdera la méthode **.set()** pour lui donner un paramètre, ou la méthode **.append()** qui retournera un nouvel objet HttpParams auquel on pourra de nouveau ajouter des paramètres :

```
let myParams = new HttpParams();
myParams = myParams.append('truc', 'machin');
myParams = myParams.append('verif', 'nope');
```

Avoir le détail de notre Réponse

Par défaut, Angular va extraire les informations contenues dans les réponses HTTP pour nous fournir un élément de travail plus concis. Si toutefois nous désirons obtenir l'ensemble de la réponse, il va nous falloir altérer nos méthodes de client HTTP.

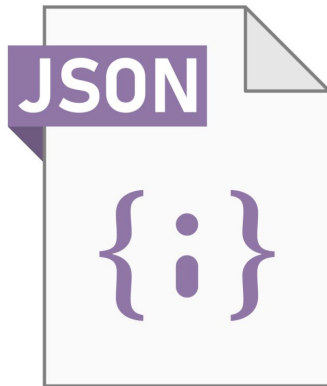
Pour modifier ce comportement, il va falloir utiliser encore une fois l'une des propriétés de notre objet contenant les options de notre méthode. La propriété qu'il nous faut cibler est cette fois-ci la propriété **observe**. Sa valeur par défaut est **body**, d'où le fait que nous ne disposions que du corps de la réponse jusqu'à maintenant. En changeant cette valeur à **response**, nous aurons donc l'ensemble de la réponse en objet exploitable. Dans cet objet, nous aurons une propriété **body**, une propriété **headers**, des **status**, etc...

Il est également possible d'observer **events**. Ces événements correspondent à des valeurs numériques, qu'il nous est possible de relier facilement au niveau de notre Typescript à une **enum** (une association de valeurs numériques avec du texte plus lisible). Il existe par exemple un événement pour signifier que la requête a été envoyée, que l'upload du fichier est en cours, qu'une réponse est arrivée, etc... Via l'opérateur de **pipe()** **tap** fourni par **rxjs**, il nous est possible de les logger sans altérer notre code.

```
this.http.post(apiURL, newDog, { observe: 'events' })
  .pipe(tap(events => {
    if (events.type === HttpEventType.Response) console.log("We got an answer !");
  }))
  .subscribe(postData => {
    console.log(postData);
  });
```

Changer le type de réponse

Il est également possible de spécifier le type de réponse que l'on est censé avoir. Par défaut, Angular va tenter de transformer notre réponse en un objet Javascript, mais il est possible de le conserver en tant que texte brut. Pour ce faire, il va falloir, dans l'objet d'options de notre requête, modifier la propriété **responseType** en lui passant par exemple la valeur de **text**. Les différents types de réponses sont définis au niveau du serveur et de l'API, et il est alors important de les connaître en lisant la documentation.



```
"header": {  
  "title": "The JSON example",  
  "descriptionText": "This is some title text."  
},  
"content": {  
  "title": "The content example text",  
  "elements": [  
    {  
      "title": "The first element",  
      "mainText": "First element main text",  
      "additionalText": "First element additional te  
    },  
    {  
      "title": "The second element",  
      "mainText": "Second element main text",  
      "additionalText": "Second element additional
```

Les Intercepteurs

Elcios Consulting

Lorsque l'on utilise les requêtes HTTP, il existe une fonctionnalité très souvent associée au client HTTP lorsque l'on réalise des applications avec Angular : les **Intercepteurs**. Les intercepteurs ont comme but principal l'altération de nos requêtes avant leur envoi réel. Par exemple, si jamais nous voulons nous identifier, puis faire des requêtes HTTP, il nous faudrait injecter des variables comme par exemple notre clé API dans la requête.

Actuellement, il nous faudrait encoder ce phénomène sur chaque requête, et donc, en cas de modification, recoder l'ensemble. Via l'utilisation des intercepteurs, on peut simplement demander l'exécution d'un intercepteur sur ces requêtes, et en cas de changement, changer l'intercepteur sans avoir à altérer la requête.

Pour réaliser un intercepteur, il nous faut faire un service (une classe) qui va implémenter **HttpInterceptor**. Cette interface va nous imposer la création de la méthode **intercept()**

```
export class AuthInterceptorService implements HttpInterceptor {  
    intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {  
        // Notre code métier  
        return next.handle(req);  
    }  
}
```

qui prend en paramètre deux choses: une **HttpRequest<T>** et un **HttpHandler**. En fin de la méthode, après l'altération de notre requête ou tout autre code métier, il va nous falloir retourner la méthode **.handle()** sur notre second paramètre (en lui donnant la requête en paramètre) pour pouvoir exécuter la requête.

L'ajout de cet intercepteur dans nos providers est un peu spécial, il passe par l'ajout d'un objet. Sa propriété **multi** permet à Angular de savoir s'il y aura plusieurs intercepteurs à exécuter avant chaque requête ou non.

```
{  
    provide: HTTP_INTERCEPTORS,  
    useClass: AuthInterceptorService,  
    multi: true  
}
```


Altérer la requête

ios Consulting

Dans notre intercepteur, on peut bien entendu altérer la requête (après tout, tel est le rôle des intercepteurs). Pour ce faire, il est important de savoir qu'une requête est un objet immuable, et il nous faudra donc en faire un clone que l'on réassignera à chaque modification. Via l'utilisation de la méthode `.clone()`, il est possible de le modifier facilement. Cette méthode peut prendre en paramètre un objet qui altèrera chacune des propriétés spécifiées de notre objet initial avant d'en retourner une copie.

Via l'utilisation d'une condition au préalable, il est possible de ne faire intervenir notre intercepteur que sur certaines requêtes, par exemple celles qui ont dans leur URL une certaine chaîne de caractère, où celles de type POST, etc...

```
intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
    if (req.url.includes("blabla")) {
        const newReq = req.clone({
            headers: req.headers.append('API-KEY', "abcdefg")
        });

        return next.handle(newReq);
    }
    return next.handle(req);
}
```


Pousser plus loin les intercepteurs

Les intercepteurs ne sont pas cantonnés à la modification des requêtes, mais peuvent également être utilisés en cas de l'interception des réponses. Pour cela, il est important de savoir que notre intercepteur retourne un observable. Il est alors tout à fait possible de nous servir des opérateurs de **rxjs**, placés dans un pipe, pour traiter les événements que notre intercepteur va intercepter.

```
return next.handle(newReq)
  .pipe(tap(event => {
    if (event.type === HttpEventType.Response) {
      console.log("Une réponse est là :");
      console.log(event.body);
    }
  }));
```

Il est également possible de faire appel à plusieurs intercepteurs. Pour ce faire, il va d'abord falloir en créer un second. Une fois fait, il va nous falloir l'ajouter simplement à la liste des intercepteurs de notre module. Attention, l'ordre a son importance, en particulier si vos intercepteurs altèrent les requêtes et qu'un autre demande l'un des éléments altérés dans son code métier !

```
const newReq = req.clone({
  headers: req.headers.append('API-KEY', "abcdefg")
});
```

```
providers: [LoggingService,
{
  provide: HTTP_INTERCEPTORS,
  useClass: AuthInterceptorService,
  multi: true
},
{
  provide: HTTP_INTERCEPTORS,
  useClass: LoggingInterceptorService,
  multi: true
}]
```

TP : Weather App

@Utopios Consulting

Objectif

Maîtriser les bases de l'asynchronisme, des requêtes HTTP et des observables.

Sujet

En vous servant du framework Angular, vous devrez réaliser une application permettant à une personne d'accéder à la météo dans la ville de son choix pour les 5 prochains jours. Pour ce faire, vous vous servirez de l'API fournie par AccuWeather, dont l'adresse se trouve ci-dessous :

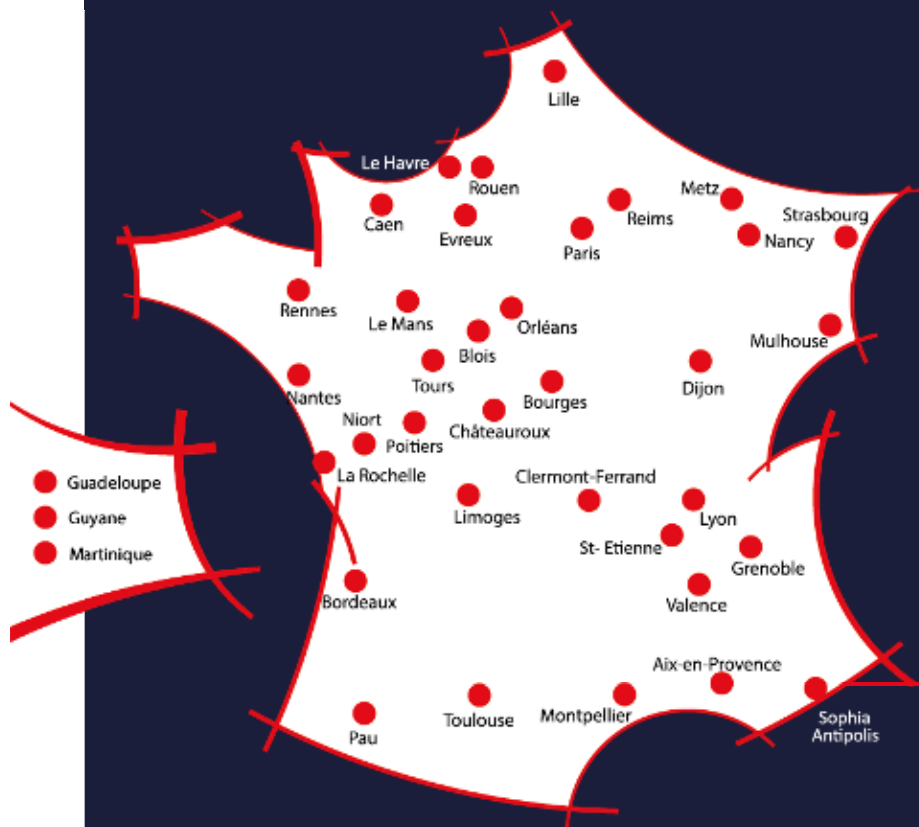
<https://developer.accuweather.com/>

Via l'utilisation de certains de ses endpoints, cette API va vous permettre d'accéder aux différentes localisations, que vous afficherez dans un premier temps dans votre application, en fonction d'une entrée textuelle.

Les localisations seront présentées dans un composant répété un maximum de 10 fois et qui contiendra par exemple le **nom de la commune**, son **descriptif** et le **pays** afin de différencier des emplacements portant des noms semblables.

Une fois ces localisations triées et affichées, vous ferez en sorte qu'un clic de l'utilisateur sur l'une de ces localités effectue une nouvelle requête à l'API pour récolter les informations de météo sur cette localité dans les 5 prochains jours.

Ces informations météo seront affichées dans un composant à part, qui sera répété 5 fois de sorte à obtenir une présentation homogène qui contiendra par exemple le **descriptif**, la **température maximale**, la **température minimale**, les **précipitations** et la **date**.



Découvrez également
l'ensemble des stages à votre disposition
sur notre site m2information.fr

m2information.fr

