Angular 2+

Préambule

Ce tutoriel va vous initier à Angular version 2 et +. En pratique, on va utiliser Angular 6, mais vous pouvez utiliser la version que vous voulez (aux évolutions du produit près).

Il s'agit d'une initiation : on ne va pas entrer dans des détails trop scabreux ; il existe plein de tutoriels pour faire des choses plus avancées et la documentation officielle d'Angular est plutôt bien faite.

Vue d'ensemble

Quelle est la différence entre Angular (et aussi VueJS, React, et d'autres) et les anciens frameworks web (comme SpringMVC) ?
Dans les anciens frameworks, les pages HTML étaient remplies d'une manière ou d'une autre avec des données par le serveur puis envoyées au browser. Le serveur était donc maître. En plus, pour passer d'une page à l'autre, il fallait passer par le serveur qui préparait une nouvelle page que chargeait le browser, ce qui n'était pas forcément très rapide. Le serveur pouvait être écrit dans n'importe quel(s) langage(s).
Dans les nouveaux frameworks, l'application est chargée une fois pour toute au démarrage. Tout est codé en JS : la navigation entre les pages, qui se fait sans passer par le serveur, est beaucoup plus souple. Le serveur ne sert qu'à fournir des WS que le client ira consulter pour récupérer des données. Le client est donc maître.

Préliminaires

Normalement, pour créer et gérer des projets Angular, on utilise la CLI Angular, CLI signifiant *Command Line Interface*, donc une série de commandes en ligne de commande (je soupçonne que les développeurs Linux ont poussé pour une interface aussi paléolithique).

Personnellement, je vais utiliser un IDE (basé sur Eclipse), Angular IDE. Mais comme il emploie au final la CLI, on va quand même jeter quelques coups d'œil sur la CLI.

Quoi qu'il en soit, il faut de toute façon installer NodeJS sur votre poste de travail pour faire fonctionner la CLI.

Ensuite on installe npm :

```
npm install -g npm@latest
```

npm est un gestionnaire de packages, similaire à apt sous Ubuntu.

Et La CLI de Angular :

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

Théorie de base

A priori, Angular fonctionne avec des composants. Un composant est un objet (au sens général, pas programmatique), et comme c'est un objet graphique (puisqu'il s'affiche), il est composé de 2 parties :

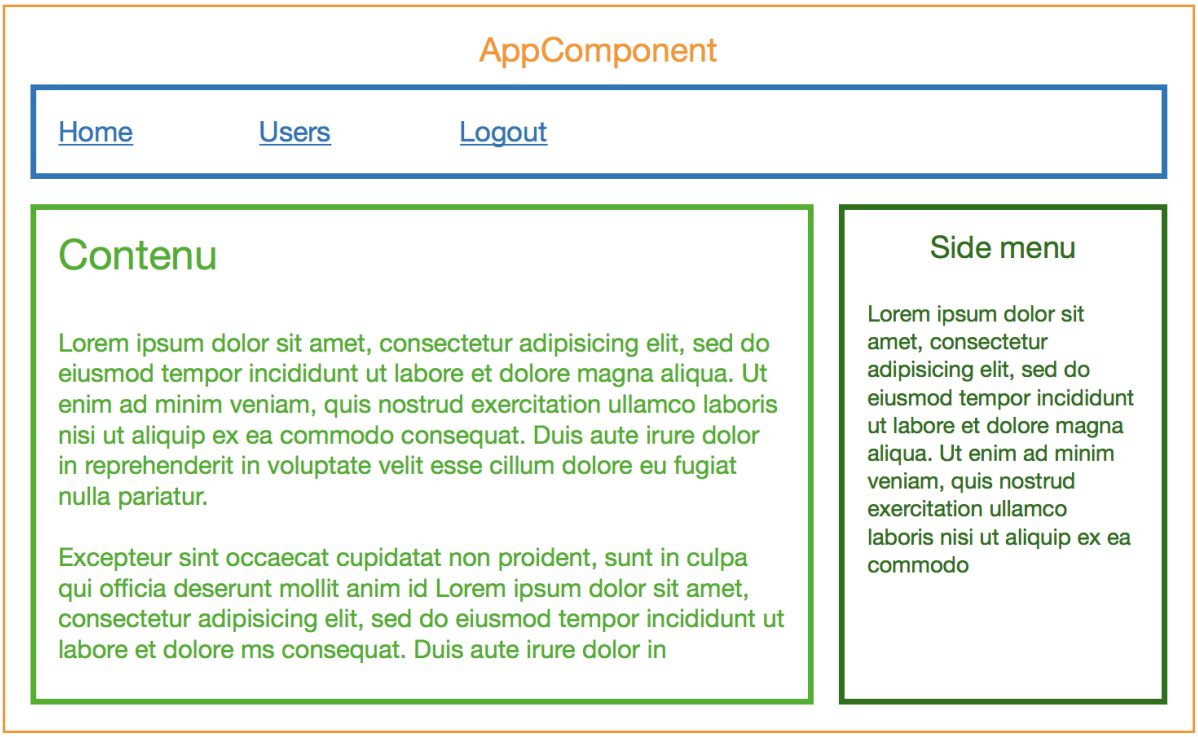
- La partie graphique (qui s'affiche)
- Et la partie traitement

La partie traitement est écrite en TypeScript (on y reviendra) et la partie graphique est elle-même composée de 2 parties :

- Une partie HTML
- Une partie CSS

En Angular, un composant est donc composé de 3 fichiers :

- Fichier TypeScript
- Fichier CSS
- Fichier HTML



Voyons l'exemple au-dessus. On a 4 composants à l'écran :

- Le contenu lui-même (en bas à gauche)
- Un menu (à droite)
- Un autre menu (en haut)
- Et le composant qui englobe les 3 précédents. Ce composant est particulier, c'est l'application, le point d'entrée d'une appli Angular.

Le TypeScript

Je ne vais pas faire un cours TypeScript, la documentation officielle est ici :
https://www.typescriptlang.org/docs/home.html

TypeScript est un langage « au dessus » de JS. En pratique Le TS est transcodé en JS puisque les browsers n'acceptent que le JS.

L'intérêt du TS est qu'il est plus « propre » que le JS, dans le sens qu'il est typé, qu'il possède nativement des Class, des interfaces, des lambda, etc …

Ce qui est troublant au début, c'est que TS ressemble à du Pascal. Si on déclare, par ex, une String et un nombre, ça donne :

```
machaine: string
index: number
```

idem pour une fonction

```
myFunc(ar: string[]) : number {

    …

  return 0 ;

}
```

Et une Class ressemble à ça.

```
export class Specie {
    id: number;
    latinName: string;
    commonName: string;
}
```

A noter que les methods d'une classe se réfèrent aux properties via un this (idem si une method se réfère à une autre method) :

```
export class PremierComponent implements OnInit {
    index = 0;
  constructor() { }

  ngOnInit() {
  }

  onClick() {
        this.index++ ;
  }
```

Pas d'affolement, Angular vous génère les squelettes des fichiers TS, vous n'aurez pas à partir de zéro.

A noter autre chose : les paramètres passés au constructeur, s'ils sont précédés de « private » ou « public » deviennent des propriétés de facto.

```
export class PremierComponent implements OnInit {
  constructor(private service: MyService ) { }

  ngOnInit() {
  }

  onClick() {
        this.service.myFunc() ;
  }
```

Si on veut déclarer une variable (dans une method), on n'utilise plus *var,* comme en JS, mais *let*. Et si c'est une constante, on met *const*.

```
onClick() {
    let var=1 ;
```

```
    const chaine=’ma chaine’
}
```

Prenons la classe Specie vue plus haut. Pour initialiser un objet Specie, on fait :

- Soit `o = new Specie() ;`
- Soit `o = {id: 1, commonName: 'Lapin', latinName: 'Lapinus'}` (on remarque que ça ressemble beaucoup à du JSON)

Pour accéder aux properties (ou aux methods), on fait :

- En lecture : `a = o.id`
- En écriture : `o.commonName = 'Vache'`

Une dernière chose. TS possède un système assez puissant pour le traitement des chaines. Si on écrit :
const str = 'gouda' ;
this.valeur = `Ce fromage est : ${str}`

Alors la property *valeur* vaudra : *Ce fromage est : gouda*.
Mais ATTENTION : pour que ça marche, il faut utiliser des ` (accent grave) et pas des '.

Ce qui était valable en JS l’est aussi en TS : les chaines sont plutôt délimitées par des ' plutôt que par des ", même si les " sont acceptées.

Créons un projet Angular

Plutôt que de continuer avec la théorie, passons à la pratique. Créons donc un projet. Appelons le *PremierProjet*.

Si on utilise la CLI, il faut faire :

```
ng new PremierProjet --skip-tests=true
```

Sinon, on fait *New > Project > Angular project* sous Angular IDE.

(Patientez, l’opération est longue, vu qu’Angular installe toutes les librairies de base)

Ne rentrons pas dans les détails, et voyons ce qu’il y a dans *src*. On y voit un fichier *index.html* qui sera affiché au démarrage dans le browser.

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>PremierProjet</title>
  <base href="/">

  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body>
  <app-root></app-root>
</body>
</html>
```

On voit qu’au final, il n’y a qu’une seule chose dans ce fichier : la balise *<app-root/>*. Cette balise appelle le composant maître, dont on parlait plus haut.

Lançons le serveur :

```
cd PremierProjet
ng serve --open
```

Ou *Run as > Angular Web Application*

On se connecte sur localhost :4200 et On voit un écran très laid. Mais ce qui est sûr c'est que tout ce truc est contenu dans le composant app-root.

Ouvrons app.component.html :

```
<div style="text-align:center">
  <h1>
    Welcome to {{ title }}!
  </h1>
  <img width="300" alt="Angular Logo"
src="
NTAgMjUwIj4KICAgIDxwYXRoIGZpbGw9IiNERDAwMzEiIGQ9Ik0xMjUgMzBMMzEuOSA2My4ybDE0LjIgMTI1LjFMMTI1IDIzMGw3OC4
5LTQzLjcgMTQuMi0xMjMuMXoiIC8+CiAgICA8cGF0aCBmaWxsPSIjQzMwMDJGIiBkPSJNMTI1IDMwdjIyLjItLjFWFWMjMwbDc4LjktND
MuNyAxNC4yLTEyMy4xTDEyNSAzMHoiIC8+CiAgICA8cGF0aCBmaWxsPSIjRkZGRiIgZD0iTTEyNSA1Mi4xTDY2LjggMTgyLjZoMTgyLjZoM
jEuN2wxMS43LTI5LjJoNDkuNGwxMS43IDI5LjJIMTgzTDEyNSA1Mi4xem0xNyA4My4zaC0zNGwxNy00MC45IDE3IDQwLjl6IiAvPgog
IDwvc3ZnPg==">
</div>
<h2>Here are some links to help you start: </h2>
<ul>
  <li>
    <h2><a target="_blank" rel="noopener" href="https://angular.io/tutorial">Tour of Heroes</a></h2>
  </li>
  <li>
    <h2><a target="_blank" rel="noopener" href="https://github.com/angular/angular-cli/wiki">CLI
Documentation</a></h2>
  </li>
  <li>
    <h2><a target="_blank" rel="noopener" href="https://blog.angular.io/">Angular blog</a></h2>
  </li>
</ul>
```

C'est bien ce qu'on voit à l'écran ! Remarquons le `{{ title }}` qui permet d'afficher la propriété *title* de la classe TS associée.

Voyons le code TypeScript :

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

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

Si on change la valeur de *title*, on voit la nouvelle valeur apparaître dans le browser. Remarquez-le @Component qui décrit les fichiers html et css associés ainsi que le selector, c.a.d le nom de la balise utilisée dans index.html.

Remarquez aussi le app.module.ts qui est une définition de module, qui en très gros, décrit tout ce qui est utilisé dans le programme courant. En particulier, il y est défini les autres modules (ici, comprendre : librairies) utilisés par le programme ainsi que les composants.

Les composants

Créons un composant :

`ng generate component premier` (ou utiliser l'IDE)

Allez dans *app.component.html*, virer tout ce qu'il y a dedans et mettez juste `<app-premier></app-premier>`

Si on regarde dans le browser, on voit `premier works !`

En fait *app-premier* est le selector (le nom) du composant *premier*. Comme on l'a mis dans le composant de l'application, on affiche donc le composant *premier*.

Dans le TS associé à *premier*, on ajoute une property de type string (disons *affichage*), on lui assigne une valeur et dans le HTML on remplace `premier works!` par {{affichage}}

Dans le browser on voit donc apparaitre la valeur de *affichage*. Cette façon de transférer des données du TS vers le HTML s'appelle *String Interpolation*.

Mais il y a moyen de faire des choses plus puissantes.

Property binding :

On peut faire réagir les properties du DOM (donc des objets HTML affichés) en fonction des valeurs du TS. On utilise les [] pour ça.
Mettons un <*span*> avec du texte dans le HTML.
Vérifiez que ça s'affiche dans le browser.
Créer une property *cacher* dans le TS et initialisez-la à false.
Changer le de la façon suivante : `<span [hidden]="cacher">`
Vérifiez que le texte s'affiche toujours.
Mettez *cacher* à true.
Vérifiez que le texte ne s'affiche pas.

Event Binding :

Cette fois on réagit aux évènement du DOM.
Enlevez le code précédent.
Rentrez dans le code :

```
<button (click)="onClick()">Appuyez</button>
Valeur : <span>{{ affichage }}</span>
```

On voit que la syntaxe est (handler)="fonction du TS à appeler". Il faut donc créer dans le TS, une fonction onClick() qui mettra à jour *affichage* de façon à ce que ça s'affiche. Et pour que ce soit un peu intéressant, faire en sorte que la fonction onClick() incrémente en plus un compteur pour *affichage*.

Le two-way Binding :

Comme son nom l'indique, c'est du binding dans les 2 sens : une valeur modifiée dans le HTML modifie une variable associée dans le TS et inversement. On utilise [()] pour ça.

Pour que ça marche, il faut déjà ajouter dans le module de l'application :

```
import { FormsModule } from '@angular/forms';
```

Et aussi FormsModule dans le array des imports un peu plus bas.

Dans le HTML du composant *premier*, on met :

```
<input type="text" [(ngModel)]="valeur"><br/>
<input type="text" [(ngModel)]="valeur"><br/>
```

(Oui, oui, c'est 2 fois la même chose)

Déclarez une property *valeur* dans le TS.

Dans le browser, vous voyez que les 2 champs d'édition se remplissent de la même chose. Que se passe-t-il ?
Si on rentre quelque chose dans le premier champ, *valeur* est mis à jour via le two-way Binding dans le TS. Et le 2ème champ est lui aussi mis à jour depuis TS.

Envoyer des données depuis l'exterieur

Ajouter dans le TS une property *extValue* et dans le HTML affichez *extValue*. Puis faites précéder la déclaration de *extValue* par `@Input()`. Le *@Input* signifie que *extValue* peut être initialisée par une valeur passée en paramètre au composant.

Si on retourne dans le HTML de l'application, on écrit `<app-premier extValue="Ceci est un essai">`
On teste dans le browser.

A noter qu'on peut aussi utiliser le property binding avec les []. Si on ajoute une property au TS de l'application (disons *appValue*), on peut écrire `<app-premier [extValue]="appValue">`

Les directives

En gros, ce sont des instructions qu'on met dans le HTML et qui permet de structurer le code. Par exemple pour faire des boucles ou mettre des directives conditionnelles.

A noter qu'il est possible de créer ses propres directives, mais on n'abordera pas le sujet.

ngIf

Comme son nom l'indique, cela permet de faire des conditionnelles et donc d'afficher ou pas des éléments du DOM.

`<span *ngIf="show">Je m'affiche</span>` (remarquez le * devant ngIf comme pour toutes les directives structurelles)

Ce qu'on voit là, c'est un texte qui va s'afficher ou non suivant la valeur de *show* (qui peut être un booléen, un nombre 0 ou 1, une chaine null ou qchose, etc). On peut évidemment mettre des conditions plus sophistiquées, allez voir la documentation d'Angular.

Faites le test en faisant varier la valeur de *show*.

ngFor

Cette fois on fait une boucle. La syntaxe est du genre : `*ngFor="let local_var of property"`

Ce n'est pas très clair ? Ok. Prenons un exemple.

Déclarons un tableau dans le TS (`chaines = ['AAA', 'BBB', 'CCC', 'DDD'];`)
Dans le HTML, mettons :

```
<ul>
<li *ngFor="let chaine of chaines">{{chaine}}</li>
```

```
</ul>
```

Remarquons qu'on met le *ngFor sur le <li> puisque c'est l'élément qu'on répète.

Il existe aussi des directives de style, permettant de positionner dynamiquement le style ou la Class dans le DOM, suivant le système de property binding (avec les [])

ngStyle

```
<h4 [ngStyle]="{color: getColor()}">
```

ngClass

```
<li [ngClass]="{'list-group-item': true,
                'list-group-item-success': appareilStatus === 'allumé',
                'list-group-item-danger': appareilStatus === 'éteint'}">
```

Pipes

Les pipes sont des moyens simples de modifier l'affichage. Comme le nom l'indique on utilise le signe pipe (|) pour envoyer les données à afficher dans un filtre. Il existe de nombreux pipes pré-définis dans Angular. On peut évidemment chainer les pipes : `{{data | filtre1 | filtre2}}`

Prenons un exemple simple et occupons-nous de *extValue* que nous affichions précédemment et transformons-le en majuscules : `{{extValue | uppercase}}`

Les pipes sont très intéressant pour les dates : créons une date dans le TS (`today = new Date()`) et affichons-la. L'affichage est correct mais pas très user friendly. Appliquons-lui le filtre *date*. C'est déjà plus élégant. On peut paramétrer *date* :

- Par exemple, avec short (date : `'short'`)
- Ou date : `'EEEE d MMMM y'`

Et on peut aussi chainer *date* avec *uppercase* (par exemple)

Il existe évidemment de nombreux autres filtres de pipe, reportez-vous à la documentation.

Services

En Angular, on fait comme en Java : on va déléguer le traitement à des services. D'une part, c'est plus propre, d'autre part le service peut être utilisé par d'autres composants. Je ne vais pas revenir sur la théorie du service étant donné qu'on l'a vue en long en large et en travers en Java.

Ce qu'il faut voir, c'est qu'Angular pratique l'injection de données à la Spring : les services déclarés dans l'application sont instanciés au démarrage de l'application, et il suffit de les passer au constructeur pour pouvoir en bénéficier. Si en plus, dans la déclaration du constructeur, on les fait précéder de *private* ou *public*, on les récupère en tant que properties implicites (voir la partie TS).

Pour créer un service :

```
ng generate service hero
```

ou faites *New > Service* dans l'IDE.

Créons le service *data.* Comme vous le remarquez, c'est une classe comme les autres. On remarque toutefois l'instruction :

```
@Injectable({
  providedIn: 'root'
})
```

Qui signifie que le service va être instancié au niveau de l'application et donc injectable dans tous les composants ET services de l'application (injection de dépendances à la Spring, donc).

Pour que ce soit intéressant, commençons par créer une Class *Machine* (*New > Other > Class*) dans *src/app*. Elle possède :

- Un id
- Une marque
- Un modèle

Une fois que c'est fait

- Créons un tableau de Machine dans le service (initialisé à quelque chose)
- Créons une fonction listMachines() dans le service qui renvoie le tableau
- Créons un tableau (vide) dans le composant *premier*
- N'oublions pas de passer le service *data* au constructeur du composant avec *private* devant (voir la partie TS)
- Dans la fonction ngOnInit() du composant (appelée tout de suite après le constructeur), on charge le tableau grâce à la fonction du service.
- Dans le HTML du composant, on affiche le tableau avec pour chaque ligne, quelque chose de la forme *modele* (*marque*)

Le Routing

Un des intérêts d'Angular, c'est qu'il est similaire à une application native : une fois le *index.html* chargé, on passe de page en page sans avoir à recharger quoi que ce soit (à part des données).

Malgré tout, il reste nécessaire de passer de page en page. Comment fait-on ?
En pratique le routing, c.a.d un ensemble de directives indiquant quel composant doit être appelé pour une url donnée, se trouve dans un seul fichier, ce qui permet aisément de s'y retrouver.

On pourrait définir le routing dans le module de l'application, mais, à mon sens, il est plus propre de créer un module à part.

Donc on crée un module nommé *app-routing* au même niveau que le module d'application.

```
ng g module app-routing –flat true
```

Ou avec l'IDE.

On copie ça dedans :

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';

const routes: Routes = [
];

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

```
  declarations: []
})
export class AppRoutingModule { }
```

C'est dans la partie surlignée en jaune qu'on fera les associations url/composant.

En plus, il faut déclarer ce module dans le module de l'application :

- Ajouter `import { AppRoutingModule } from './app-routing.module';` en haut
- Et ajouter *AppRoutingModule* dans le tableau des imports en dessous (l'auto-completion fait bien le boulot aussi).

A ce stade, on va dire qu'on va d'abord arriver sur une page contenant un lien, lequel nous redirigera sur la page dont on a l'habitude (*premier*).

Créons donc un composant *debut* qui ne contient qu'un lien :

```
<a routerLink="/premier">…</a>
```

On voit donc 2 choses :

- Avec ce lien, on a associé implicitement l'url */premier* au composant *premier*. Il va donc falloir déclarer cela dans le fichier de routing.
- Dans la balise <a, au lieu de mettre l'url derrière un *href*, on a mis un *routerLink.* Ce dernier est l'équivalent du *href* mais sans avoir à recharger la page. On doit donc l'employer.

Modifions le fichier de routing :

```
Const routes : Routes = [
{path : 'debut', component : DebutComponent},
{path : 'premier', component : PremierComponent},

] ;
```

On voit qu'on a un tableau d'associations url/composant. N'oubliez pas de faire l'import des composants si l'auto-complétion ne le fait pas.

ATTENTION : dans le champ *path*, on ne met jamais le / de début

Etant donné que *debut* est le premier composant à s'afficher, on va faire une redirection pour dire que c'est lui qui s'affiche par défaut.

On ajoute donc :

```
{ path: '', redirectTo: '/debut', pathMatch: 'full'},
```

C'est presque fini : dans le HTML du composant d'application, on remplace la balise *<app-premier>* par *<router-outlet>* pour signifier que cette partie-là de la page est prise en charge par le système de routing.

On teste maintenant que ça fonctionne.

Pour être sûr qu'on a bien compris, on va créer 2 autres composants :

- footer qui va contenir un texte bidon
- menu qui va contenir deux liens :
    - un vers debut
    - un vers premier
- On modifie le composant d'application pour positionner le menu et le footer dans le HTML.

On teste avec le menu que seule la partie centrale bouge.

Maintenant, on va faire une redirection de façon programmatique.
Dans le HTML de *debut*, on change le lien par un bouton qui fait un *(click)*. Dans la fonction appelée, on fait `this.router.navigate(['/premier']);` (on doit passer un array à *navigate()* qui est limité à 1 exemplaire dans ce cas). N'oubliez pas de passer la classe *Router* au constructeur.

On vérifie que ça marche.

Et on va terminer avec le classique passage de paramètres dans les url.

Revenons dans *premier*. Dans la liste des Machines, ce qui serait bien, pour chaque ligne, ce serait d'ajouter un lien vers les détails de la machine. On se doute que l'url va être quelque chose comme ça : */detail/8* (si 8 est l'id de la machine).
Commençons par créer le composant *detail*. Dans le HTML, faire 3 lignes pour afficher l'id, le modèle et la marque.

Dans le module de routage, ajoutons :

```
{ path: 'detail/:id', component: DetailComponent }
```

On voit bien que le *:id* va désigner ce qui est passé en paramètre à l'url (il peut évidemment y avoir plusieurs paramètres).

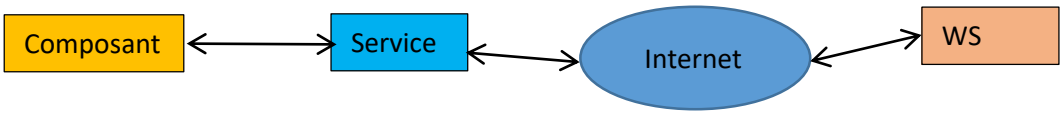
On modifie *premier* pour ajouter un lien à chaque ligne de la liste. Ensuite dans le *ngOnInit()* de *detail*, on va récupérer le paramètre : `const id = +this.route.snapshot.params['id'];`
Remarquez le + qui indique qu'on convertit une string en number. Quant à *route* il est passé en paramètre du constructeur et de type *ActivatedRoute*.
Ensuite, il faudrait récupérer la Machine dont le id vient d'être récupéré. Il faut donc déclarer dans le service *data* une fonction qui fait ça. Disons `getMachine(id: number ): Machine`. Pour info pour parcourir une boucle, c'est quelque chose comme ça : `for (const m of machines)` (avec *machines* qui est un array de Machine).
Une fois qu'on a fait ça, il ne reste plus dans le ngOnInit() de *detail* qu'à récupérer la Machine associé à l'id et à la stocker dans une property.

Dernier petit détail, on peut aussi décider de rajouter dans le HTML de *detail* un bouton pour faire un back. Pour ça on déclare une fonction goBack() dans le TS et qui fait : `this.location.back()`. *location* est de type *Location*.

Les Observables

C'est un point central d'Angular. Il gère en pratique les communications asynchrones.

Quand on se connecte, disons à un WS, les requêtes et les retours se font de manière asynchrone par rapport au corps du programme. Ce qui veut dire qu'on ne peut **pas** faire ceci :



On ne peut pas parce que le fonctionnement ci-dessus est SYNCHRONE.

En pratique, le service doit renvoyer un truc que l'on appelle un Observable qui attend qu'une opération asynchrone se termine avant de prévenir l'appelant.
En donc, le Service va renvoyer un Observable au Composant. Et le composant va faire un subscribe() auprès de l'Observable. Le subscribe() va dire à l'Observable que quand il aura fini son opération asynchrone, il appelera la fonction passée en paramètre au subscribe().
En fait, l'Observable appelle 3 fonctions qui peuvent être passées au subscribe().

- Une qui récupère les valeurs renvoyées par l'Observable
- Une qui récupère une erreur
- Une qui est appelée quand l'Observable a fini de bosser

Si on regarde ce que ça donne :

```
observable.subscribe(
    (value) => {console.log('data is = '+value) }, // première fonction
    (error) => {console.log('Uh-oh, an error occurred! : ' + error);}, // deuxième fonction
    () => {console.log('Observable complete!');} // troisième fonction

    );
```

Et on voit apparaître les lambdas …. Chaque lambda est donc une fonction passée au subscribe().
Ce qu'il y a avant le => désigne une fonction anonyme (elle n'a pas de nom) avec ou sans paramètre.
Ce qu'il y a après le => désigne le corps de la fonction.

Tout cela est très abstrait et n'aura d'application pratique que quand on verra les connections réelles aux WS JSON.

Mais on peut voir un peu mieux comment marche les Observables avec un exemple simple.

La fonction *interval(N)* de la library *RxJs* (celle qui fournit les Observables) renvoie un Observable et appelle la première fonction du *subscribe()* tous les N millisecondes.

Créons un composant *observer*. Dans le ngOnInit(), on crée un Observable de type *interval()* et on fait un *subscribe()* sur cet Observable avec seulement la première fonction (les deux autres ne sont jamais appelées).

```
const counter = interval(1000);
counter.subscribe(
  (value) => {this.seconds = value; },
);
```

Dans le HTML de *observer*, on affiche *seconds* et on voit qu'il s'incrémente de manière asynchrone (n'oubliez pas de modifier le routing, éventuellement).

A noter qu'il est possible de faire un *unsubscribe()* pour éviter de continuer à utiliser un Observable.

HttpClient

HttpClient est un objet Angular qui permet de faire des requêtes HTTP et qui renvoie un Observable.
Imaginons que HttpClient fasse un GET pour récupérer quelque chose auprès d'un WS. Il se passe ceci :

- HttpClient fait un GET via sa fonction get()
- get() renvoie un Observable (immédiatement sans attendre le retour du GET)
- L'appelant fait un subscribe() sur l'Observable retourné.

- Quand la requête revient, la (première) fonction passée dans le subscribe() est appelée avec les données en retour du GET.

A priori HttpClient ne gère pas que les données en JSON, mais on ne va s'occuper que de cela pour simplifier. D'autant que la conversion JSON/TS est automatique. Et, pour simplifier, on supposera que les url sont de type REST.

On a les fonctions suivantes pour HttpClient :

- get(url) :Observable
- delete(url):Observable (Attention, l'Observable n'appelle pas la fonction de données)
- post(url,objet,options): Observable
- put(url,objet,options): Observable

(Il y en a d'autres, évidemment et les paramètres peuvent changer, mais on donne ici une version simplifiée)

Je ne l'ai pas précisé au chapitre précédent mais les Observables et le TS en général sont typés. Aussi si le *get()* renvoie un array de *Machine*, il s'écrit *get<Machine[]>()* et renvoie un *Observable<Machine[]>*.

Au fait que sont les *options* qu'on a vues plus haut ? Eh bien ce sont des options qu'on transmet dans la requête HTTP, typiquement des champs du header HTTP. Par exemple si on fait un POST ou un PUT en JSON, il faut ajouter dans le header, le fait qu'on envoie du JSON. Ça donne :

```
httpOptions = {
    headers: new HttpHeaders({ 'Content-Type': 'application/json' })
  };
```

Pour utiliser HttpClient, avant toute chose, il faut ajouter dans le module d'application :
`import { HttpClientModule } from '@angular/common/http';`
Et mettre HttpClientModule dans l'array des imports

Dans le répertoire *assets*, on crée un fichier *machines.json* et on met dedans :
`[{"id": 0, "modele": "Galaxy S7", "marque": "Samsung"}, {"id": 1, "modele": "Frigo B1", "marque": "Brandt"},{"id": 2, "modele": "Mac Book", "marque": "Apple"}]`
(Ce fichier va simuler un WS)

Ensuite, on va dans le service *data*. On modifie la fonction *listeMachines*, de la façon suivante :

```
On modifie donc le retour de la fonction : elle ne retourne plus directement un Machine[] mais un Observable qui renvoie à la première fonction des données de type Machine[]
  listMachines(): Observable<Machine[]> {
    On appelle donc la fonction get() de HttpClient qui appelle le fichier vu plus haut.
    return this.http.get<Machine[]>('/assets/machines.json');
  }
```

N'oubliez pas de passer HttpClient dans le constructeur. Il peut arriver que l'auto-complétion n'insère pas le bon import. Ce doit être :
`import { HttpClient } from '@angular/common/http';`

Ensuite on modifie l'appel dans le composant *premier* :

```
On appelle donc listMachines(), la fonction du service data. Cette dernière renvoie un Observable sur lequel il faut souscrire. On fait donc un subscribe() avec juste la fonction de données (lesquelles sont de type Machines[] vu la déclaration de la fonction dans le service.
this.service.listMachines().subscribe(machines => {this.machines = machines; });
```

On vérifie que ça marche.

Ensuite pour tester la gestion des erreurs :

- On modifie l'url dans *data* de façon à ce qu'elle soit fausse
- On modifie l'appel de listMachine() de façon à ajouter la fonction d'erreur dans le subscribe()
- On stocke l'erreur dans une property
- On affiche cette erreur (avec un ngIf) dans le HTML de *premier*.

On peut faire plein d'autre choses avec les Observables et/ou HttpClient, comme une gestion d'erreur centralisée ou insérer des fonctions entre l'Observable et l'appelant, etc …. On peut se référer au chapitre « Pour aller plus loin » pour plus de renseignements.

Un dernier point auquel vous serez confronté : pour l'exercice plus haut, on a mis le fichier JSON au même endroit que l'appli Angular. Ce qui est un peu idiot, vu que, généralement les WS seront sur d'autres serveurs.

Le problème c'est que pour des raisons de sécurité, un programme Angular lancé sur un certain serveur (ici *localhost:4200*) n'a pas le droit de se connecter à un autre serveur (par ex : *localhost:8080*). C'est ce qu'on appelle le CORS (Cross-origin resource sharing).

En pratique il faut sur le serveur distant (celui du WS) spécifier qu'on accepte les connections d'un autre serveur (ici *localhost:4200*).

Pour aller plus loin

Ce cours n'était qu'une initiation à Angular volontairement allégée pour éviter que les stagiaires se noient. On peut continuer avec :

- Le tutorial officiel d'Angular que je trouve très bien fait : https://angular.io/tutorial
- Le tutorial d'OpenClassRoom qui est à mon sens assez incompréhensible en tant que tutoriel mais qui peut apporter des informations complémentaires pour peu qu'on ait quelques connaissances de base en Angular : https://openclassrooms.com/fr/courses/4668271-developpez-des-applications-web-avec-angular