Angular : composants

Corrigé

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

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

  public width = 100;
  public source = 'https://interactive-examples.mdn.mozilla.net/
  public links: {title: string, target: string}[];

  constructor() { }

  ngOnInit() {
  }

  public generateLinks() {
    this.links = [
      { title: 'Google', target: 'http://www.google.fr'},
      { title: 'Facebook', target: 'http://www.facebook.fr'},
      { title: 'Twitter', target: 'http://www.twitter.fr'},
    ];
  }

}
```

```html
<img [width]="width" [src]="source" (click)="generateLinks()" />
<ul *ngIf="links">
    <li *ngFor="let link of links"><a [href]="link.target">{{link.title}}</a></li>
</ul>
```

Angular : composants

@Input()

- Le composant qui appelle `CustomButton`
  peut lui passer des paramètres grâce au décorateur **`@Input`**
- Ajoutez à la classe `CustomButtonComponent`
  un attribut public **`imgTitle`** de type chaîne de caractères
- Préfixez-le du décorateur **`@Input()`**
- Dans le template, utilisez **`imgTitle`** comme valeur de l'attribut **`title`** de l'image
- Modifiez `app.component.html` pour donner une valeur à **`imgTitle`**
- Testez !

Angular : composants

@Input()

custom-button.component.ts

```
@Input() public imgTitle: string;
```

custom-button.component.html

```
<img [width]="width" [src]="source" [title]="imgTitle" (click)="generateLinks()" />
```

app.component.html

```
<app-custom-button imgTitle="Titre de l'image au survol"></app-custom-button>
```

Angular : composants

Bonus : un peu de style

- Sur une image, on peut utiliser l'attribut `width`…
  mais on souhaite modifier tout le CSS, sur n'importe quoi
- Méthode naïve :

```
<div [style]="'width: ' + (2 * nbColumns) + 'px'"></div>
```

- Méthode Angular :

```
<div [style.width.px]="2 * nbColumns"></div>
```

- Cela génère un attribut `style='width: 240px'`
  On peut en mettre autant qu'on veut,
  et avec toutes les propriétés CSS qu'on veut

Angular : composants

Binding (2/2) : la banane dans la boîte

- Reprenons cette ligne :

```
<img [width]="width" [src]="source" (click)="generateLinks()" />
```

- On voit des [crochets] qui vont chercher une information dans la partie TypeScript du composant
- On voit des (parenthèses) qui envoient de l'information vers la partie TypeScript du composant
- Ce n'est pas plus compliqué que ça :
  - Une donnée entrant dans le HTML est entre [crochets]
  - Une donnée sortante est entre (parenthèses)
- Vous l'avez compris : tous les événements (`click`, `keyup`, `mousedown`, etc.) seront entre (parenthèses)

Angular : composants

Binding (2/2) : la banane dans la boîte

- Parfois, on veut que la donnée circule librement entre le HTML et le TypeScript
- Par exemple :
  - On souhaite binder les champs d'un formulaire de façon à ce qu'une saisie change la donnée dans le TS, mais qu'une opération dans le TS change le contenu du champ
- On utilise alors `[(ngModel)]='myAttribute'`
  - C'est-à-dire qu'on a à la fois les crochets et les parenthèses

```
<input type="text" name="lastname" [(ngModel)]="lastname" />
```

- Moyen mnémotechnique : la banane dans la boîte !

Angular : composants

ViewChild

- Une autre méthode permet d'impacter le HTML avec le TS

```
<img #myImage src="http://site/ab.jpg" />
```

```
@ViewChild('myImage', {static: true}) image: ElementRef;
```

```
@ViewChild(CustomButtonComponent, {static: true}) myButton: CustomButtonComponent;
```

- Mais cette méthode n'est que rarement utilisée, car elle bypass la notion de binding (VanillaJS)

Angular : composants

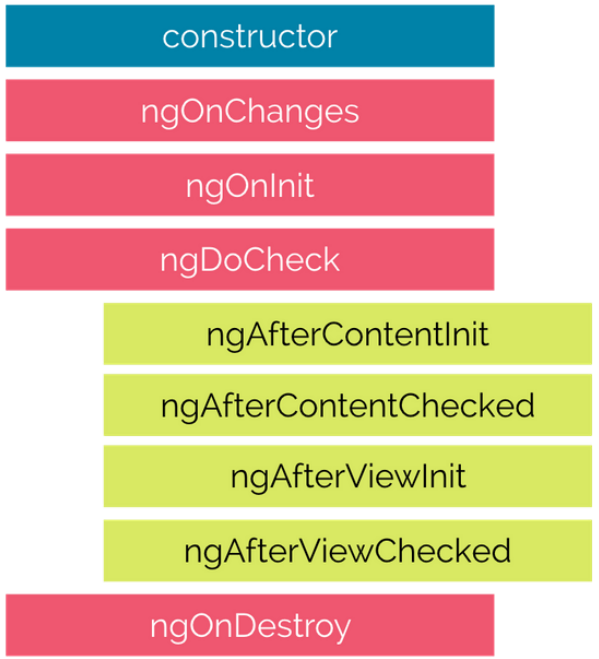
Lifecycle hooks

- Tous les composants (et les autres modules) ont un cycle de vie

Lorsqu'on appelle un composant, son constructeur est appelé

S'en suit un tas d'étapes, qui peuvent se répéter en cas de changement

Lorsqu'on change de page, que le composant n'est plus utilisé, il est détruit



Source : https://codecraft.tv/courses/angular/components/lifecycle-hooks/

- Les méthodes ici présentées sont appelées « *lifecycle hooks* » car elles sont appelées aux différentes étapes de la vie du composant

Angular : composants

Lifecycle hooks

- Si on reprend notre composant généré à l'aide d'angular-cli

Le corps de cette fonction sera exécuté à l'« initialisation » du composant (voir schéma)

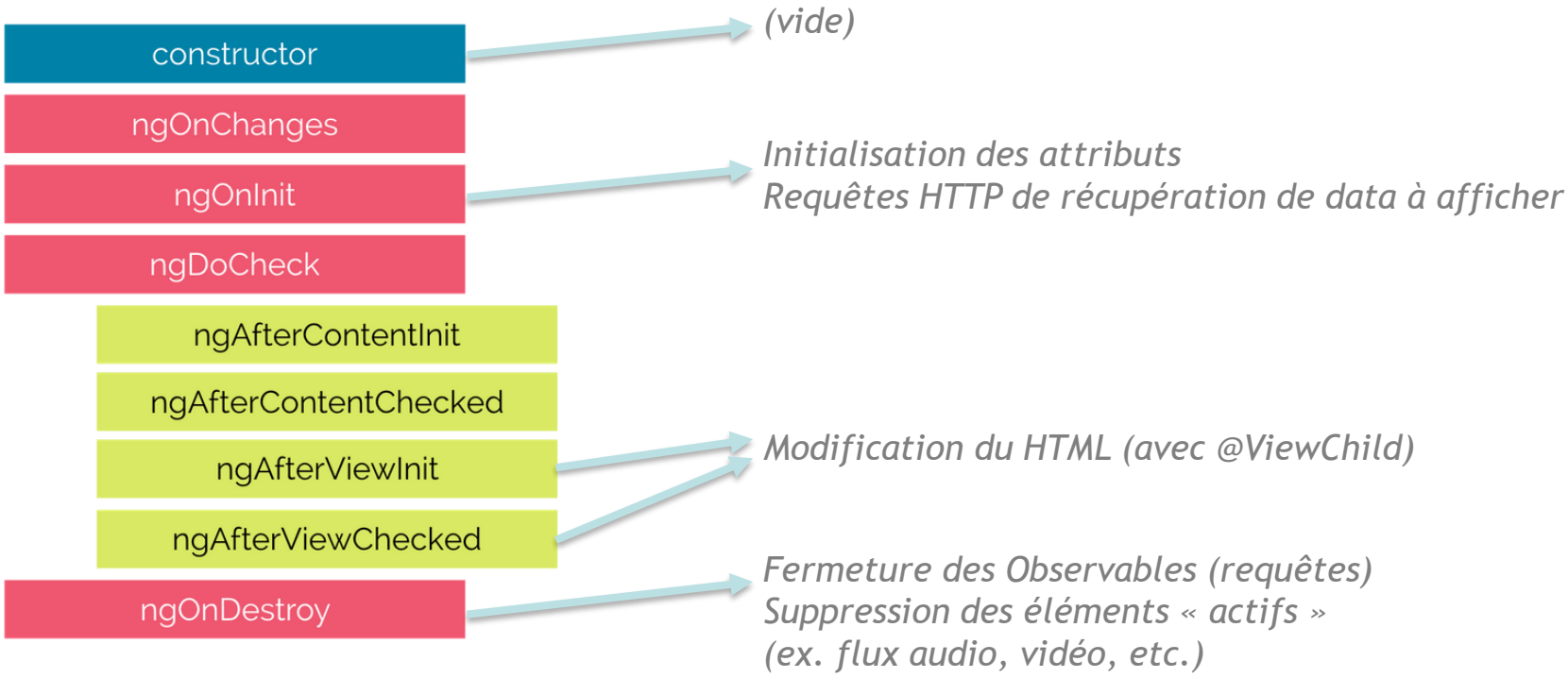
Pour que ngOnInit soit appelée, on doit implémenter l'interface OnInit

De manière générale,
pour utiliser un hook,
il faut implémenter l'interface associée…
et on peut le faire avec autant de hook que souhaité !

Angular : composants

Lifecycle hooks

- Exemple d'utilisation :



Angular : composants

Conclusion

- Les composants sont des bouts de HTML autonomes
- `AppModule` est le « hub » qui déclare tout
- Les composants sont constitués d'1 HTML, 1 CSS, 1 TS
- `{{myAttr}}` pour aller chercher un attribut
- `[src]='myUrl'` pour binder l'attribut `myUrl` *(one-way)*
- `(click)='action()'` pour binder la méthode sur l'événement
- `[(ngModel)]='myAttr'` pour faire du binding *two-way*
- Les lifecycle hooks permettent de déclencher des actions à des moments de la vie d'un composant

ANGULAR : ROUTING

Angular : routing

router-outlet

- Ouvrir `app.component.html`

```
<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://angular.io/cli">CLI Documentation</a></h2>
  </li>
  <li>
    <h2><a target="_blank" rel="noopener" href="https://blog.angular.io/">Angular blog</a></h2>
  </li>
</ul>
<router-outlet></router-outlet>
```

A la navigation, le composant cible vient s'insérer à l'endroit du router-outlet

Angular : routing

Définition des routes

- Commençons par supprimer la balise `app-custom-button` du composant `App`
- Nous allons l'intégrer à l'aide de la navigation
- Ouvrir `app-routing.module.ts`

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


const routes: Routes = [];

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

Angular : routing

Définition des routes

- Pour une fois, Angular ne nous fournit pas d'exemple… voici donc comment ça fonctionne :

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

import { DashboardComponent }   from './dashboard/dashboard.component';
import { HeroesComponent }      from './heroes/heroes.component';
import { HeroDetailComponent }  from './hero-detail/hero-detail.component';

const routes: Routes = [
  { path: '', redirectTo: '/dashboard', pathMatch: 'full' },
  { path: 'dashboard', component: DashboardComponent },
  { path: 'detail/:id', component: HeroDetailComponent },
  { path: 'heroes', component: HeroesComponent }
];

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

path: le chemin dans l'URL

component: le composant à charger

Ainsi, si je vais à /heroes, c'est le composant HeroesComponent qui sera inséré dans le router-outlet

Angular : routing

Définition des routes

- Adaptons donc `app-routing.module.ts` pour que l'adresse http://localhost:4200/button conduise au composant `CustomButton`

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { CustomButtonComponent } from './custom-button/custom-button.component';


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

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

Angular : routing

Définition des routes

- On teste en allant sur http://localhost:4200/button

- En général, on a au moins un `path: ‘’`
  qui conduit au composant d’accueil

- `path: ‘**’` permet de spécifier un comportement
  par défaut si aucune route ne matche
  avec la demande de l’utilisateur
  - → *Page 404*

Angular : routing

Go routing

- On souhaite un lien qui nous permettrait d'accéder à cette page sans saisir l'URL (c'est mieux !)

- Ouvrir `app.component.html`

- Ajouter à la liste des liens un lien vers notre « page »
  - Pour cibler une « page » de l'application, on n'utilise pas `href`, mais `routerLink`

- On voit alors que la page ne s'actualise pas : le principe des *Single Page Application* !

Angular : routing

Go routing avec des paramètres

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

import { DashboardComponent }   from './dashboard/dashboard.component';
import { HeroesComponent }      from './heroes/heroes.component';
import { HeroDetailComponent }  from './hero-detail/hero-detail.component';

const routes: Routes = [
  { path: '', redirectTo: '/dashboard', pathMatch: 'full' },
  { path: 'dashboard', component: DashboardComponent },
  { path: 'detail/:id', component: HeroDetailComponent },
  { path: 'heroes', component: HeroesComponent }
];

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

Sur cet exemple du tuto officiel, l'URL `detail` accepte un paramètre tel que `…/detail/452` pour afficher les infos du héros d'ID = 452

Pour passer un tel paramètre via le `routerLink`, on adapte l'URL

```
<a routerLink="/detail/{{hero.id}}">
```

```
<a [routerLink]="'/detail/' + hero.id"></a>
```

Angular : routing

Nested routes

- Il est possible de définir des « sous-routes »

```
const routes: Routes = [
  {path: '', redirectTo: 'home', pathMatch: 'full'},
  {path: 'find', redirectTo: 'search'},
  {path: 'home', component: HomeComponent},
  {path: 'search', component: SearchComponent},
  {
    path: 'artist/:artistId',
    component: ArtistComponent,
    children: [
      {path: '', redirectTo: 'tracks'}, (1)
      {path: 'tracks', component: ArtistTrackListComponent}, (2)
      {path: 'albums', component: ArtistAlbumListComponent}, (3)
    ]
  },
  {path: '**', component: HomeComponent}
];
```

Source : https://codecraft.tv/courses/angular/routing/nested-routes/

Angular : routing

Conclusion

- Le routing dans Angular est très simple,
  configuré dans un unique fichier (`app-routing.module.ts`),
  et facile à maintenir

- La navigation se fait à l'aide de `routerLink`

- Pour des questions d'accessibilité,
  on positionnera toujours les `routerLink` sur des liens `<a>`

- Vous manipulerez beaucoup
  ce genre de choses dans **Tour of Heroes**

TOUR OF HEROES (1/2)

Tour of Heroes

- **Tour of Heroes** est un tutoriel très complet sur les différentes fonctionnalités et bonnes pratiques d'Angular

- Il a été créé et est maintenu par Google
  - Donc a priori, ça tient la route… et c'est à jour

- C'est un passage obligé pour tous les développeurs Angular

Tour of Heroes

https://angular.io/tutorial

- Dans **Tour of Heroes**, vous allez créer une petite application de gestion de super-héros
- Vous allez utiliser les directives courantes, faire du binding, du routing, etc.
- Vous ferez également un formulaire simple
- Enfin, vous créerez des **services**, avant que je vous en parle en français
- Arrêtez-vous avant le « **6. HTTP** », nous reprendrons plus tard

OUTILS AVANCÉS

SERVICES, PIPES, DIRECTIVES, EVENTEMITTER

Outils avancés

- On a vu comment créer des composants
- Dans Tour of Heroes, vous avez également créé un **service** (`HeroService`), qui récupérait les données d'une fausse base
- Nous allons voir comment créer et à quoi servent :
  - Les services
  - Les pipes
  - Les directives

Outils avancés

- Les services contiennent :
  - Le code métier
  - Les appels externes (AJAX, etc.)
- De manière générale, on part du principe que :
  - Les **composants** sont la vue, ils gèrent l'affichage
  - Toute la mécanique autre doit être dans un **service**
- L'agencement des services (nombre, découpage, etc.) est libre et doit être rationalisé

Outils avancés

- Dans `HeroService`, il n'y a qu'une méthode :
  - `getHeroes(): Observable<Hero[]>`

- Cette méthode récupère « en base » les héros,
  et les retourne sous la forme d'un Observable (cf. plus tard)

- Le composant va appeler cette méthode pour pouvoir disposer des héros et en faire ce qu'il veut

- Vous l'avez vu : lorsqu'un module quelconque (composant, service, etc.) souhaite utiliser un service, il le déclare en **argument de son constructeur**

```
constructor(private heroService: HeroService) { }
```

- On peut en déclarer autant que nécessaire

```
constructor(
  private routeService: Route,
  private heroService: HeroService,
  private machinService: MachinService
) { }
```

Outils avancés

- Les pipes sont des éléments Angular
  qui permettent de raffiner l'affichage d'une donnée
- Concrètement, c'est ce qui va nous permettre,
  par exemple, de transformer :

```
2019-11-27T09:28:45
```

en :

```
Mercredi 27 septembre, 09h28
```

Outils avancés

Pipes

- Il existe plusieurs pipes natifs Angular, parmi lesquels :
  - `Date` : pour afficher une date selon un format paramètre
  - `Uppercase` : pour afficher la chaîne en capitales
  - `Percent` : pour afficher un pourcentage
  - `Currency` : pour afficher un montant
  - etc.

https://angular.io/api?type=pipe

- Un pipe s'utilise… grâce au symbole pipe (`|`)
- Ses paramètres éventuels sont ajoutés à la suite précédés de « : »

Outils avancés

Pipes

```
{{ dateObj | date }}               // output is 'Jun 15, 2015'
{{ dateObj | date:'medium' }}      // output is 'Jun 15, 2015, 9:43:11 PM'
{{ dateObj | date:'shortTime' }}   // output is '9:43 PM'
{{ dateObj | date:'mm:ss' }}       // output is '43:11'
```

```
<!--output '26%'-->
<p>A: {{a | percent}}</p>

<!--output '0,134.950%'-->
<p>B: {{b | percent:'4.3-5'}}</p>

<!--output '0 134,950 %'-->
<p>B: {{b | percent:'4.3-5':'fr'}}</p>
```

```
<!--output '$0.26'-->
<p>A: {{a | currency}}</p>

<!--output 'CA$0.26'-->
<p>A: {{a | currency:'CAD'}}</p>

<!--output 'CAD0.26'-->
<p>A: {{a | currency:'CAD':'code'}}</p>

<!--output 'CA$0,001.35'-->
<p>B: {{b | currency:'CAD':'symbol':'4.2-2'}}</p>
```

Outils avancés

Pipes

- On peut bien sûr créer des pipes personnalisés

```
import { Pipe, PipeTransform } from '@angular/core';
/*
 * Raise the value exponentially
 * Takes an exponent argument that defaults to 1.
 * Usage:
 *   value | exponentialStrength:exponent
 * Example:
 *   {{ 2 | exponentialStrength:10 }}
 *   formats to: 1024
*/
@Pipe({name: 'exponentialStrength'})
export class ExponentialStrengthPipe implements PipeTransform {
  transform(value: number, exponent?: number): number {
    return Math.pow(value, isNaN(exponent) ? 1 : exponent);
  }
}
```

On met le décorateur @Pipe

On implémente PipeTransform

On écrit la méthode `transform`
qui prend en entrée :
- La valeur « pipée »
- Les éventuels paramètres

et qui retourne la **valeur transformée**

Le type de retour est **personnalisable**

Outils avancés

Pipes

- On peut, dès lors (et une fois déclaré dans l'`AppModule`), l'utiliser dans nos templates :

```
<h2>Power Booster</h2>
<p>Super power boost: {{2 | exponentialStrength: 10}}</p>
```

Outils avancés

- Les **directives** sont des mots-clés que l'on ajoute aux balises HTML pour spécifier des comportements du DOM
- `*ngIf`, `*ngFor`, `*ngSwitch` sont des **directives structurelles** (« *structural directives* »)
- On peut créer des **directives attributs** (« *attribute directives* »)

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

@Directive({
  selector: '[appHighlight]'
})
export class HighlightDirective {
  constructor() { }
}
```

On utilise le décorateur `@Directive`

Le code exécuté sera cette fois situé dans le constructeur

Outils avancés

highlight.directive.ts

```
import { Directive, ElementRef } from '@angular/core';

@Directive({
  selector: '[appHighlight]'
})
export class HighlightDirective {
    constructor(el: ElementRef) {
       el.nativeElement.style.backgroundColor = 'yellow';
    }
}
```

my-comp.component.html

```
<p appHighlight>Highlight me!</p>
```

L'élément HTML sur lequel la directive est placée

Outils avancés

Directives

```
import { Directive, ElementRef, HostListener, Input } from '@angular/core';

@Directive({
  selector: '[appHighlight]'
})
export class HighlightDirective {

  constructor(private el: ElementRef) { }

  @Input('appHighlight') highlightColor: string;

  @HostListener('mouseenter') onMouseEnter() {
    this.highlight(this.highlightColor || 'red');
  }

  @HostListener('mouseleave') onMouseLeave() {
    this.highlight(null);
  }

  private highlight(color: string) {
    this.el.nativeElement.style.backgroundColor = color;
  }
}
```

```
<h1>My First Attribute Directive</h1>

<h4>Pick a highlight color</h4>
<div>
  <input type="radio" name="colors" (click)="color='lightgreen'">Green
  <input type="radio" name="colors" (click)="color='yellow'">Yellow
  <input type="radio" name="colors" (click)="color='cyan'">Cyan
</div>
<p [appHighlight]="color">Highlight me!</p>
```

Outils avancés

Récapitulons

SERVEUR WEB

Calculs locaux

SERVICES

COMPOSANTS

DIRECTIVES

PIPES

Outils avancés

EventEmitter

- Les composants ont parfois besoin d'interagir entre eux sans forcément passer par un service
- On utilise alors les `EventEmitter`, pour « émettre » une valeur que les autres composants peuvent saisir
- Rappel :

  - Une donnée entrant dans le HTML est entre [crochets]
  - Une donnée sortante est entre (parenthèses)

- L'objectif est donc :

```
<zippy (open)="onOpen($event)" (close)="onClose($event)"></zippy>
```

Outils avancés

EventEmitter

```
<zippy (open)="onOpen($event)" (close)="onClose($event)"></zippy>
```

- On utilise le décorateur `@Output` dans le composant qui a un message à émettre

```
export class Zippy {
  visible: boolean = true;
  @Output() open: EventEmitter<any> = new EventEmitter();
  @Output() close: EventEmitter<any> = new EventEmitter();

  toggle() {
    this.visible = !this.visible;
    if (this.visible) {
      this.open.emit(null);
    } else {
      this.close.emit(null);
    }
  }
}
```

Dans le composant parent (qui utilise `zippy`), on définit `onOpen` et `onClose`, qui doivent être déclenchés quand `zippy` émet un message (par `.emit(…)`)

`$event` contient la valeur émise

C'est le même principe que `(click)='…'`

HTTP & RXJS

HTTP & RxJS

- Angular facilite de façon incroyable les requêtes serveurs
- Il met à disposition un service `HttpClient` (anciennement `Http`) qui dispose des méthodes HTTP usuelles :
  - `get()`
  - `post()`
  - `put()`
  - `etc.`
- En paramètre de la fonction utilisée, on précise ce que l'on veut récupérer :

```
this.http.get<number>('http://site/data/count');
```

```
this.http.get<Person>('http://site/people/265');
```

Angular parse tout seul les données récupérées !

HTTP & RxJS

- `HttpClient` est un service, qu'on récupère donc dans le constructeur du service qui souhaite s'en servir

```
import { HttpClient } from '@angular/common/http';
```

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

HTTP & RxJS

- Les appels à `get`, `post`, etc. retournent des `Observable`, qui sont une part importante de la bibliothèque **RxJS**
- Tant qu'on n'appelle pas la méthode `subscribe()` d'un `Observable`, l'appel n'est pas effectué
- Exemple :

```
const person = this.http.get<Person>('http://site/people/265');
// person contient un Observable
// Cette initialisation ne fait aucun appel au serveur

person.subscribe();
// Cette fois, l'appel est effectué
```

HTTP & RxJS

Observables

```
const person = this.http.get<Person>('http://site/people/265');
// person contient un Observable
// Cette initialisation ne fait aucun appel au serveur

person.subscribe();
// Cette fois, l'appel est effectué
```

- On peut (et on souhaite, a priori…) manipuler la donnée récupérée dans le `subscribe`

```
const personObservable = this.http.get<Person>('http://site/people/265');
personObservable.subscribe(personFromServer => {
    this.person = personFromServer;
});
```

- `personFromServer` est directement de type **`Person`**, car `personObservable` est un **`Observable<Person>`**

HTTP & RxJS

Observables

```
const personObservable = this.http.get<Person>('http://site/people/265');
personObservable.subscribe(personFromServer => {
    this.person = personFromServer;
});
```

- Ou plus simplement :

```
this.http.get<Person>('http://site/people/265')
    .subscribe(person => {
        this.person = person;
    });
```

HTTP & RxJS

- Une fois qu'un `Observable` a été `subscribe`, il n'est plus utilisable
  - On ne peut pas `subscribe` deux fois un même `Observable`
- Il est possible d'appliquer des pipes aux Observables de la même façon que les pipes du template :

```
this.http.get<Person>('http://site/people/265')
    .pipe(
        map(person => {
            // Do something with person
        })
    )
    .subscribe();
```

- Ainsi, on peut spécifier le comportement de retour avant même d'exécuter la commande (pour, par exemple, passer l'Observable prêt-à-exécuter à un service/composant)

HTTP & RxJS

- Pour bien comprendre, imaginons que vous allez faire les courses, et qu'on vous explique ce qu'il faudra faire en revenant

```
this.http.get<Person>('http://site/people/265')
    .subscribe(person => {
        this.person = person;
    });
```

Traitement dans le `subscribe`

```
this.http.get<Person>('http://site/people/265')
    .pipe(
        map(person => {
            // Do something with person
        })
    )
    .subscribe();
```

Traitement dans un `pipe`

Après votre retour du supermarché, on vous explique que tel commission va dans le placard, telle commission va au frigo, et une fois fini, il faut nourrir le chat

Quand vous reviendrez, quelle que soit l'heure de votre départ, il faudra séparer ce qui va au frais et ce qui n'y va pas, et quand ce sera terminé, il faudra nourrir le chat

HTTP & RxJS

- Classiquement, on suit l'architecture suivante :

person.service.ts

```
function getPersonById(id: number): Observable<Person> {
    return this.http.get<Person>('http://site/people/265')
        .pipe(
            map(person => {
                // Do some stuff with the person
                return person;
            })
        );
}
```

my-custom-comp.component.ts

```
function myFunctionInView(idPerson: number) {
    this.personService.getPersonById(idPerson)
        .subscribe(person => this.viewedPerson = person);
}
```

- On prépare toute la mécanique dans le service (si nécessaire), et on appelle `subscribe` dans le composant
- Au moment du `subscribe`, le pipe est exécuté, et son résultat est inséré dans `this.viewedPerson`

HTTP & RxJS

Observables

- Il y a une autre façon d'appeler `subscribe` dans un composant : le pipe `Async`

person.service.ts

```
function getPersonById(id: number): Observable<Person> {
    this.http.get<Person>('http://site/people/265')
}
```

my-custom-comp.component.ts

```
public viewedPerson$: Observable<Person> = this.personService.getPersonById(  12  );
```

Le pipe `Async` appelle la méthode `subscribe` sur l'Observable

Ici, le `*ngIf` remplit `viewedPerson` (qui n'existe pas dans le composant) avec le retour de `viewedPerson$`

my-custom-comp.component.html

```
<div *ngIf="(viewedPerson$ | async) as viewedPerson">
    <span>{{ viewedPerson.name }}</span>
    <span>{{ viewedPerson.firstname }}</span>
</div>
```

TOUR OF HEROES (2/2)

Tour of Heroes

- Vous allez maintenant reprendre **Tour of Heroes**, pour faire la partie « 6. HTTP »
(et les précédentes si ce n'est pas fait)

- Dans cette partie, vous allez simuler un serveur avec une base de données de super-héros

- Vous allez requêter cette liste de héros, et implémenter les fonctions basiques de CRUD