

Créer une application avec Angular



Dans [un autre chapitre](#), nous avons présenté Angular et donné un aperçu de son fonctionnement. Dans ce chapitre, nous allons aller plus loin en construisant notre première application Angular.

Pour ce tutoriel, nous reprendrons l'application démarrée dans le chapitre portant sur les bases. Clonez le master de l'application :

```
git clone git@github.com:Cevantime/tour-of-heroes-typescript.git
```

Pour avoir une idée de l'objectif, vous pouvez vous placer sur la branche "angular-only".

Nous suivrons dans les grandes lignes [le tutoriel proposé par Google](#) en le personnalisant un petit peu.

I. Travail sur les composants

1) Création d'un composant avec Angular cli

Dans le chapitre précédent, nous avons affiché une liste de héros directement au sein de notre composant app. Ce n'est pas une très bonne pratique car le code que nous avons produit n'est pas ré-exploitable à un autre endroit de mon application. Pour rendre notre liste de héros réutilisable dans un autre contexte, le mieux est de l'isoler sous la forme d'un composant.

Un composant regroupe trois fichiers (voire davantage), ce qui peut être fastidieux à créer. Pour générer ce composant, le plus rapide est d'utiliser Angular CLI. Entrez la commande suivante :

```
ng generate component heroes-list
```

Cela nous génère un dossier à l'intérieur du répertoire `src/app` contenant les trois fichiers de notre composant + un fichier de spécification de test. Pour l'instant ces fichiers n'ont que le code minimal nécessaire au bon fonctionnement du composant.

Angular CLI ne s'est pas contenté de générer ces trois fichiers : il a également ajouté notre composant à la liste des déclarations du module *app.module.ts*. C'est ainsi que le programmeur doit indiquer à Angular que notre nouveau composant appartient bien au module de notre application

```
@NgModule({
  declarations: [
    AppComponent,
    HeroesListComponent // cette ligne a été ajoutée suite à notre commande
  ],
  imports: [
    BrowserModule,
    AppRoutingModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

2) Utilisation de notre nouveau composant

Pour utiliser notre composant dans notre module, nous allons l'insérer dans le template racine de notre application *app.component.html*. Pour ce faire nous n'avons qu'à inclure la **balise correspondant à notre composant**. Pour savoir quelle est cette balise : nous devons nous rendre dans *src/app/heroes-list/heroes-list.component.ts* nouvellement généré :

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

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

  constructor() { }

  ngOnInit() {
  }

}
```

Ici la propriété **selector** nous apprend que la balise correspondant à notre composant est `app-heroes-list`.

Ajoutons cette balise dans *src/app/app.component.html*:

```
<h1>Bienvenue sur {{ title }}</h1>

<app-heroes-list></app-heroes-list>
```

Nous devrions obtenir :

Bienvenue sur La tour des héros

heroes-list works!

"Heroes-list works" correspond bien au contenu du template de notre nouveau composant (cf.

`src/app/heroes-list/heroes-list.component.html`).

3) Listing de nos héros

Dans notre composant, nous voulons maintenant afficher une liste de héros. Nous voulons que cette liste soit gérée côté Typescript pour être dynamique. Ajoutons donc un tableau contenant nos différents héros dans le Typescript de notre composant. Nos héros ne sont pas très complexes : ils possèdent un id et un nom :

```
export class HeroesListComponent implements OnInit {  
  
  heroes = [  
    { id : 1, name : 'Batman' },  
    { id : 2, name : 'Superman' },  
    { id : 3, name : 'Spiderman' },  
  ];  
  // ...  
}
```

Si peu complexes qu'ils soient, nos héros possèdent une structure commune, laquelle définit un type. Déclarons ce type dans une classe que nous importerons dans notre composant. Pour rester organisé, nous allons créer un dossier *types* dans *src/app* qui contiendra cette classe Hero. Créons donc notre fichier *src/app/types/hero.type.ts* dans notre application avec le code suivant :

```
export default class Hero {  
  id: number;  
  name: string;  
}
```

Et utilisons cette classe dans notre script.

```
// ...  
import Hero from '../types/hero.type';  
  
// ...  
  
heroes: Hero[] = [  
  { id : 1, name : 'Batman' },  
  { id : 2, name : 'Superman' },  
  { id : 3, name : 'Spiderman' },  
];
```

Et maintenant, affichons notre liste de héros et leurs id associés dans notre template !

```
<ul class="heroes">  
  <li *ngFor="let hero of heroes"><span class="badge">{{ hero.id }}</span> {{ hero.name }}</li>  
</ul>
```

4) Appliquer des styles

Remarquez que nous avons ajouté quelques classes CSS propres à notre composant (*heroes* et *badges*) nous allons styliser notre composant en utilisant le fichier *heroes-list.component.css*. Cela nous permet de n'appliquer ces règles qu'à l'intérieur du composant:

```

/* HeroesComponent's private CSS styles */

.selected {
  background-color: #CFD8DC !important;
  color: white;
}

.heroes {
  margin: 0 0 2em 0;
  list-style-type: none;
  padding: 0;
  width: 15em;
}

.heroes li {
  cursor: pointer;
  position: relative;
  left: 0;
  background-color: #EEE;
  margin: .5em;
  padding: .3em 0;
  height: 1.6em;
  border-radius: 4px;
}

.heroes li.selected:hover {
  background-color: #BBD8DC !important;
  color: white;
}

.heroes li:hover {
  color: #607D8B;
  background-color: #DDD;
  left: .1em;
}

.heroes .text {
  position: relative;
  top: -3px;
}

.heroes .badge {
  display: inline-block;
  font-size: small;
  color: white;
  padding: 0.8em 0.7em 0 0.7em;
  background-color: #607D8B;
  line-height: 1em;
  position: relative;
  left: -1px;
  top: -4px;
  height: 1.8em;
  margin-right: .8em;
  border-radius: 4px 0 0 4px;
}

```

Nous obtenons :

Bienvenue sur La tour des héros

1 Batman

2 Superman

3 Spiderman

Si nous voulons par contre changer des styles globaux, tels que la police de caractères, nous le ferons dans le fichier *src/style.css*:

```
/* Application-wide Styles */

h1 {
  color: #369;
  font-family: Arial, Helvetica, sans-serif;
  font-size: 250%;
}

h2, h3 {
  color: #444;
  font-family: Arial, Helvetica, sans-serif;
  font-weight: lighter;
}

body {
  margin: 2em;
}

body, input[type="text"], button {
  color: #888;
  font-family: Cambria, Georgia;
}

a {
  margin-top: 20px;
  font-family: Arial;
  background-color: #eee;
  border: none;
  cursor: pointer;
  cursor: hand;
  padding: 5px 10px;
  border-radius: 4px;
  margin: 3rem 0rem;
  color: #607D8B;
  text-decoration: none;
}

a:hover {
  background-color: #cfd8dc;
}

/* everywhere else */

* {
  font-family: Arial, Helvetica, sans-serif;
}
```

avec pour résultat :

Bienvenue sur La tour des héros

1 Batman

2 Superman

3 Spiderman

5) Afficher les détails d'un héros à la sélection

Lorsque nous cliquons sur un héros, nous aimerions pouvoir afficher ses informations détaillées.

Pour cela, notre script doit réagir au clic sur un des héros de ma liste. Nous devons donc mettre en place un lien template -> script et utiliser les parenthèses `()`. Nous réagissons à l'évènement click appliqué à notre élément de liste `li`. Lorsque notre héros sera sélectionné, nous stockerons dans notre classe le héros sélectionné et afficherons ses détails.

Créons une variable dans notre script pour contenir le héros sélectionné :

```
// ...
export class HeroesListComponent implements OnInit {

  heroes: Hero[] = [
    { id : 1, name : 'Batman' },
    { id : 2, name : 'Superman' },
    { id : 3, name : 'Spiderman' },
  ];

  selectedHero: Hero;

  constructor() { }

  //...
```

Nous exécuterons une méthode `selectHero` qui prendra en paramètre le héros à sélectionner. Codons cette fonction :

```
// ...
constructor() { }

selectHero(hero: Hero) {
  this.selectedHero = hero;
}

// ...
```

Et maintenant, faisons appel à cette fonction lors du clic sur un héros

```
<ul class="heroes">
  <li *ngFor="let hero of heroes" (click)="selectHero(hero)"><span class="badge">{{ hero
.id }}</span> {{ hero.name }}</li>
</ul>
```

De plus, pour marquer la sélection visuellement, nous ajoutons une classe `selected` à notre héros lorsque celui est sélectionné. Pour cela, nous utilisons le binding script -> template à l'aide

des crochets `[]` :

```
<ul class="heroes">
  <li *ngFor="let hero of heroes" [class.selected]="selectedHero == hero" (click)="selectHero(hero)">
    <span class="badge">{{ hero.id }}</span> {{ hero.name }}
  </li>
</ul>
```

La sélection est maintenant active. Affichons les détails du héros sélectionné. Nous n'afficherons ces détails que si nous avons un héros sélectionné. Donc nous devons utiliser la directive structurelle `*ngIf` :

```
<ul class="heroes">
  <li *ngFor="let hero of heroes" [class.selected]="selectedHero == hero" (click)="selectHero(hero)">
    <span class="badge">{{ hero.id }}</span> {{ hero.name }}
  </li>
</ul>
<div *ngIf="selectedHero">
  <h2>Détails de {{selectedHero.name | uppercase}} : </h2>
  <div><span>id: </span>{{selectedHero.id}}</div>
</div>
```

Nous obtenons :

Bienvenue sur La tour des héros

- 1 Batman
- 2 Superman
- 3 Spiderman



6) Isoler une portion de code dans un nouveau composant

Il est possible (et même probable) que le code du détail de notre héros soit réutilisé à d'autres endroits de notre application. Pour rendre réutilisable ce code, nous allons l'isoler dans un composant. Créons ce composant :

```
ng generate component hero-details
```

Le code de notre composant est généré dans un dossier `src/app/hero-details` aux seins des 4 fichiers habituels. Transférons le code HTML du détail de notre héros dans `src/app/hero-details/hero-details.component.html`. Sachant que notre composant est sensé être réutilisé dans n'importe quel contexte, nous renommerons `selectedHero` en simplement `hero` afin d'avoir une appellation plus neutre.

```
<h2>Détails de {{hero.name | uppercase}} : </h2>
<div><span>id: </span>{{hero.id}}</div>
```

Nous supprimons également le *ngIf qui sera géré autre part au moment de l'insertion de notre composant. La variable `hero` n'existe pas encore dans notre script, créons-là :

```
import { Component, OnInit } from '@angular/core';
import Hero from '../types/hero.type';

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

  hero: Hero;

  constructor() { }

  ngOnInit() {
  }

}
```

Dans notre template `heroes-list.component.html`, faisons appel à notre composant. Nous ne l'afficherons que si un héros est sélectionné grâce à *ngIf.

```
<ul class="heroes">
  <li *ngFor="let hero of heroes" [class.selected]="selectedHero == hero" (click)="selectHero(hero)">
    <span class="badge">{{ hero.id }}</span> {{ hero.name }}
  </li>
</ul>
<app-hero-details *ngIf="selectedHero"></app-hero-details>
```

Si vous testez votre application maintenant, cela ne marche pas. Nous avons en effet un problème avec notre composant `app-hero-detail` qui ne sait pas quel héros il doit afficher. Pour résoudre ce problème, nous avons deux choses à faire :

- Nous devons dire à notre composant `app-hero-detail` que son champs `hero` sera passé en entrée depuis l'extérieur (en l'occurrence, depuis le composant `app-heroes-list`). Pour cela, nous utilisons l'annotation `@Input` devant `hero` (sans oublier de l'importer) :

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

// ...

export class HeroDetailsComponent implements OnInit {

  @Input() hero: Hero;

  // ...

}
```

- Nous devons passer le héros sélectionné depuis notre composant `app-heroes-list` vers `app-hero-details`. Pour cela, nous utilisons les crochets pour accéder au `hero` de `app-hero-details` :

```
<app-hero-details *ngIf="selectedHero" [hero]="selectedHero"></app-hero-details>
```

De cette façon, notre héros est bien transmis de `app-heroes-list` vers `app-hero-details`.

II. Service et Routing

Nous avons bien avancé mais notre application peut encore faire mieux.

1) Navigation sur plusieurs "pages"

Jusqu'à présent, tous nos composants s'exécutent sur la même "page", c'est à dire sur la même URL.

Voyons maintenant comment naviguer entre nos pages.

a. 1 page = 1 composant

Dans une application Angular, la manière la plus classique de créer une page est de créer simplement un composant qui contiendra le template propre à cette page. Nous aurons en fait notre composant app dans lequel viendra s'encaster un composant dynamique qui changera d'une page à l'autre.

b. Le router

Pour gérer ce dynamisme et cette navigation, nous devons faire plusieurs choses mais la plus importante est de configurer notre routeur de façon à associer une URL à un composant. Nous voulons ici que notre liste de héros soit associée à l'URL de base qui est à la racine de notre site. Pour configurer cela, nous allons dans le fichier `src/app/app-routing.module.ts`. Il s'agit en fait d'un module dans lequel nous pouvons déclarer nos routes, c'est-à-dire nos associations URL <-> page. Allons-y :

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { HeroesListComponent } from '../heroes-list/heroes-list.component';

const routes: Routes = [
  {
    path: '',
    component: HeroesListComponent
  }
];

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

Comme vous le voyez, nous définissons une route associée au composant `HeroesListComponent` dont le chemin est... vide. Nous ne devons renseigner en fait que la partie dynamique de l'URL c'est-à-dire celle qui sera amenée à changer au cours de notre navigation sur le site. La racine de notre site correspond donc à un chemin vide.

c. La directive router-outlet

Pour rendre dynamique l'affichage, il ne faut plus placer `app-heroes-list` directement dans `app-root` mais le remplacer par un pseudo-composant appelé **router-outlet**, lequel va définir la position à laquelle composant dynamique doit être inséré. Plaçons dans `app.component.html` :

```
<h1>Bienvenue sur {{ title }}</h1>

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

d. Ajout d'un composant d'édition de héros

Bien, nous avons un composant qui est susceptible de changer en fonction de la page visitée. Testons cela en créant un nouveau composant qui sera consacré à l'édition d'un héros.

```
ng generate component hero-edit
```

Pour naviguer vers notre composant, déclarons une route qui pointe vers lui.

```
// ...
import { HeroEditComponent } from './hero-edit/hero-edit.component';
const routes: Routes = [
  {
    path: '',
    component: HeroesListComponent
  },
  {
    path: 'edit',
    component: HeroEditComponent
  }
];
// ...
```

Si vous saisissez l'URL localhost:4200/edit dans votre navigateur, notre composant s'affiche bien en lieu et place de notre liste de héros.

Bienvenue sur La tour des héros

hero-edit works!

2) Éditer le bon héros

a. Ajouter un paramètre dynamique dans notre route

Notre route n'est pas encore complète. En effet, pour éditer un héros, il faut passer une information concernant l'identité du héros dans l'URL et la prendre en compte dans notre document pour éditer le héros voulu. Nous allons ajouter dans notre chemin un paramètre dynamique qui est l'id du héros que nous souhaitons éditer.

```
{
  path: 'edit/:id',
  component: HeroEditComponent
}
```

Le paramètre 'id' est précédé de deux points pour indiquer à angular qu'il peut -qu'il va!- changer

b. Récupération de la valeur du paramètre

Pour récupérer la valeur de cet id dans le script de notre composant d'édition, nous allons devoir récupérer l'objet qui représente notre route dans notre application. Cette route est récupérable dans Angular grâce à ce que l'on appelle l'**injection de dépendance**. Nous verrons cela plus loin. Pour l'instant, sachez simplement qu'on peut exiger l'injection de notre route dans le constructeur de notre composant en le modifiant simplement comme ceci :

```
constructor(private route: ActivatedRoute) { }
```

Ainsi notre script aura notre route disponible sous la forme de `this.route`. Ainsi, nous pouvons récupérer l'id du héros comme ceci :

```
//...
id;

constructor(private route: ActivatedRoute) {
  this.id = route.snapshot paramMap.get('id');
}
// ...
```

Notre route active est susceptible de changer à tout moment, nous en créons donc une capture (un snapshot) et nous allons chercher l'id dans les paramètres. Ainsi, nous pouvons afficher cet id dans notre template `hero-edit.component.html`.

Pour avoir un résultat, nous saisissons `localhost/edit/2` dans la barre d'adresse et nous obtenons :

Bienvenue sur La tour des héros

Édition du héros n°2

c. Création d'un service de héros

Nous souhaitons maintenant récupérer le héros correspondant à l'id que nous récupérons. Le problème est que la liste contenant nos héros n'est présente que dans le composant `app-heroes-list` et qu'il n'y aurait pas de sens à utiliser ce composant dans `app-hero-edit`. Pour résoudre ce problème, nous allons isoler notre liste de héros dans une classe qui sera **utilisable dans nos autres classes**. Ce faisant, nous créons notre **premier service** :

```
ng generate service hero
```

Notre service `hero.service.ts` est créé dans `src/app` ainsi que le fichier de test associé.

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

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

  constructor() { }
}
```

Notez que notre service possède une annotation `@Injectable` qui le rend injectable dans n'importe quelle classe déclarée dans mon application (voir plus loin)

Ajoutons notre liste de héros à notre service ainsi que deux méthodes :

- une méthode `getHeroes` qui retournera notre liste de héros
- une méthode `getHeroById` qui retournera un héros grâce à son id

```
export class HeroService {

  private heroes: Hero[] = [
    { id : 1, name : 'Batman' },
    { id : 2, name : 'Superman' },
    { id : 3, name : 'Spiderman' },
  ];

  constructor() { }

  getHeroes(): Hero[] {
    return this.heroes;
  }

  getHeroById(id: number): Hero {
    return this.heroes.find((hero) => hero.id === id);
  }
}
```

d. Injection de notre service dans nos composants

Afin d'utiliser notre service dans nos composants, nous allons nous servir d'une fonctionnalité très pratique d'Angular qui est l'*injection de dépendance*. Angular est en mesure d'instancier lui-même notre service et de le passer à d'autres classes de l'application si celles-ci le passent en paramètre de leur **constructeur**. Angular va alors se référer au **type** demandé en paramètre pour injecter le bon service :

Mettons cela en pratique dans notre composant app-heroes-list qui n'a désormais plus besoin d'héberger la liste des héros :

```
// ...

export class HeroesListComponent implements OnInit {

  selectedHero: Hero;
  heroes: Hero[];

  constructor(private heroService: HeroService) { }

  // ...
}
```

Notre liste de héros démarre vide. Pour l'initialiser, il faudra faire appel à notre service. Un endroit parfait pour initialiser notre liste est dans la méthode `ngOnInit()` qui s'exécute à l'initialisation du composant :

```
ngOnInit() {
  this.heroes = this.heroService.getHeroes();
}
```

De même, injectons notre service dans le script du composant app-hero-edit pour récupérer le héros correspondant à l'id de la route. Il n'est plus utile de récupérer l'id et de le stocker, nous récupérerons le héros directement !

```
// ...
export class HeroEditComponent implements OnInit {

  hero;

  constructor(private route: ActivatedRoute, private heroService: HeroService) { }

  // ...
}
```

Initialisons le héros de ngOnInit :

```
ngOnInit() {  
  const id = Number(this.route.snapshot.paramMap.get('id'));  
  this.hero = this.heroService.getHeroById(id);  
}
```

Nous devons aussi mettre à jour notre template hero-edit.component.html en conséquence :

```
<h2>Édition du héros {{ hero.name }}</h2>
```

Nous obtenons :

Bienvenue sur La tour des héros

Édition du héros Superman

Nous récupérons bien notre héros. Il ne reste plus qu'à l'éditer au moyen d'un champs associé au nom de notre héros :

```
<h2>Édition du héros {{ hero.name }}</h2>  
  
<form>  
  <p>  
    <label>name:  
      <input [(ngModel)]="hero.name" placeholder="name">  
    </label>  
  </p>  
  <p>  
    <input type="submit" value="Éditer">  
  </p>  
</form>
```

Nous pouvons maintenant éditer notre héros et les changements sont automatiquement reflétés dans le DOM.

Bienvenue sur La tour des héros

Édition du héros Batman

name:

3) Naviguer entre nos routes

a. Écrire un lien simple

Nous voulons maintenant pouvoir naviguer entre nos deux pages. Du côté de la liste des héros, nous voulons ajouter un bouton "Modifier" dans les détails du héros de façon à pouvoir accéder à la page d'édition sans avoir à entrer une URL à la main.

Pour cela, nous allons créer un lien muni de la directive `routerLink` qui remplace le href dans Angular. Rendez-vous dans hero-details.component.html et ajoutez ce lien :

```
<h2>Détails de {{hero.name | uppercase}} : </h2>
<div>
  <span>id: </span>{{hero.id}}
  <a routerLink="/edit/{{ hero.id }}">Modifier {{ hero.name }}</a>
</div>
```

Maintenant, lorsque l'on clique sur le lien nous sommes bien dirigés vers la bonne page. C'est encore plus joli avec un peu de css dans hero-detail.component.css :

```
/* HeroDetailComponent's private CSS styles */

label {
  display: inline-block;
  width: 3em;
  margin: .5em 0;
  color: #607D8B;
  font-weight: bold;
}
```

b. Revenir en arrière dans l'historique

Maintenant, sur la page d'édition, nous allons ajouter un bouton permettant de revenir en arrière dans l'historique de navigation. Cela se fait en injectant dans notre script hero-edit.component.ts un objet de type Location.

```
import { Location } from '@angular/common';

// ...

constructor(private location: Location) { }
```

Cet objet location possède une méthode **back** qui nous permet de revenir en arrière dans l'historique. Créons une méthode goBack() dans notre composant et utilisons-la dans notre template :

```
goBack() {
  this.location.back();
}
```

et

```
<a (click)="goBack()">Revenir en arrière</a>
```

c. Aller vers une adresse depuis le code grâce au routeur

Nous souhaitons également le rediriger vers la liste des héros lorsque notre utilisateur sauvegarde ses changements. Pour cela, nous utiliserons le routeur d'Angular, injectable lui aussi, et possédant une méthode **navigate** :

```
constructor(
  private route: ActivatedRoute,
  private heroService: HeroService,
  private location: Location,
  private router: Router) { }
```

Créons une méthode **save()** qui utilise cette méthode navigate :

```
save() {  
  this.router.navigate(['/']);  
}
```


Navigate prend en paramètre non pas une chaîne mais un tableau contenant les différents segments de notre chemin.

Appelons-la lors de la soumission du formulaire !

```
<!-- ... -->  
<form (submit)="save()">  
<!-- ... -->
```

Notre navigation fonctionne !

Bienvenue sur La tour des héros

- 1 Batman | 
- 2 Superman
- 3 Spiderman

Détails de BATMAN :

id: 1 [Modifier Batman](#)

III. Utiliser une vraie fausse API

1) Installation de notre vrai fausse api

Tout semble bien fonctionner dans notre application. Seulement, dans un cas réel, nos données transitent sur un serveur et sont sauvegardées au moyen d'une api. Nous devons dialoguer avec cette **api** qui stocke les données dans une base de données. Pour les besoins de notre cours, nous utiliserons une fausse api qui enregistrera les données dans la mémoire de votre ordinateur et non sur un serveur distant (ça nous évitera de faire des bêtises!)

Installons cette fausse api.

```
npm install angular-in-memory-web-api --save
```

Nous créons également un service prenant en charge notre fausse base de données :

```
import { InMemoryDbService } from 'angular-in-memory-web-api';
import Hero from './types/hero.type';
import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'root',
})
export class InMemoryDataService implements InMemoryDbService {
  createDb() {
    const heroes = [
      { id: 11, name: 'Batman' },
      { id: 12, name: 'Superman' },
      { id: 13, name: 'Catwoman' },
      { id: 14, name: 'Ironman' },
      { id: 15, name: 'Prettywoman' },
      { id: 16, name: 'Gentleman' },
      { id: 17, name: 'Megaman' },
      { id: 18, name: 'Magneto' },
      { id: 19, name: 'Adama' },
      { id: 20, name: 'Wolverine' }
    ];
    return { heroes };
  }

  genId(heroes: Hero[]): number {
    return heroes.length > 0 ? Math.max(...heroes.map(hero => hero.id)) + 1 : 11;
  }
}
```

Notre service, pour dialoguer avec notre fausse base de données, doit implémenter InMemoryDbService. et fournir une base de donnée ainsi qu'une manière de générer de nouveaux id. Inutile de trop s'attarder trop sur ce code qui n'est pas l'essentiel du sujet.

Notre module app.module.ts doit également être configuré pour prendre en compte le module :

```
// ...
import { HttpClientModule } from '@angular/common/http';
import { HttpClientInMemoryWebApiModule } from 'angular-in-memory-web-api';
import { InMemoryDataService } from './in-memory-data.service';

// ...
imports: [
  // ...
  HttpClientModule,
  HttpClientInMemoryWebApiModule.forRoot(
    InMemoryDataService, { dataEncapsulation: false }
  ),
  // ...
],
//...
```

2) Appel à l'API dans notre service de héros

a. Le service Http

Ce n'est plus à notre service d'héberger notre "base de données" (notre liste de héros, en fait) mais à notre API. Remplaçons donc notre liste de héros par l'url de l'API :


```
// ...
export class HeroService {

    private heroesUrl = 'api/heroes';

    constructor() { }

    // ...
}
```

Nous ne pouvons plus utiliser notre tableau directement dans nos méthodes `getHeroes` et `getHeroById`. Pour obtenir nos données, nous devons faire une requête http vers notre api. Pour cela, nous allons utiliser un service fourni par angular et que nous avons ajouté à notre module dans le 1) : le client Http. Injectons-le dans le constructeur de notre service :

```
constructor(private http: HttpClient) { }
```

b. Les observables

Le service `HttpClient` va effectuer notre requête à l'API. Pour ne pas bloquer notre application pendant l'envoi et la réception de ces requêtes (qui prend beaucoup de temps par rapport à l'exécution normale de notre application), `HttpClient` effectue ses requêtes de façon asynchrone. Cela signifie que les résultats des requêtes ne sont pas retournables directement dans notre service puisqu'ils seront générés bien après la fin des dites fonctions. Pour régler ce problème, `HttpClient` retourne une catégorie d'objet appelée **Observable**. Très utilisés au sein d'Angular, les Observables sont en mesure de **notifier** des **Observer** lorsqu'ils émettent des résultats. Transformons un peu nos deux méthodes `getHeroes` et `getHeroById` afin d'utiliser `HttpClient` et retourner nos fameux Observables.

```
getHeroes(): Observable<Hero[]> {
    return this.http.get<Hero[]>(this.heroesUrl);
}

getHeroById(id: number): Observable<Hero> {
    return this.http.get<Hero>(this.heroesUrl + '/' + id);
}
```

Petite nouveauté ici, si vous n'êtes pas ultra familier avec le Typescript. Nous voyons que la méthode `get` ainsi que le type de retour de notre fonction sont affublés d'un couple de chevrons `<>` encastrant un type (`Hero[]` dans le premier cas et `Hero` tout court dans l'autre). Cette notation correspond à un aspect avancé de l'orientation objet appelé la **généricité**. Dans notre cas, `Observable<Hero[]>` signifie simplement que notre Observable est un Observable qui, une fois la requête terminée, retournera un tableau de héros. De la même manière, `Observable<Hero>` retournera simplement un héros. Préciser ces attributs génériques aidera à déboguer nos Observer.

c. Exploiter nos Observables dans nos composants

Désormais, notre service retourne des Observables et non des tableaux. Nos observables enverront des résultats de façon asynchrone et ne sont donc pas exploitables de la même façon que des tableaux. Pour les utiliser, mon composant va devoir se mettre dans la position d'**Observer**. Pour ce faire, on utilisera une méthode de nos Observables appelée `subscribe`. Cette méthode prendra un paramètre une fonction qui sera utilisée comme callback prenant en paramètre nos fameux résultats.

Dit comme ça, cela a l'air bien compliqué mais l'implémentation de tout cela en Typescript tient sur une seule ligne. Ainsi, dans la méthode `ngOnInit()` de notre composant `app-heroes-list`, nous aurons :

```
ngOnInit() {
  this.heroService.getHeroes().subscribe((heroes: Hero[]) => this.heroes = heroes);
}
```

Et dans ngOnInit de app-hero-edit :

```
ngOnInit() {
  const id = Number(this.route.snapshot.paramMap.get('id'));
  this.heroService.getHeroById(id).subscribe((hero: Hero) => this.hero = hero);
}
```

Nous obtenons le même résultat que précédemment mais cette fois, nous passons par une api.

d. Sauvegarder, insérer et détruire des données

Pour ce faire, créons une méthode *updateHero* dans notre service pour demander à l'api de mettre à jour notre héros. Nous utiliserons la méthode http PUT et nous n'aurons même pas besoin de préciser l'id de notre héros, l'API se débrouillera pour aller le chercher dans le héros lui-même :

```
updateHero(hero: Hero): Observable<any> {
  return this.http.put<any>(this.heroesUrl, hero);
}
```

C'est prometteur mais pas encore suffisant. L'API demande au client de passer un certain header http dans les requêtes PUT et POST pour comprendre le format sous lequel lui est envoyé le héros (en l'occurrence, du JSON). Pour éviter de se prendre la tête, on stockera ce paramètre supplémentaire directement dans notre service de héros.

```
// ...
import { Observable } from 'rxjs';

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

@Injectable({
  providedIn: 'root'
})
export class HeroService {
  //...
```

Notre constante sera passée en troisième paramètre de notre méthode put() :

```
updateHero(hero: Hero): Observable<any> {
  return this.http.put<any>(this.heroesUrl, hero, httpOptions);
}
```

De la même manière, nous pouvons ajouter des méthodes pour ajouter et supprimer des héros.

```
insertHero(hero: Hero): Observable<Hero> {
  return this.http.post<Hero>(this.heroesUrl, hero, httpOptions);
}

deleteHero(hero: Hero): Observable<any> {
  return this.http.delete<any>(this.heroesUrl + '/' + hero.id);
}
```

Ces méthodes seront utilisables dans nos composants pour compléter notre application !

À suivre !

