



Présentation du Framework

PRÉSENTATION D'ANGULAR

Basic skills of Angular dev



HTML5 and CSS



JavaScript



Knowledge of Angular practices



Modules and components



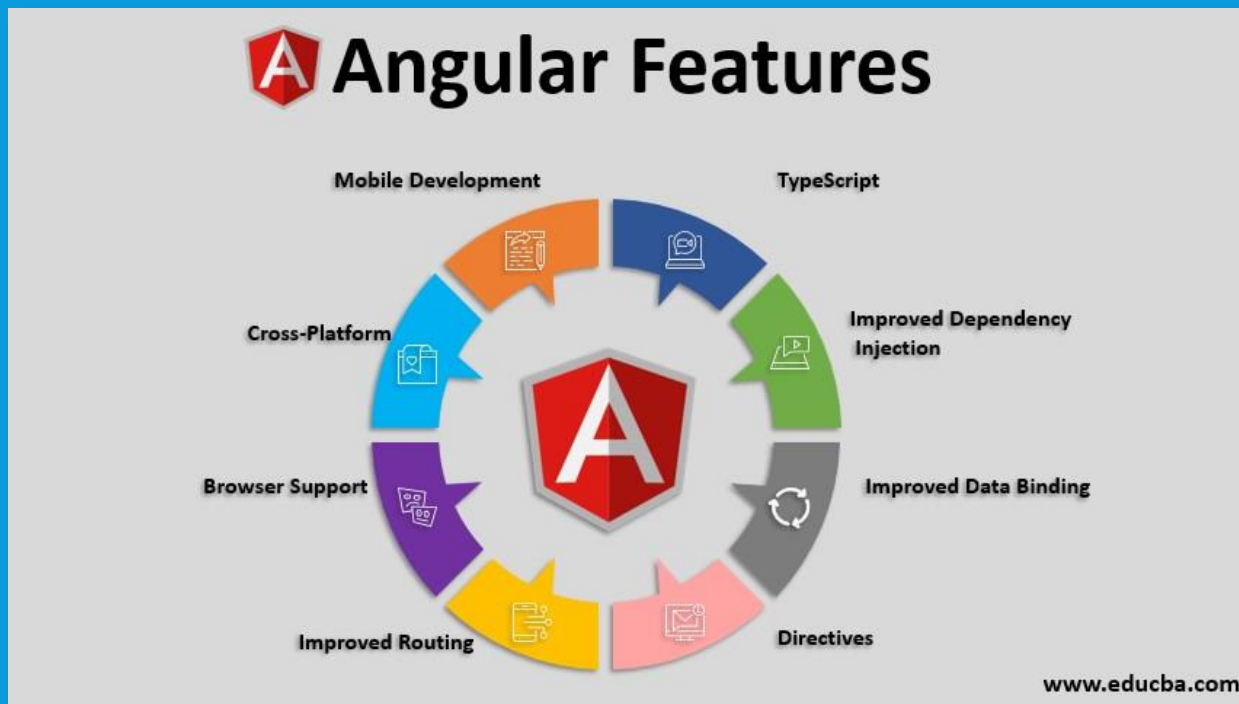
SPA building



Understanding of web services used

- Framework Javascript de Google
- Un composant : assemblage de trois fichiers :
 - HTML : Structurer l'application.
 - SCSS (surcouche à CSS) : Styliser l'application.
 - TYPESCRIPT (superset de JavaScript : Dynamiser l'application (routage, service...))
 - <https://www.typescriptlang.org/>

LES AVANTAGES D'ANGULAR



- Framework complet
 - L'ensemble des éléments de l'application est chargé (contenu, images, CSS et JavaScript) dans un unique fichier HTML
- Structures des applications précises suivant les "best practices".
 - modulaire et robuste du fait de Typescript avec les classes et interfaces
 - Lisible avec l'utilisation des fonctions dites lambda ou arrow
 - Typage strict

INSTALLATION D'ANGULAR

- Installer Node.js => <https://nodejs.org/fr/>
 - Permet d'avoir à disposition **npm**, outil de gestion des dépendances.
- Installer CLI d'Angular
 - **npm i -g @angular/cli**
 - *i pour install*
 - *-g pour installation de manière globale du package.*
 - **ng v** donne la version d'Angular CLI.
 - *Cela permet de s'assurer que l'installation est bonne.*

CRÉATION D'UNE APPLICATION ANGULAR

- **ng new name_project --style=scss --skip-tests=true**
 - *--style pour indiquer le type de style (scss, css, less par exemple).*
 - *--skip-tests=true pour ne pas mettre les fichiers de tests.*
- Ajout du bootstrap
 - Dans le dossier du projet : **npm install bootstrap@version --save**
 - *--save : ajout au package .json du projet*
 - Puis dans le fichier du projet **angular.json** :
 - Rubrique « **architect** » : ajout dans « styles » de *"./node_modules/bootstrap/dist/css/bootstrap.css"*,
- Lancement du server de développement : **ng serve**
 - <http://localhost:4200>

DÉPENDANCES

- Un projet peut avoir besoin d'une ou plusieurs dépendances :
 - Angular v12 : Le framework principal de l'application
 - Bootstrap 5 : Feuille de style CSS pour les formulaires, boutons, outils de navigation, ect...
 - ngx-bootstrap : Ensemble de composants utilisant Bootstrap directement intégré à l'univers Angular
 - ngx-toastr : Module qui permet de lancer des notifications à l'utilisateur sous forme de toasts
 - ngx-charts : Bibliothèque qui permet de générer des graphiques (barre, secteur, ect...)
 - Font Awesome : Bibliothèque d'icônes vectorielles, libre d'utilisation
 - Jest : Framework de test Javascript
 - ANGULAR RESSOURCES

STRUCTURE D'UN PROJET ANGULAR

- Un projet Angular est une arborescence de composants avec AppComponent comme composant racine



Sur l'exemple ci contre, on aurait alors :

- un composant pour le menu : HeaderComponent
- un composant pour le contenu : MainContentComponent
- un composant pour le Menu sur le côté : SideMenuComponent

COMPOSITION DU PROJET

- Index.html : fichier principal du projet
 - **<app-root>** est la racine de l'application
 - Plus, précisément l'**AppComponent** (le composant racine). **Unique** et appelé au démarrage de l'application donc obligatoire.
 - Dans le dossier **app** que se trouve le composant (**component**) décomposé en :
 - **AppComponent.html** : le contenu du component au format html.
 - **AppComponent.scss** : le style qui sera appliqué à ce component.
 - **AppComponent.ts** (fichier Typescript) : la logique du component.
 - Dans chaque component, on y trouve la déclaration de sa composition :

```
@Component({  
  selector: 'app-root',  
  templateUrl: './app.component.html',  
  styleUrls: ['./app.component.scss']  
})
```


CRÉATION D'UN COMPONENT

- Ng generate component name
 - Création des 3 fichiers propres à mon « component » dans un sous-dossier "name" à la racine de "app"
 - name.component.scss
 - name.component.html contenant la déclaration du template de notre composant.
 - Ici est défini le sélecteur (préfixé par app- par défaut) qui nous servira à l'insérer dans notre composant principal
 - name.component.ts
 - Contient le constructeur du composant.
 - Contient la méthode d'initialisation du composant.
 - @Component est un décorateur qui détermine les comportements nécessaires à l'utilisation du composant. (importé depuis le package @angular/core).
 - Plus la mise à jour de app.module.ts
 - Déclaration du module correspondant à notre composant dans le module principal.

COMPONENT : PARAMÈTRES D'ENTRÉE

- Un component Angular peut prendre de 0 à plusieurs paramètres d'entrée. Si un paramètre d'entrée est défini, il faudra spécifier une valeur lors de l'instanciation. Pour définir un paramètre d'entrée il faut utiliser l'annotation `@Input`, comme ceci : `@Input() myParam: string;`

```
// my-component.component.ts
import { Component, Input } from '@angular/core';

@Component({
  selector: 'app-my-component',
  templateUrl: './my-component.component.html',
  styleUrls: ['./my-component.component.scss']
})
export class MyComponent {
  @Input() myParam: string = "";
}
```



Vous pouvez donc instancier le component MyComponent dans votre vue html avec le paramètre d'entrée, comme ceci :

```
<my-component [myParam]="value"></my-component>
```

Note : La propriété value doit être présent dans le component AppComponent pour qu'elle soit liée au component instancié.

COMPONENT : PARAMÈTRES DE SORTIE

- Un component Angular peut prendre de 0 à plusieurs paramètres de sortie. Un paramètre de sortie renvoie toujours un objet de type EventEmitter. Le développeur doit également définir le type de valeur qui sera renvoyé dans l'évènement.
- Pour définir un paramètre de sortie il faut utiliser l'annotation @Output et le type EventEmitter, comme ceci : @Output() myAction: EventEmitter<string> = new EventEmitter();

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

@Component({
  selector: 'app-my-component',
  templateUrl: './my-component.component.html',
  styleUrls: ['./my-component.component.scss']
})
export class MyComponent {
  @Input() myParam: string = "";
  @Output() myAction: EventEmitter<string> = new EventEmitter();
}
```



Nous pouvons donc définir dans notre component à quel endroit l'évènement de sortie sera déclenché. Par exemple sur le clic d'un bouton. Il faudra alors appeler la méthode emit de l'objet EventEmitter pour déclencher l'évènement de sortie, par exemple comme ceci :

```
<button type="button" (click)="myAction.emit('ParamSortie')">Action</button>
```

Nous pouvons ensuite lier le paramètre de sortie myAction sur une action lorsque le component est instancié, comme ceci :

```
<my-component (myAction)="action($event)"></my-component>
```

AFFICHER DES DONNÉES DYNAMIQUES

- Méthodes principales pour exploiter une donnée :
 - Liaison par données : String interpolation avec `{{ property }}`
 - L'interpolation permet de récupérer la valeur d'une propriété d'un composant pour l'injecter dans la vue HTML. De plus, Angular met à jour l'affichage dans le template si la propriété du composant est modifiée.

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

```
<p>Mis en ligne le {{ createdAt }}</p>
```

- Liaison par propriétés (ou liaison par attribut) : Property binding avec `[property] = "condition"`
 - Le property binding permet quant à lui de modifier la valeur d'une propriété d'un élément du DOM et ceci grâce au crochet. Il est possible également d'appliquer une classe CSS suivant une condition. Il faut utiliser le « property binding » avec comme nom « **class** » suivi du nom de la classe à appliquer. Pour appliquer la classe `.myClass` vous devrez donc faire `[class.myClass]`.

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

```
<p [class.isImportant]="hero === 'Thor' ">Bonjour Thor !<p>
```

AFFICHER DES DONNÉES DYNAMIQUES

- Méthodes principales pour exploiter une donnée :
 - Liaison par évènements : Event binding avec `(name_function)="onName()"`
 - déclencher des actions sur des évènements qui surviennent sur la page comme un clic sur un bouton. Les évènements peuvent être écoutés, en utilisant les parenthèses, et on peut appeler une fonction du component lors du déclenchement
 - `<button type="button" (click)="action()">Action</button>`
 - *La méthode `action()` est défini dans le `component.ts`*
 - Liaison bi directionnelle : Two way binding avec `[(ngModel)]="attribut"`
 - `ngModel` provient de `FormsModule` d'Angular
 - `<input type="text" class="form-control" [(ngModel)]="appareilName">`

CONDITIONNEZ L'AFFICHAGE DES ÉLÉMENTS HTML : *NGIF

- Mise en place de Directive (ajout d'un comportement à un élément)
 - Tester la valeur d'une variable grâce à la directive structurelle *ngIf. Vous devez l'appeler sur la balise qui devra s'afficher si la condition est remplie
 - *ngIf touche à la structure du document.

```
<p *ngIf="property === 'value'">Bonjour le monde !</p>  
<p *ngIf="faceSnap.location">Photo prise à {{ faceSnap.location }}</p>
```

*Ici *ngIf évalue l'existence de la propriété soit :*

- "truthy" si existante et différente de null, false ou en fonction du type
- "falsey" si indéfini

AFFICHEZ DES LISTES D'ÉLÉMENTS : *NGFOR

- Mise en place de Directive (ajout d'un comportement à un élément)
 - Utilisation du *ngFor : Pour boucler sur un tableau vous devez utiliser la directive structurelle *ngFor. Vous devez l'appeler sur la balise à répéter :

```
<ul>
```

```
<li *ngFor="let property of list; let i=index">{{ i }} + {{ property }}</li>
```

```
</ul><
```

- ✓ 0 + Hulk
- 1 + Spider-Man
- **2 + Thor**
- 3 + IronMan
- 4 + Black Widow

AJOUTEZ DU STYLE DYNAMIQUE : [NGSTYLE] DIRECTIVE PAR ATTRIBUT

- Les fichiers SCSS :
 - Le style s'applique au component seulement et pas au parent, ni aux enfants.
 - Pas d'héritage : encapsulation des styles.
 - Permet de styliser un component par rapport aux autres.
 - Le fichier style.scss permet de mettre du style global à l'application.
- Utilisation de [ngStyle] :
 - Prends comme argument les **clés** (les styles CSS à modifier) et les **valeurs** que doit prendre le style.
 - Exemple : ajout de couleur à un texte

```
<span [ngStyle]="{ color: 'rgb(o, ' + faceSnap.snaps + ',o)'}">nombre de snaps : {{ faceSnap.snaps }}</span>
```


METTEZ DE LA CLASSE [NGCLASS]

DIRECTIVE PAR ATTRIBUT

- Ajout et retrait d'une classe
 - [ngClass] prend en argument en **clés**, un nom de classe CSS à appliquer et en **valeur** la condition qui doivent être remplies pour que la classe s'applique ou pas.

```
<div class="faceCard" [ngClass]="{snapped: faceSnap.snapTitle === 'Oops un Snap!'}">
```

```
...
```

```
</div>
```

- *snapped représente une classe décrite dans le component.scss*
- *Ici si la condition est remplie soit égale à 'Oops un Snap' alors j'applique la classe snapped à la div*

CONCLUSION SUR LES DIRECTIVES

- les directives : des classes qui permettent d'ajouter des comportements à des éléments HTML, ou même à des composants.
- des directives structurelles pour modifier la structure du document :
 - *ngIf pour ajouter un élément ou non au DOM selon une condition donnée ;
 - *ngFor pour ajouter autant d'éléments au DOM qu'il y a d'éléments dans un tableau.
- des directives par attribut pour gérer dynamiquement les styles :
 - [ngStyle] pour paramétrer des styles selon des valeurs venant du TypeScript ;
 - [ngClass] pour ajouter et retirer des classes CSS selon une condition donnée.

LES PIPES : MODIFIER LA CASSE

- Il existe trois pipes fournis par Angular pour modifier la casse :
 - **LowerCasePipe** : on affiche le texte en **minuscules**
 - **UpperCasePipe** : on affiche le texte en **majuscules**
 - **TitleCasePipe** : on affiche le texte avec une majuscule au début de chaque mot, avec le reste du mot en minuscules
- `<h2>{{ faceSnap.title | uppercase }}</h2>`
- `<h2>{{ faceSnap.title | lowercase }}</h2>`
- `<h2>{{ faceSnap.title | titlecase }}</h2>`

LES PIPES : FORMATER LES DATES

- Angular offre des formatages de dates comme pour la casse
- Pour configurer un pipe, on ajoute ":" puis la configuration
 - Longdate : `<p>Mis en ligne le {{ faceSnap.createdDate | date: 'longDate' }}</p>`
 - `<p>Mis en ligne le {{ faceSnap.createdDate | date: 'dd/MM/yy, à HH:mm' }}</p>`
 - `<p>Mis en ligne {{ faceSnap.createdDate | date: 'à HH:mm, le d MMMM yyyy' }}</p>`
- Documentation :
 - <https://angular.io/api/common/DatePipe>

LES PIPES : FORMATAGE DES NOMBRES

- Le formatage des **nombres** selon les règles de locale de l'application. Il y en a trois :
 - **DecimalPipe** – facilite l'affichage de nombres avec des **chiffres après la virgule** (qui met une virgule plutôt qu'un point, par exemple).
 - **PercentPipe** – formate les chiffres en **pourcentage**.
 - **CurrencyPipe** – permet d'afficher des nombres sous forme de **monnaie** très facilement.
 - `<p>PercentPipe ex 0.336 {{ 0.336 | percent }}</p>`
 - `<p>PercentPipe par défaut {{ 0.336 | percent: '1.0-1' }}</p>`
 - `<p>CurrencyPipe {{ 33 | currency }}</p>`
 - `<p>CurrencyPipe {{ 33 | currency: 'EUR' }}</p>`

LE PIPE ASYNC

- Le pipe `async` est un cas particulier mais extrêmement utile dans les applications Web, car il permet de gérer des données asynchrones, par exemple des données que l'application doit récupérer sur un serveur.
 - `<p>Mis à jour : {{ lastUpdate | async | date: 'yMMMMEEEEd' | uppercase }}</p>`
 - Dans l'exemple ci-dessus, le pipe permet d'attendre l'affichage de la donnée avant d'appliquer les autres pipes.
- Pour plus d'informations, vous pouvez vous référer à la [documentation Angular](#).

LES SERVICES

- Un service est un objet dont l'instanciation est gérée par Angular. Cet objet est unique dans toute l'application et il peut être injecté dans n'importe lequel de vos composants ou bien encore dans un autre service.
- Les services comme permettent donc :
 - facilite la maintenance, la lisibilité et la stabilité du code
 - Centralise les données, chaque partie de l'application aura accès aux mêmes informations, évitant beaucoup d'erreurs potentielles.
- Injection et instances :
 - Pour être utilisé dans l'application, un service doit être injecté, et le niveau choisi pour l'injection est très important. Il y a trois niveaux possibles pour cette injection :
 1. dans AppModule : ainsi, la même instance du service sera utilisée par tous les composants de l'application et par les autres services ;
 2. dans AppComponent : comme ci-dessus, tous les composants auront accès à la même instance du service mais non les autres services ;
 3. dans un autre component : le component lui-même et tous ses enfants (c'est-à-dire tous les composants qu'il englobe) auront accès à la même instance du service, mais le reste de l'application n'y aura pas accès.

LES SERVICES : CRÉATION ET INJECTION

- Pour créer un service il suffit de créer une classe précédée de l'annotation `@Injectable`. Puis il faut informer votre application qu'un nouveau service est disponible. Pour cela vous devez le déclarer dans le tableau `providers` du module `AppModule`

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

@Injectable()
export class AppService {
  myParams: string[] = [];

  myFunction(param: string): void {
    this.myParams.push(param);
  }
}
```

```
...

@NgModule({
  declarations: [AppComponent],
  imports: [
    BrowserModule,
    AppRoutingModule,
    BrowserAnimationsModule
  ],
  providers: [AppService], // Déclaration du nouveau service
  bootstrap: [AppComponent]
})
export class AppModule { }
```


LES SERVICES : CRÉATION ET INJECTION

- Vous pouvez maintenant l'injecter dans un component pour l'utiliser en ajoutant un paramètre au constructeur. Ce paramètre comprend un nom suivi du type du service à injecter, par exemple :

```
import { AppService } from './app.service';

@Component({
  selector: 'app-my-component',
  templateUrl: './my-component.component.html',
  styleUrls: ['./my-component.component.scss']
})
export class MyComponent {
  constructor(private appService: AppService) {} // Injection du service AppService
```

- Pour résumer, au démarrage de l'application Angular va instancier un nouvel objet de type AppService puis va l'injecter automatiquement dans les composants ou services qui en ont fait la demande via leurs constructeurs. Cet objet est unique et est partagé dans toute l'application.

LES SERVICES : HTTPCLIENT

- Le service HttpClient permet de lancer une requête HTTP. Pour utiliser ce service vous devez appeler la méthode qui correspond à la méthode HTTP utilisée. Attention : Il faut spécifier le type de valeur retourné par la requête en utilisant les chevrons après le nom de la méthode.

```
this.httpClient.get<string[]>('http://my-url.com').subscribe(data => {  
    console.log(data);  
});
```

Par exemple pour une requête get vous utiliserez la méthode get(). Cette méthode prend en paramètre l'URL appelée et renvoie un Observable. Nous pouvons souscrire à cet Observable pour exécuter une fonction de callback une fois que la requête sera résolue.

LES SPA (SINGLE PAGE APPLICATIONS)

- Les **Single Page Applications** (ou **SPA**) sont de plus en plus courantes dans le monde du développement web moderne.
- Une application SPA est chargée entièrement au démarrage et pour ce faire il faut utiliser le routeur.
 - Pour utiliser le routeur il faut d'abord créer un fichier de routing dans votre module puis déterminer des routes. Ce fichier de routing est déclaré dans le module principal AppComponent dans le tableau des imports @NgModule :

```
imports: [  
  BrowserModule,  
  AppRoutingModule  
],...
```

- génération du routing :
 - ng generate module app-routing
 - *Crée le fichier app-routing.module.ts*

LES SPA : DÉFINITION DES ROUTES

- Ensuite dans notre module de routing, on détermine une constante route qui contiendra alors nos différentes routes.

```
const routes: Routes = [  
  { path: '', redirectTo: 'accueil', pathMatch: 'full' },  
  { path: 'accueil', component: AccueilComponent },  
  { path: 'reservations', component: ReservationsComponent },  
],  
{ path: '**', component: NotFoundPageComponent }  
];
```

- *Explications :* - La première ligne sert à indiquer quel route prendre si aucune n'est précisée dans l'URL. Ici le visiteur sera redirigé sur la route accueil - La route accueil va afficher ce qui est prévu par le component `AccueilComponent` - La route `reservations` va afficher ce qui est prévu par le component `ReservationsComponent` - La dernière ligne précise que si la route demandée par l'utilisateur ne match avec aucune des précédentes, c'est alors le component `NotFoundPageComponent` qui va s'afficher.

LES SPA : UTILISATION DES ROUTES

- Le router outlet
 - Le component spécifié par la route sera chargé à la place du component `<router-outlet></router-outlet>`. Dans notre exemple ce component doit donc être déclaré dans le component principal AppComponent.
 - Exemple du fichier app.component.html :

```
<h1>{{ title }}</h1>
<router-outlet></router-outlet>
```
 - *Par exemple si le visiteur arrive sur l'URL `http://localhost:4200/accueil`, le component `AccueilComponent` sera affiché à la place de `<router-outlet></router-outlet>`.*

LES SPA : LE PASSAGE DE ROUTES

- 1^{er} Méthode : la directive routerLink

- `Continuer vers Snapface`

- Option 1 : ajout d'une classe css de comportement

- `Home`

- Ici une classe CSS qui surligne le lien sélectionné.
 - Attention : le parent du lien sélectionné est lui-même surligné

- Option 2 : supprimer le surlignement du parent

- `Home`

LES SPA : LE PASSAGE DE ROUTES

- 2^{ème} Méthode : Naviguez avec le routeur
 - Il existe une autre approche pour changer de route dans une application Angular. Vous pouvez injecter le **Router** dans vos composants et changer de route **dans le code Typescript**.

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

```
@Component({  
  selector: 'app-root',  
  templateUrl: './app.component.html',  
  styleUrls: ['./app.component.scss']  
})  
export class AppComponent {  
  constructor(private router: Router) {}  
  
  myFunction() {  
    this.router.navigate(['reservations']);  
  }  
}
```

Dans le fichier html du component :

```
<button (click)=" myFunction()"> 'reservations </button>
```

LES SPA : ACTIVATEDROUTE

- Il est très souvent utile dans une application dynamique d'avoir accès à la **route active**. Le component que vous allez associer à cette route pourra **recupérer** un id afin de demander à un service de lui envoyer l'objet correspondant.
 - 1 - créer la route dans notre constante ROUTES de app-routing.module.ts
 - ... { path: 'page/:id', component: SinglePageComponent }
 - *Le **:id** spécifie ce qui se trouvera après **page/** correspondant à un id*
 - 2 - récupérer le paramètre de route, on utilise ActivatedRoute dans le composant qui va afficher notre élément

```
constructor(private pageService: PageService, private route: ActivatedRoute) { }
```

Dans ngOnInit(), on récupère notre "id"

```
ngOnInit(): void {  
    // je recupere l'id puis fais appel à la méthode pour récupérer mon objet à afficher  
    const pagId = +this.route.snapshot.params['id'];  
    this.page = this.pageService.getPageById(pagId);  
}
```


LES SPA : ACTIVATEDROUTE

- le **snapshot** de la route (un **snapshot** est un **aperçu instantané** d'une valeur qui change au cours du temps).
- Les paramètres de la route sont en string. Pour récupérer un type number, on va faire du typecast en utilisant **+expression** ou **Number(...)**
- Enfin dans le component parent qui fait lance l'appel à mon élément à afficher, je crée une méthode qui va faire appel à mon élément désiré

```
onViewPage() {  
  this.router.navigateByUrl(`page/${this.page.id}`);  
}
```

GUARD

- Une guard est un service qu'Angular exécutera au moment où l'utilisateur essaye de naviguer vers la route sélectionnée. Ce service implémente l'interface `CanActivate`, et donc doit contenir une méthode du même nom qui prend les arguments `ActivatedRouteSnapshot` et `RouterStateSnapshot` (qui lui seront fournis par Angular au moment de l'exécution) et retourne une valeur booléenne, soit de manière synchrone (boolean), soit de manière asynchrone (sous forme de Promise ou d'Observable)

```
export class AuthGuardService implements CanActivate {  
  
  constructor(private authService: AuthService, private router: Router) {}  
  
  canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): Observable<boolean> | Promise<boolean> | boolean {  
    if (this.authService.isAuthenticated) { return true; } // l'utilisateur est authentifié  
    else {  
      this.router.navigate(['/auth']); // il ne l'est pas alors redirection vers la page auth  
    }  
    return false; // pour satisfaire tous les conditions de sortie de la méthode  
  }  
}
```

GUARD

- Ensuite au niveau de nos routes, on place notre canActivate sur celles qu'on le souhaite protéger :

```
const appRoutes: Routes = [  
  { path: 'appareils', canActivate: [AuthGuard], component: AppareilViewComponent },  
  { path: 'appareils/:id', canActivate: [AuthGuard], component: SingleAppareilComponent },  
  { path: 'auth', component: AuthComponent },  
  { path: '', component: AppareilViewComponent },  
  { path: 'not-found', component: FourOhFourComponent },  
  { path: '**', redirectTo: 'not-found' }  
];
```

LES MODULES

- Les modules permettent de regrouper un ensemble de composants, de services, de directives, etc... pour découper son application par fonctionnalités. Ce qui permettra une meilleure organisation du code.
- Pour un nouveau module, les métadonnées que vous devez spécifier sont :
 - **declarations** : Un tableau qui contient les composants du module
 - **imports** : Un tableau qui contient les modules ou composants externes à importer dans le nouveau module
 - **exports** : Un tableau qui contient les composants ou module qui peuvent être utilisés ailleurs dans l'application
 - **providers** : Les services définis dans ce nouveau module
- Attention : Le nouveau module doit importer CommonModule depuis @angular/common pour pouvoir utiliser les fonctionnalités de base telles que *ngIf ou *ngFor.
- Pour que le nouveau module soit utilisable, il doit être déclaré dans les imports du module principal de l'application : app.module.ts.

LES MODULES : ROUTING

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';

import { MyFeatureComponent } from './my-feature.component';
import { MyFeatureService } from './my-feature.service';

@NgModule({
  declarations: [
    MyFeatureComponent
  ],
  imports: [
    CommonModule, // Import obligatoire pour utiliser les fonctionnalités de base
    MyFeatureRoutingModule // Déclaration du nouveau fichier de routing
  ],
  providers: [
    MyFeatureService
  ]
})
export class MyFeatureModule {}
```

- Routing du module : Vous pouvez déclarer un nouveau fichier de routing pour votre nouveau module.

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

import { SuperHeroesComponent } from './super-heroes.component';

const routes: Routes = [
  { path: 'my-feature', component: MyFeatureComponent }
];

@NgModule({
  // Attention il faut appeler la méthode forChild pour déclarer un fichier
  // de routing supplémentaire
  imports: [RouterModule.forChild(routes)],
  exports: [RouterModule]
})
export class MyFeatureRoutingModule {}
```

AJOUTER DES PROPRIÉTÉS PERSONNALISÉES

- Création d'une classe objet
 - Création d'un sous-dossier `models` dans `app`
 - Création d'une classe `name.model.ts`
 - Déclaration de nos propriétés
 - Déclaration du constructeur à notre objet
 - Raccourcie dans la création offert par Typescript
 - `constructor(public id: number...)`
 - Création de l'objet par `new Name()`
 - Transmission du parent vers l'enfant
 - `<app-face-snap [faceSnap]="mySnap"></app-face-snap>`
 - utilisation depuis l'extérieur d'un component enfant en utilisant le décorateur `@Input()`
 - `@Input() faceSnap!: FaceSnap;`

LES SERVICES : CENTRALISEZ LA LOGIQUE

- L'objectif est de mettre en place un service dédié aux données
- Dans la classe service, on y implémente la logique d'accès aux données.
- Cette logique sera ensuite utilisé dans les composants en modifiant les constructeurs pour faire appel au service.

LES OBSERVABLES AVEC RXJS

- Objet qui émet des informations dans le temps.
 - Les Observables sont mis à disposition par RxJS, un package tiers qui est fourni avec Angular. Il s'agit d'un sujet très vaste => toute la documentation sur le site [ReactiveX](https://reactivex.io/).
 - Pour pouvoir avoir accès aux Observables, il faut importer le type Observable
 - `Import { Observable, of } from 'rxjs';`
 - Maintenant que vous avez un Observable, il faut l'observer ! Pour cela, vous utiliserez la fonction **subscribe()**, qui prendra comme arguments entre une et trois fonctions anonymes pour gérer les trois types d'informations que cet Observable peut envoyer :
 - des données,
 - une erreur,
 - un message complète.

LES OBSERVABLES AVEC RXJS

- EventEmitter
 - Module de @angular/core qui permet de partager des données entre les composants à l'aide des méthodes emit() et subscribe(). Il observe les changements et les valeurs et émet les données vers les composants abonnés à cette instance.

```
// La fonction get retourne un observable sur lequel on souscrit
// La fonction de callback sera exécutée quand le serveur aura renvoyé le résultat

this.httpClient.get<string>('mon_url').subscribe(value => {
  console.log(value);
});
```

L'OBSERVABLES SUBJECT ET OPÉRATEURS

- L'Observable permet non seulement de réagir à de nouvelles informations, mais également d'en émettre.
 - une variable dans un service, par exemple, qui peut être modifié depuis plusieurs composants et qui fera réagir tous les composants qui y sont liés en même temps.
 - Voici l'intérêt des Subjects dans les formulaires par exemple.
- Un opérateur se place entre l'Observable et l'Observer et peut filtrer et/ou modifier les données reçues avant même qu'elles n'arrivent à la Subscription.
 - Voici quelques exemples rapides:
 - `map()` : modifie les valeurs reçues — peut effectuer des calculs sur des chiffres, transformer du texte, créer des objets...
 - `filter()` : comme son nom l'indique, filtre les valeurs reçues selon la fonction qu'on lui passe en argument.
 - ... [Operators](#)

```
import { filter } from 'rxjs/operators';

// Récupère des articles depuis le serveur et filtre sur l'auteur
this.httpClient.get<Article[]>('mes_articles_url').pipe(
  filter(article => article.author === 'Jean Dupont')
).subscribe(value => {
  console.log(value);
});
```

LES FORMS – MÉTHODE TEMPLATE

- Les template driven permettent d'utiliser la directive ngModel directement dans la vue HTML. Elle permet de faire le lien entre le champ d'un formulaire et une propriété d'un component.
- Tout d'abord pour utiliser les template driven forms il faut importer dans un module le FormsModule depuis @angular/forms.
- Ensuite il suffit de créer un formulaire dans la vue d'un component en utilisant la directive ngModel qui est maintenant disponible.

LES FORMS – MÉTHODE TEMPLATE

```
<form (ngSubmit)="onSubmit(f)" #f="ngForm">
  <div class="form-group">
    <label for="name">
      Nom de l'appareil
    </label>
    <input type="text" id="name" class="form-control" name="name" ngModel>
  </div>
  <div class="form-group">
    <label for="status">
      État de l'appareil
    </label>
    <select id="status" class="form-control" name="status" ngModel>
      <option value="allumé">Allumé</option>
      <option value="éteint">Éteint</option>
    </select>
  </div>
  <button class="btn btn-primary" type="submit">Enregistrer</button>
</form>
```

L'attribut `#f` est ce qu'on appelle une référence locale. Vous donnez simplement un nom à l'objet sur lequel vous ajoutez cet attribut

L'objet `NgForm` (à importer depuis `@angular/forms`) comporte beaucoup de propriétés très intéressantes ; Vous pouvez en trouver la liste complète dans la documentation officielle.

LES FORMS – MÉTHODE TEMPLATE

- Validation des données

- `<input type="text" id="name" class="form-control" name="name" ngModel required>`
 - Ici, on ajoute une directive `required` devant `ngModel` rendant le champs obligatoire
- `<button class="btn btn-primary" type="submit" [disabled]="f.invalid">Enregistrer</button>`
 - Pour intégrer l'état de validation du formulaire, on lie la propriété `disabled` du bouton à la propriété `invalid` du formulaire.


- Exploiter des données

```
onSubmit(form: NgForm) {  
  const name = form.value['name'];  
  const status = form.value['status'];  
  this.appareilService.addAppareil(name, status);  
  this.router.navigate(['/appareils']);  
}
```

LES FORMS – MÉTHODE RÉACTIVE

- À la différence de la méthode template où Angular crée l'objet du formulaire, pour la méthode réactive, vous devez le créer vous-même et le relier à votre template.
 - Chaque élément de notre formulaire est un « *FormGroup* »
 - Chaque « *FormGroup* » peut contenir 1 ou plusieurs « *FormControl* »
 - Chaque « *FormControl* » contient une valeur par défaut et un system de validation.
- Pour créer un formulaire réactif, rien de plus facile. Vous utilisez le *formBuilder* d'Angular ou directement les *FormGroup* avec les *FormControls* :

```
export class NewUserComponent implements OnInit {  
  userForm: FormGroup;  
  constructor(private FormBuilder: FormBuilder) {}  
  ngOnInit() {  
    this.initForm();  
  }  
}
```



```
initForm() {  
  this.userForm = this.formBuilder.group({  
    firstName: ['', Validators.required],  
    lastName: ['', Validators.required],  
    email: ['', [Validators.required, Validators.email]],  
    drinkPreference: ['', Validators.required],  
    hobbies: this.formBuilder.array([])  
  });  
}
```

LES FORMS – MÉTHODE RÉACTIVE

CÔTÉ TEMPLATE HTML

```
<div class="col-sm-8 col-sm-offset-2">
  <form [formGroup]="userForm" (ngSubmit)="onSubmitForm()">
    <div class="form-group">
      <label for="firstName">Prénom</label>
      <input type="text" id="firstName" class="form-control" formControlName="firstName">
    </div>
    ...

    <!-- liaison de la div avec formArrayName -->
    <div formArrayName="hobbies">
      <h3>Vos hobbies</h3>
      <!-- repetition du formgroup pour chaque hobbies recuperé par getHobbies -->
      <div class="form-group" *ngFor="let hobbyControl of getHobbies().controls; let i = index">
        <!-- input s'affiche avec le name -->
        <input type="text" class="form-control" [formControlName]="i">
      </div>
      <button type="button" class="btn btn-success" (click)="onAddHobby()">Ajouter un hobby</button>
    </div>
    <button type="submit" class="btn btn-primary" [disabled]="userForm.invalid">Soumettre</button>
  </form>
</div>
```

INTERACTION AVEC UN SERVEUR EN HTTPCLIENT (ICI FIREBASE)

- Depuis le service
 - constructor(private httpClient: HttpClient) {}
 - Exemple de méthode d'inscription vers Firebase :

```
saveAppareilsToServer() {  
  this.httpClient  
    .post('https://httpclient-demo.firebaseio.com/appareils.json', this.appareils)  
    .subscribe(  
      () => {  
        console.log('Enregistrement terminé !');  
      },  
      (error) => {  
        console.log('Erreur ! : ' + error);  
      }  
    );  
}
```

- post() permet de lancer un appel POST, prend comme premier argument l'URL visée, et comme deuxième argument le corps de l'appel, c'est-à-dire ce qu'il faut envoyer à l'URL
- l'extension .json de l'URL est une spécificité Firebase, pour lui dire que vous lui envoyez des données au format JSON
- la méthode post() retourne un Observable. C'est en y souscrivant que l'appel est lancé
- dans la méthode subscribe(), vous prévoyez le cas où tout fonctionne et le cas où le serveur vous renverrait une erreur.

INTERACTION AVEC UN SERVEUR EN HTTPCLIENT (ICI FIREBASE)

- Depuis le service
 - Exemple de méthode de récupération depuis Firebase

```
getAppareilsFromServer() {  
  this.httpClient  
    .get<any[]>('https://httpclient-demo.firebaseio.com/appareils.json')  
    .subscribe(  
      (response) => {  
        this.appareils = response;  
        this.emitAppareilSubject();  
      },  
      (error) => {  
        console.log('Erreur ! : ' + error);  
      }  
    );  
}
```

La méthode `get()` retourne également un `Observable`, mais TypeScript a besoin de savoir de quel type elles seront (l'objet retourné est d'office considéré comme étant un `Object`).

Vous devez donc, dans ce cas précis, ajouter `<any[]>` pour dire que vous allez recevoir un array de type `any`, et que donc TypeScript peut traiter cet objet comme un array : si vous ne le faites pas, TypeScript vous dira qu'un array ne peut pas être redéfini comme `Object`.

MERCI POUR VOTRE ATTENTION

"Codez comme si le gars qui finit par maintenir votre code est un psychopathe violent qui sait où vous vivez." - Jeff Attwood

QUESTIONS ?