Angular

1. Introduction
2. Projet Angular
3. Binding
4. Pipes
5. Directives
6. @Input / @Output
7. Services et Injection
8. Formulaires
9. Routing
10. Storage
11. Observable
12. HttpClient
13. Publication

Table des matières

1. Introduction

Introduction

Table des matières

1.1 Angular ? C'est quoi ?

https://angular.io/

Angular est un framework de développement front-end mis en place par Google et basé sur Javascript (au travers de NodeJS et Typescript)

Il permet de mettre en place des SPA (Single Page Application). Autrement dit : la navigation entre les "pages" semble invisible pour l'utilisateur. Il a l'impression de rester constamment sur la même page puisqu'il n'y a pas de rechargement de l'intégralité du site d'une page à l'autre.

La première version du framework date de 2016 et se met à jour régulièrement pour répondre aux attentes du monde professionnel.

1.2 Avantages / Inconvénients

Pour

- Framework complet pour le front
- Injection de dépendances
- Compilation rapide
- Documentation complète et simple
- Evolue constamment
- Modèle proche du MVVM
- Avantage du Typescript

Contre

- Lourdeur de la syntaxe
- Pas d'accès direct à une DB
- Migration d'une version à l'autre problématique (les librairies mettent du temps à suivre)
- Courbe d'apprentissage énorme

1.3 NodeJS

https://nodejs.org/en/

En quelques mots : NodeJS est un environnement back-end développé en Javascript sur base de la machine virtuelle V8, moteur d'exécution JS (qui se cache dans Chrome).

Il permet de manière très légère, de mettre en place des APIs (grâce à Express) et permet l'exécution du Javascript côté serveur.

Il est associé à un gestionnaire de paquet - **NPM** - qui fonctionne via un terminal et permet d'ajouter des dépendances tel que Angular, à un projet.

1.4 Typescript

https://www.typescriptlang.org/

Il s'agit d'un langage développé par Microsoft pour faciliter et "sécuriser" la production de code Javascript.

Typescript est en réalité une surcouche du JS. Ce qui signifie que tout code JS est un code Typescript valide.

Il apporte les notions de programmation orienté objet et de typage statique des variables qui font cruellement défaut à Javascript pour en faire un langage beaucoup plus structuré et facile d'accès.

1.5 Installation d'Angular

1. Installer NodeJS (prendre la version LTS sur le site officiel)
2. Installer Angular en ligne de commande (WIN+R => CMD)

npm install -g @angular/cli

Le paramètre -g permet une installation globale au niveau du système et pas uniquement dans le dossier ou vous vous situez

3. Vérifier que l'installation est bien terminée en tapant

ng version

1.6 Outils

L'éditeur utilisé dans le cadre du cours sera Visual Studio Code

https://code.visualstudio.com/

Il permettra d'installer des plug-ins spécifiques à Angular :

- **Angular Schematics** => exécution de commande via un menu contextuelle
- **Angular Snippets**
- **Angular Language Service**

Les deux suivants amènent des aides au code comme l'auto complétion ou des raccourcis d'écriture.

2. Projet Angular

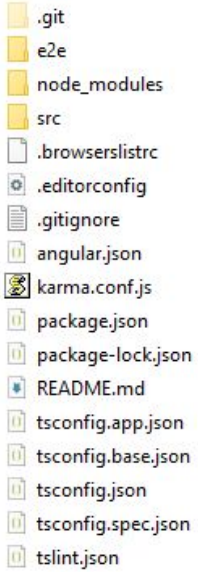
Mise en place

Projet Angular

Table des matières

2.1 Commandes et options

- **ng new “nom_du_projet”** => Crée un dossier “nom_du_projet” en y copiant tout ce qui est nécessaire au framework



Options

--skip-tests	Désactive la copie des fichiers de test unitaires
--routing	Ajoute le routage à l’application
--strict (true/false)	Utilisation du mode strict qui force l’initialisation des variables/propriété dans le constructeur
--skip-git	Désactive l’initialisation par défaut du dépôt Git

Liste des options : https://angular.io/cli/new

2.1b Installation du projet

Une fois la commande basique exécutée (ng new “projet”), le système vous posera 2 questions

1. Ajouter angular routing ? => répondre oui
2. Mode de template css ? => choisir SCSS (pour plus de compatibilité avec les différents framework/librairie graphiques)

Ces questions peuvent être évitées en ajoutant les options adéquates à la commande de base.

2.1c Démarrer le serveur -- NG SERVE

La commande - **ng serve** - compile et démarre l'application sur le port 4200 (par défaut).

Dans le cas où vous auriez plusieurs applications qui tournent en parallèle, il est possible de modifier le port de l'app

```
ng serve --port=42XX
```

L'option --open ouvre l'app dans le navigateur par défaut une fois celle ci compilée

Une fois compilée et démarrée, l'application pourra être modifiée sans avoir à relancer le serveur. Le processus "**Ahead-of-time**" se charge de recompilé en direct l'application à chaque sauvegarde de fichier.

2.2 Arborescence du projet

Les différents dossiers et fichiers importants

Fichier/dossier	Utilité
/e2e	Dossier réservé aux tests end user
/node_modules	Contient tout les modules nécessaire au fonctionnement du framework, ainsi que les module ajouté par la suite (=> ne pas toucher !!!)
/src	Dossier principal de développement. C'est là qu'on va travailler
angular.json	Configuration du workspace et du projet
package.json	Liste et version des packages utilisés dans le projet
tsconfig.json	Configuration de la compilation Typescript
tslint.json	Configuration permettant la vérification du code à chaud sans le compiler

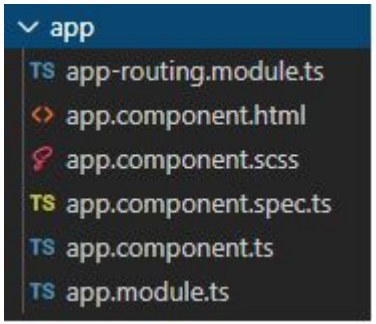
2.2 Arborescence du projet

Détails du dossier /src

Fichier/dossier	Utilité
/app	C'est la que l'application prend vie. On y intégrera nos pages ainsi que leur logique
/assets	Contiendra les éventuelles média supplémentaires (images, sons, vidéos) nécessaires à l'applications
/environments	Permettra de localiser les variables d'environnement tel que les adresses d' API
index.html	Point d'entrée de l'application. Contiendra les imports css/js éventuelles
styles.scss	Feuille de style par défaut de l'application
main.ts	Permet de définir le module de démarrage le l'app (laisser par défaut de préférence)

2.3 app module

La programmation Angular est dite “Modulaire” ce qui signifie que nous allons mettre en place plusieurs modules et les faires communiquer entre eux. Nous verrons cela plus tard. Attardons nous sur le module principale de notre application. Celui créé par défaut lors de la création d’un projet et qui sera le point d’entrée de notre app



Il s’agit du root module (module parent) qui englobe toute l’application.

Il propose son propre routage, il s’agit également du point d’entrée vers les différentes logiques, fonctionnelle et visuelle, de votre application

Pour des raisons pratiques, nous ferons l’impasse sur tous les fichiers suffixés .spec.ts . Ces fichiers étant réservés aux tests unitaires.

2.3 app.module.ts

Le fichier app.module.ts représente la configuration du module principal et propose différentes parties que nous allons détaillés ici

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

Le décorateur **@NgModule** signifie au système que le fichier est le point d'entrée du module et permet d'apporter une structure à celui ci

declarations : [] => Nous y déclarons le/les composants utilisé(s) par le module
imports : [] => Contient les différents modules importés et utilisés par celui en cours
exports : [] => Permet d'exporter tout ou partie des modules/composants utilisé par celui en cours
providers : [] => Contiendra des informations pour l'injection de dépendances
bootstrap : [] => Le composant d'amorçage du module (si composant il y a)

export class AppModule { } => signifie simplement que le module est exporté et donc importable ailleurs dans l'application

2.4 Module => création

Un module contiendra d'office un fichier "***nom.module.ts***". Qui, à l'instar du app.module.ts contiendra le décorateur **@ngModule.**

ng g m nom_du_module => crée un nouveau module

Aucune obligation n'existe quand à la présence d'un composant ou d'un routing spécifique dans un nouveau module (exception faite pour le AppModule)

Il est néanmoins possible de créer un composant à inclure directement dans un module.

ng g c nom_du_composant -m nom_du_module

Ajouter l'option **--routing** pour inclure "***nom-routing.module.ts***" au module dés sa création

2.5 Component

Notre module principale, en plus d'intégrer un routing (que nous détaillerons plus tard), et un fichier de configuration (app.module.ts), intègre un component composé de 3 fichiers

- app.component.html => Template visuel
- app.component.scss => Feuille de style
- app.component.ts => Code behind en typescript

Un component est l'unité de travail en Angular. Autrement dit, nos pages seront basées sur des components. Chacun étant composé de ces 3 fichiers (html/css/ts)

ng generate component nom_du_composant || ng g c nom_du_composant

==> Commande de création d'un nouveau composant

2.5 Component => TS

C'est ici que nous mettrons en place toute la logique des composants au travers du code Typescript.

Ce fichier est composé d'un décorateur **@Component** et d'une classe complète avec son constructeur et ses imports.

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

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

Les imports représentent tout modèles, modules ou services nécessaires au fonctionnement du composant.

C'est dans la classe que nous intégrerons la logique fonctionnelle (propriétés, méthodes, constructeur, ...)

2.5 Component => TS

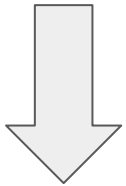
Le décorateur décrit 3 propriétés du composant

- **selector** => Permet d'intégrer ce composant dans n'importe quelle page html
- **templateUrl** => Url de la page html liée au composant
- **styleUrls** => Url de la feuille de style du composant

Exemple d'appel par sélecteur

Fichier TS

```
selector: 'app-root',
```



Fichier HTML

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

2.5 Component => HTML

Le fichier.html d'un component contiendra comme son extension l'indique, du code HTML principalement. Mais aussi divers choses qui nous viennent du framework Angular :

- Directives
- Bindings
- Pipes

Qui seront bien évidement vues en détails dans les chapitres dédiés

2.6 Intro aux cycles de vie des composants

Le cycles de vie d'un composant (**Hooks**) représente les différents instants de son existence. Chacun d'eux, demande que la classe du composant implémente l'interface nécessaire à son fonctionnement

OnInit => Exécute le contenu de la méthode *ngOnInit()* à l'initialisation du composant

Ne s'exécute qu'une seule fois au premier appel du composant

```
export class HomeComponent implements OnInit {
  ngOnInit(): void {
  }
```

2.6 Intro aux cycles de vie des composants (suite)

OnDestroy => Exécute le contenu de la méthode *ngOnDestroy()* à la destruction du composant

Permet de libérer la mémoire à la destruction du composant (désinscription aux Observables, fermeture de connexion éventuelle, ...)

```
export class HomeComponent implements OnDestroy {
  ngOnDestroy(): void {
    throw new Error('Method not implemented.');
  }
```

2.6 Intro aux cycles de vie des composants (fin)

Les deux hooks présentés précédemment sont les plus fréquemment utilisés. **OnInit** est d'ailleurs implémenté de base à la création d'un nouveau composant.

Il en existe bien d'autres qui peuvent être utiles à différents moments du développement. Ils sont détaillés dans la doc officielle d'Angular

https://angular.io/guide/lifecycle-hooks

2.7 Les bases de la navigation

Il est nécessaire de spécifier les routes d'accès à nos composants pour pouvoir naviguer de manière fluide. Nous utilisons les "routing-module" pour se faire.

app-routing.module.ts

```
const routes: Routes = [
  { path : 'home', component : HomeComponent}
];
```

HTML

```
<a routerLink="/home">Home</a>
```

```
<a href="/home">Home</a>
```

L'utilisation de l'attribut "***HREF***" implique un rechargement complet du site visible à l'oeil nu, ce qui va à l'encontre du principe de Single Page App.
L'attribut "***ROUTERLINK***" offre un chargement invisible. En réalité, seul le composant appelé par le routing est chargé à l'endroit ou se trouve la balise **<router-outlet></router-outlet>** dans le template HTML

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

2.7 Les bases de la navigation

```
const routes: Routes = [
  { path : 'home', component : HomeComponent, children : [
    { path : 'demo', component : DemoComponent}
  ]}
];
```

Grâce à la propriété “**children**”, il est possible de définir des liens enfants, qui pourrait correspondre à une architecture Dossier/Sous-dossier.
Pour rendre ceci fonctionnel, il ne faudra pas oublier d’ajouter **<router-outlet></router-outlet>** au niveau du composant parent (**HomeCompenent**)

L’exemple précédent donne lieu à l’url :

http://localhost:4200/home/demo

2.7 Les bases de la navigation

Dans le cas d'une navigation inter-module, il sera nécessaire d'importer le routing des modules secondaires au niveau du module principal de l'application pour que celui ci ait accès aux routes déclarées dans les autres.

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

```
{path : 'nouveau', loadChildren: () =>
  import('./nouveau/nouveau.module').then(m => m.NouveauModule)},
```

Il faut également déclarer le lien vers le nouveau module dans le routing du ***AppModule***.

Nous utilisons cette méthode pour créer un **Lazy Loading** entre les modules. Ce qui veut dire que les sous modules et leur composants ne seront chargés qu'à la demande et pas directement au lancement de l'application.

Nous choisissons le Lazy-Loading Vs le Eager-Loading pour des questions de performances

Exercice

Mettre en place 2 nouveaux modules (Demo et Exercice). Comprenant chacun un composant d'amorçage et un routing spécifique.

Créer un composant de navigation et l'inclure au template de base (menu à gauche de la page)

Tous les composants et modules doivent être accessible à partir de la page d'accueil de l'application

3. Bindings

Bindings

3.1 Property Binding

3.2 Event Binding

3.3 Attribute Binding

Table des matières

3.1 Property bindings

Le property-binding crée une liaison entre une propriété Typescript et le template HTML. Il s'agit de faire interagir les 2 fichiers de manières à apporter un certain dynamisme à nos pages.

Il existe 2 types de property-binding :

- One-way => lecture du contenu d'une propriété dans le template visuel
- Two-way => liaison montante et descendante d'une propriété

(Les propriétés déclarée dans le fichier .TS sont par défaut public. Il est possible de changer cette accessibilité, mais une propriété "private" ne pourra pas être accessible dans la page HTML)

3.1 Property bindings : exemple

	Typescript	Html
One Way	`maVariable : string = "World"`	`<p>Hello {{maVariable}} !!!</p>`
Two Way	`maVariable : string = "World"`	`<input type="text" [(ngModel)]="maVariable">`

Si la valeur de *maVariable* est modifiée de quelque manière que ce soit (ts ou html), la mise à jour se fait en temps réel des deux côtés sans rafraichir la page.

Il est nécessaire d'importer FormsModule au niveau du module pour que [(ngModel)] soit disponible

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

3.2 Event Binding

L'event binding permet de lier un événement JS (ex : onClick, onChange, onFocus,....) du template HTML à une méthode définie dans le typescript.

```
maMethode() : void {
  //do something
}
```

```
<input type="button"
       (click)="maMethode()"
       value="Click">
```

Il est évidemment possible de passer des paramètres à notre méthode

3.3 Attribute Binding

Il est possible de conditionner l'apparition d'un attribut grâce à une expression booléenne dans l'html.

```
<input type="button" value="click ici" [disabled]="true">
```

Il est également possible de lier l'attribut à une propriété

TS

```
isClickable : boolean;
```

HTML

```
<input type="button"
       value="click ici"
       [disabled]="isClickable">
```

4. Pipes

Les Pipes

4.1 Les pipes. Comment ça marche ?

4.2 Les principaux pipes

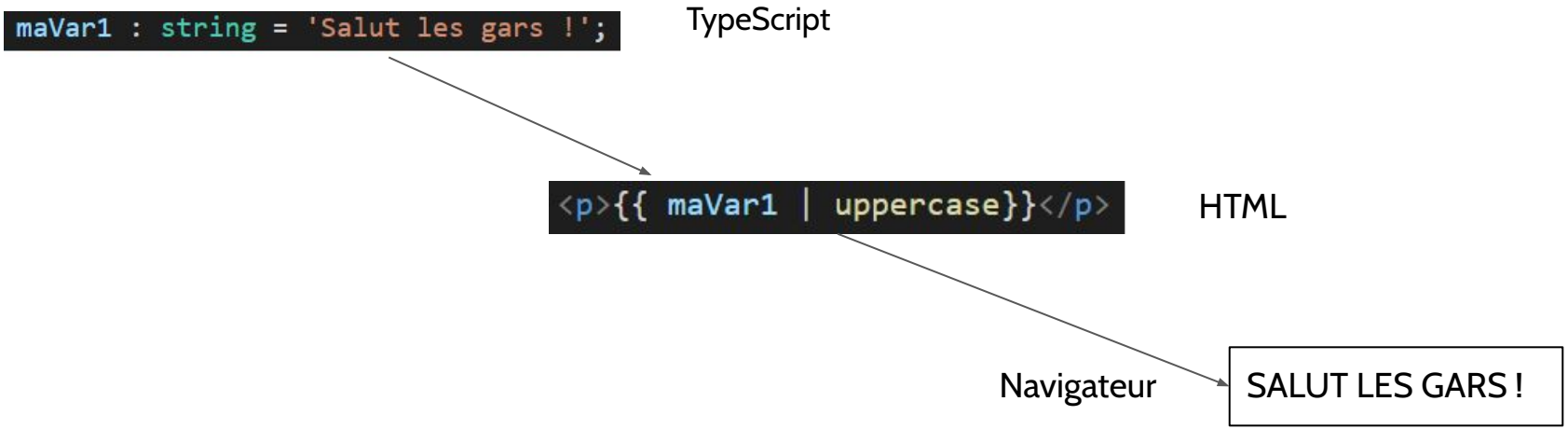
4.3 Custom Pipe

Table des matières

4.1 Les pipes - Comment ça marche ?

Les pipes permettent de formater ou transformer l'affichage d'une propriété dans un binding.

Le nom "pipe" vient du caractère utilisé pour y faire appel



4.2 Les principaux pipes

Pipe	Utilité
| date : 'format'	Formate l'affichage de la date
| uppercase	Transforme la chaîne en majuscule
| lowercase	Transforme la chaîne en minuscule
| titlecase	Le premier caractère de chaque mot en majuscule
| currency	Prend une devise en paramètre (EUR/USD/…)
| json	Convertit la valeur en chaîne Json (utile pour le débuggage)

Il s'agit des principaux pipes utilisés : Liste complète sur la doc officielle

4.3 Les custom pipes

Il est possible de créer des **pipes custom** permettant de répondre à nos attentes précises comme des conversion de valeur, format d'affichage spécifique ou tout autre besoin rencontré.

ng g pipe nom_du_pipe

```
@Pipe({
  name: 'toFahrenheit'
})
```

Le décorateur @Pipe permet de définir le nom d'appel du pipe

```
{{maVar | toFahrenheit }}
```

La classe ainsi créée implémente l'interface PipeTransform

4.3 Les custom pipes

Il est possible de définir nos propres paramètres pour notre pipes. Il demande de définir la méthode *transform()*

```
transform(value: unknown, ...args: unknown[]): unknown {
  return null;
}
```

1 Le premier paramètre(**value : unknown**) représente la propriété sur laquelle le pipe va être appliqué. Il est bien entendu que les types doivent correspondre.

2 Le second (...args: unknown[]), tous les paramètres supplémentaires éventuels qui pourraient accompagner le pipe

3 En dernier lieu, il sera nécessaire de définir le type de retour de notre pipe

4.3 Les custom pipes (Problème)

Nous rencontrons dès lors un problème d'import au niveau de nos différents modules.
Le système vous signifie que votre pipe n'est pas inclu/trouvé par le module Demo.
Il faut recourir au subterfuge du "**SharedModule**".

Autrement dit : un module qui servira de passerelle pour vos différents *pipes, directives, services ou composants* **communs et partagés** entre différents modules. Ce module devra être importé dans chaque module ou vos éléments communs doivent se retrouver

ng g m Shared

SharedModule

```
@NgModule({
  declarations: [
    ToFahrenheitPipe
  ],
  exports : [
    ToFahrenheitPipe
  ]
})
```

DemoModule

```
@NgModule({
  declarations: [DemoComponent,
  imports: [
    CommonModule,
    DemoRoutingModule,
    SharedModule
  ],
  bootstrap : [DemoComponent]
})
```

Exercice

Mettre en place un chronomètre

- 3 Boutons : Start, Pause, Reset (actif/inactif en fonction des besoins)
- Affichage “xx minutes xx secondes”

Astuce : cherchez du côté de setInterval() et du string Interpolation de Typescript ;)

5 Les directives

Les directives

5.1 C'est quoi une directive ?

5.2 Components Directives

5.3 Structural Directives

5.4 Custom Directive

Table des matières

5.1 C'est quoi une directive ?

Les directives sont des classes qui ajoutent un comportement supplémentaire aux éléments de vos applications Angular. Avec les **Built-in directives d'Angular**, vous pouvez gérer les listes, les styles et ce que voient les utilisateurs.

Elles s'appliquent sur des composants/balises html pour en conditionner l'affichage

```
<div *ngIf="isVisible">
    <p>Hello World !!!</p>
</div>
```

Il en existe différents types, et la possibilité d'en créer sois-même.

- Directive de composants (NgClass, NgStyle, NgModel)
- Directive Structurelle (NgIf, NgFor, NgSwitch)
- Directive d'attributs (custom directive)

5.2 Component Directives

Elles permettent de “toucher” à l’affichage de votre page.

- NgStyle (permet d’ajouter du code CSS inline à vos balises)
- NgClass (ajoute une classe CSS en fonction d’une expression booléenne)
- NgModel (Comme déjà vu, permet le binding two-way d’une propriété)

5.3 Structural directives

Permettent d'intégrer des structures conditionnelle dans votre template visuel.

- NgIf (Conditionne l'affichage de contenu sur base d'une expression booléenne)
- NgFor (Permet de répéter un affichage *n* fois. Sert principalement à parcourir des collections)
- NgSwitch (Permet des structure conditionnelle type Switch/case comme son nom l'indique)

5.4 Custom Directives

La création de custom directive se fait via la commande :

ng g directive nom_de_la_directive

```
@Directive({
  selector: '[nom_d_appel]'
})
```

```
<div nom_d_appel>
    Plein de blabla
</div>
```

Il est nécessaire, comme pour les pipes, de déclarer la directive dans le module qui l'utilisera

5.4 Custom Directives

```
@HostListener('mouseenter') onMouseEnter() {
  //Instruction
}
```

Le décorateur @HostListener “écoute” ce qu’il se passe et réagit en fonction de l’ évènement précisé entre ( )

L’utilisation par injection de ElementRef permet d’accéder à l’élément “el” appelant la directive

```
constructor(private el : ElementRef) { }
```

Exercice

Sur base d'un modèle Link, mettre en place un menu de navigation réactif en fonction d'une liste de lien définie dans le TS de nav-component

- Chaque module sera représenté dans le menu (Demo, Exercices) et sera l'occasion d'un sous-menu affichée/cachée sur un click

```
export class Link{
  public title : string;
  public url? : string;
  public children? : Link[];
  public isVisible? : boolean;
}
```

6. @Input @Output

Discussion entre composants

@Input @Output

6.1 Faire discuter les composant

6.2 **@**Input

6.3 **@**Output

Table des matières

6.1 Faire discuter les composants

Nous avons précédemment vu qu'il est possible d'intégrer un composant à un autre grâce à l'appel par sélecteur.

Il nous est, par conséquent, possible de créer un dialogue entre le composant appelant (parent) et le composant appelé (enfant) en nous servant principalement du **sélecteur** et des décorateurs **@Input** et **@Output**

D'envoyer des informations du parent à l'enfant au travers de propriétés.

Et de faire réagir le parent en fonction de ce qu'il se passe dans l'enfant via des **EventEmitter**

6.2 @Input

Grâce au décorateur **@Input** , l'enfant est en mesure de recevoir une information transmise par son parent.

Appel par sélecteur du côté parent en définissant une propriété à transmettre à l'enfant

```
<app-enfant title="Titre du composant Enfant">
</app-enfant>
```

Récupération et utilisation de la propriété dans le composant enfant

TS
```
@Input() title: string;
```

HTML
```
<h3>{{title}}</h3>
```

Le décorateur **@**Input doit être importer de **@**angular/core

6.3 @Output

Le décorateur **@Output** permet de définir une réponse de l'enfant vers le parent au travers d'un *EventEmitter* que nous mettrons en place.

Mise en place de l'EventEmitter côté enfant. Son nom fera office de "type d'évènement" utilisable du côté parent

```
@Output() monEvent : EventEmitter<string>

constructor() {
  this.monEvent = new EventEmitter<string>();
 }
```

Event Binding côté parent pour récupérer la réponse de l'enfant au travers de l'Event mis en place dans ce dernier

```
<app-enfant (monEvent)="reactToChildren($event)">
</app-enfant>
```

Déclenchement de l'event côté enfant

```
this.monEvent.next("paramètres")
```

Le décorateur **@**Output et EventEmitter doivent être importer de **@**angular/core

6.4 NgContent

Du contenu peut être défini dans le parent pour l'enfant. Et afficher dans l'enfant via le sélecteur **<ng-content></ng-content>**

```
<app-enfant>
    <p>Contenu de l'enfant défini dans l'appel côté parent</p>
</app-enfant>
```

```
<h3>{{title}}</h3>
<ng-content></ng-content>
```

Exercice

Mettre en place une shopping list en deux composant.

Dans le premier, un champ texte permettant l'ajout d'un article à la liste
Dans le deuxième, l'affichage de la liste et la possibilité de supprimer un article.

Les deux composant doivent être afficher sur la même page

7 Service et Injection

Les services

7.1 Pourquoi et comment ?

7.2 Injection de dépendances

Table des matières

7.1 Les services pourquoi et comment

Les services permettent de regrouper des fonctionnalités liées entre elles (comme la consommation d'un end-point d'API) pour une meilleure maintenabilité.

Ou encore des fonctionnalités partagées par toute l'application (ex : la gestion des rôles utilisateur ou des méthodes de calcul) => Grand potentiel de réutilisabilité

Ils peuvent également fournir à un module ou un composant les outils nécessaires à leur fonctionnement

En bref, ils constituent le coeur de la logique fonctionnelle de l'application.

Et atout non négligeables, ils bénéficient de l'injection de dépendances native en Angular.

7.1 Les services pourquoi et comment

Créer un service : **ng g s dossier/nom_du_service**

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

  constructor() { }
}
```

Le décorateur **@Injectable** signifie que la classe est injectable en tant que service.
Le paramètres **ProvidedIn** est spécifique à l'injection de dépendances.

7.2 Injection de dépendances

Il s'agit d'un design pattern laissant l'application décidé de la nouvelle instanciation d'une classe de service ou de l'utilisation de l'instance existante.

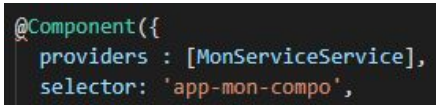
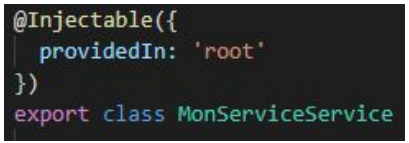
Le framework Angular embarque l'injection de dépendance de manière native. Et nous permet de définir la portée de nos instances (module, composant, application). Il nous faudra dès lors définir deux choses : Le provider du service et ou injecter ce dernier.

Du provider dépend la portée et la durée de vie de l'instance du service. Une fois défini (voir slide suivant), nous utiliserons l'injection par constructeur pour injecter notre service

```
export class MonCompoComponent implements OnInit {

  constructor(private _service : MonServiceService) { }
```

7.2 Injection de dépendances

<table>
<tr><th>Provider</th><th>Portée et durée de vie</th><th>Mise en place</th></tr>
<tr><td>Component</td><td>L'instance n'est disponible que durant la vie du composant. (voir Cycle de vie) Et uniquement dans le composant (et ses enfants) Une nouvelle instance du service sera instanciée à chaque appel de constructeur du composant</td><td>A définir dans le tableau providers de @Component
  </td></tr>
<tr><td>Module eager-loaded (Singleton)</td><td>Disponible pour toute l'application dès son lancement</td><td rowspan="3">Le paramètre providedIn de @Injectable définit le provider du service 'root' ou 'nom_du_module'
 
Ou dans le tableau providers de @NgModule</td></tr>
<tr><td>Module lazy-loaded (Singleton)</td><td>Disponible à partir du moment ou le module est chargé. Peut causer des erreurs s'il est injecté avant que le module ne soit chargé</td></tr>
<tr><td>Root Module (Singleton)</td><td>Disponible pour toute l'application dès son lancement</td></tr>
</table>

Exercice

Optimiser la liste de course avec l'utilisation d'un service qui gère la liste et les méthodes qui peuvent s'y appliquer

Ajouter la possibilité de gérer le nombre d'articles voulu (Ex : Lait x 6, Tomates x 3, ...)

8 Formulaires

Les formulaires

Table des matières

8.1 Les différents formulaires

La plupart des interactions utilisateur se font au travers de formulaires. Nous avons deux options à notre portée.

- **Template-driven Forms => Input** lié à une propriété par ***Two-way binding.***

  Rappel : Il faut importer **FormsModule** au niveau du module ou de l'app

- **Reactive Forms** permettant de gérer des groupes de *contrôle* et de valider les champs de formulaires

  Il est nécessaire d'importer **ReactiveFormsModule** au niveau du module ou de l'app

Nous allons privilégié les Reactive Forms

8.2 Le FormBuilder

La classe intégrée à Angular, **FormBuilder,** nous permet de déclarer facilement nos formulaires réactifs. Elle devra être injecté dans le constructeur. L'objet ainsi instancié s'occupera de gérer les "***contrôle***" de formulaire et d'y inclure des "***Validator***"

```
constructor(
  private _formBuilder : FormBuilder
) { }
```

Sans oublier de l'importer de ***@angular/forms***

L'objet **_formBuilder** nous propose dès lors 3 possibilités : array, control et group

8.3 formControl

Le contrôle est l'unité de base du formulaire, il représente un input présenté dans l'html

```
let myControl = new FormControl(null, Validators.required)
```

Le premier paramètre de construction permet de définir la valeur d'initialisation contenue dans l'input lié à notre contrôle.

Le second représente le/les **Validator**(s) que nous voudrions appliquer au contrôle.
=> voir plus loin dans le chapitre

```
let myControl = this._formBuilder.control(null, [Validators.required])
```

Nous préférons laisser le soin au **FormBuilder**, de gérer la création et l'instance de notre contrôle. Le laissant jouer son rôle de chef d'orchestre pour les **Reactive Forms**

8.3 formControl

Une fois notre contrôle défini côté TS, nous devrons le lié à un Input de formulaire par l'attribut **formControlName**

```
<form>
    <input type="text" formControlName="myControl">
</form>
```

Remarque : Nous utiliserons très rarement un contrôle seul. Nous préférons utiliser l'option du Template-Driven Form le cas échéant

8.4 formGroup

Un **FormGroup** regroupe plusieurs **FormControl**, voir un formulaire complet pour aider à la validation globale de celui ci.

```
myFormGroup : FormGroup
```

Déclarons une propriété contenant le **FormGroup**

```
this.myFormGroup = this._formBuilder.group([
  {control1 : [null, Validators.required]},
  {control2 : [null, Validators.required]},
])
```

Ensuite, nous initialisons le groupe en laissant le **FormBuilder** se charger de tout.
La méthode **.group()** permet d'instancier un tableau d'objet de type **FormControl** très simplement

8.4 formGroup

La mise en place côté HTML d'un formGroup est très simple.

- Lier le **formGroup** grâce à la directive **[formGroup]** du Formulaire (ce qui a pour effet de remplacer l'attribut "**action=page.xxx**" de <form> et de lier facilement les **formControl** au différents éléments).

```
<form [formGroup]="myFormGroup">
    <input type="text" formControlName="control1">
    <input type="text" formControlName="control2">
</form>
```

- Lier ensuite chaque **Input** au **formControl** qui lui est dédié

8.5 formArray

Permet de créer un tableau de contrôle. Dans le cas ou on ignore le nombre de contrôles à créer ou si de nouveaux contrôle doivent être ajouté en cours de route

```
this.myFormGroup = this._formBuilder.group(
  {
    'myArray': this._formBuilder.array([])
  }
```

Une fois déclaré, il est nécessaire de pouvoir y accéder :

```
getArray() : FormArray {
  return this.myFormGroup.get('myArray') as FormArray
}
```

Le type FormArray représente une collection de FormControl. Et qui dit collection, dit push(), remove(), ...

8.5 formArray

Notre FormArray, désormais facilement accessible, peut être manipuler comme n'importe quelle collection

Ajout d'un contrôle :

```
this.getArray().push(new FormControl(null, null))
```

Affichage des contrôle :

```
<div *ngFor="let c of getArray().controls;
             let i = index">
    <p>{{i+1}} <input type="text"
                      [formControlName]="i"></p>
</div>
```

Remarque : Le nom de chaque contrôle correspond à son index dans la collection

8.6 Validators principaux

Ils vont permettre de vérifier que les valeurs entrées correspondent à ce qui est attendu.

Validators.min(X)	Oblige une valeur numérique minimale X
Validators.max(X)	Limite à une valeur maximale X
Validators.required	Force une valeur non vide
Validators.email	Impose de respecter le format E-mail “x@x.x”
Validators.minLength(X)	Longueur minimale de X de la valeur (chaîne de caractère)
Validators.maxLength(X)	Longueur maximale de X de la valeur (chaîne de caractère)
Validators.pattern(regex)	Oblige à correspondre à une expression régulière

8.7 Validation de formulaire

Grâce à nos objets **formGroup** ou **formContol**, nous avons accès à une propriété **valid** qui retourne **True/False** en fonction des validations imposées par nos **validators.** Ce qui nous permet d'afficher ou non des messages d'erreur (*ngIf) ou de soumettre ou non le formulaire.

Afficher un message d'erreur

```
<p><input type="email"
    formControlName="control2">
</p>
<p *ngIf="!myFormGroup.get('control2').valid">
    Pas un e-mail valide
</p>
```

Autoriser la validation du formulaire

```
<p><input type="submit"
    [disabled]="!myFormGroup.valid"
    value="Valider"
    >
</p>
```

8.8 Soumission et récupération de données

Une fois validé, nous soumettons le formulaires via un ***Event Binding*** **(ngSubmit)=”method()”**.

```
<form [formGroup]="myFormGroup"
      (ngSubmit)="SubmitForm()">
```

Et nous récupérons les données grâce à la propriété “**value**” du FormGroup.

```
SubmitForm() {
  let values = this.myFormGroup.value
  let myProp1 = values["control1"]
  let myProp2 = values["control2"]
  console.log(myProp1+" "+myProp2)
}
```

8.9 Custom Validator

Angular nous donne la possibilité de définir nos propres **Validators**.

Il suffit de mettre en place une méthode qui retourne un objet de type ValidatorFn (fonction de validation)

```
MaValidation() : ValidatorFn | null {
  return (control : FormControl) => {
    //Code fonctionnel
    return null
  }
```

Il reste à appeler cette fonction de validation dans la liste des Validators de notre champ

```
myControl1 : [null, [this.MaValidation()]],
```

Remarque : La best-practice nous suggère d'externaliser ces Custom Validators dans un but de réutilisabilité

9 Routing

Routing

Table des matières

9.1 Routing module

Rappel :

Comme vu précédemment, chaque module **peut** inclure son propre routing. Bien que non nécessaire, il est conseillé d'adopter cette pratique pour chaque module regroupant plusieurs composants. (ex : Admin panel, DashBoard, User feature, ...)

Nous prendrons donc le parti mettre en place des "children" et nous retrouver avec un template de route similaire à : "monsite.com/module/composant".

Dans le cas d'un module d'import/export tel que le "SharedModule" mis en place. Aucun routing n'est nécessaire.

9.1 Routing module

Les routes sont définies dans le tableau d'objet de type **Routes**. Pour qu'une route soit valide, l'objet attend au minimum une propriété "path" qui défini le chemin d'accès. La suite des paramètres dépend des besoins.

Le décorateur @NgModule du routing est nécessaire pour importer RouterModule et en définir la portée.

- forRoot(routes) => il s'agit du routing principal de l'application
- forChild(routes) => tout sous-module de routing dépendant de son parent

9.1 Routing module

Dans la propriété “path” des routes, il est possible de définir un chemin par défaut.

path : ‘’** => signifie que les chemins non définis précédemment seront prit en charge par cette route.

```
{path : '**', redirectTo : 'notFound'},
```

Nous pouvons dès lors rediriger vers une autre route (*redirectTo*).

path : ‘’ => représente le chemin vide, il est donc possible de rediriger vers l’endroit voulu en cas d’url *http://monsite.com/* . Autrement dit, nous choisissons le point d’entrée

9.2 Eager VS Lazy Loading

Chaque module (excepté le root) peut être défini, au sein du routing principal, comme charger au démarrage de l'application (Eager-loaded) ou charger à la demande (Lazy-loaded).

Dans un soucis de performance, le lazy-loading est recommandé. En effet, si un module est inutile lors d'un traitement ou de la navigation d'un utilisateur (ex : Adminpanel). Il n'est pas nécessaire de surcharger le serveur.

En cas de lazy loading. Il est important de bien gérer l'injection de dépendances et en particulier les providers de service. (Un service dont le provider n'est pas chargé, n'existe pas)

9.3 Router

La classe Router permet de gérer les routes au sein d'un composant ou d'un service

Il est nécessaire d'injecter la classe **Router** dans le constructeur du composant ou service qui l'utilisera

```
constructor(
  private _router : Router
) { }
```

```
this._router.navigate(['maRoute/'+this.parameter])
```

Nous pouvons envoyer l'utilisateur vers la route voulue, avec la possibilité d'ajouter des paramètres et les traiter dans le routing module

```
{path : 'maRoute/:param'}
```

9.4 ActivatedRoute

La classe **ActivatedRoute** représente la route active (ou qui vient d'être emprunter lors de la navigation). Il est possible d'y récupérer des paramètres de route. Des objets chargé par Resolver ou simplement la connaissance d'où on vient.

```
this.parameter = this._activatedRoute.snapshot.params['param']
```

Grâce au tableau params se trouvant dans la propriété snapshot. Nous récupérons facilement nos paramètres de route.

Remarque : **la propriété snapshot représente la route à l'instant T**

9.5 Guard

Il existe 4 types de gardes :

- **CanActivate** : *puis-je accéder au composant ?*
- **CanDeactivate** : *puis-je quitter le composant ?*
- **CanActivateChild** : *Puis-je accéder au "children" d'un path ?*
- **CanLoad** : *Puis-je accéder au module ciblé par le path ?*

Il nous est possible de définir si une route peut être activée ou pas. Peut-on accéder au composant/module voulu, ou pas ?

- **ng g guard nom_de_guard**

Quelque soit le type de garde choisi, il faudra utiliser son type dans la déclaration du path à garder dans le routing

- return True => route activée
- return False => route inaccessible

activate : GuardName (dans les paramètres d'une route pour activer la garde)

```
export class MyGuard implements CanActivate
```

Exercice

Mettre en place un CRUD de fan de série.

Chaque fan sera défini par son nom, sa date de naissance(interdit au moins de 13 ans), et la liste de leur séries préférées (juste le titre).

Aucune limitation du nombre de séries définie par l'utilisateur à la création de son profil

Sur la page de modification de profil, il doit être possible d'ajouter ou supprimer des séries.

- 4 composants (liste des fans, détails, création, mise à jour)
- Seule l'année est vérifiée pour l'âge des fans
- Si un champ est visible, il sera obligatoirement rempli

10 Storage

Les storages

10.1 LocalStorage vs SessionStorage

10.2 Méthodes associées

Table des matières

10.1 Différence entre les deux storages

Le sessionStorage garde les infos voulues jusqu'à ce que le navigateur ou l'onglet soit fermé

Le localStorage quand à lui est disponible jusqu'à ce que l'application vide le storage ou que l'utilisateur vide le cache de son navigateur.

Pour les deux : stockage jusqu'à 10mo pour la plupart des navigateurs

10.2 Méthodes associées

Qu'il s'agisse du **local** ou du **session** storage, les commandes sont identiques. Seul le comportement global est différent

`.setItem('key', this.value)`	Ajoute un valeur en session (de type string uniquement)
`.getItem('key')`	Récupère la valeur stockée à la clé "key"
`.removeItem('key')`	Supprime une clé et la valeur associée de la session
`.clear()`	Vide complètement la session (clés et valeurs)

11 Observable

Les observable

11.1 La librairie RXJS

11.2 Subject / BehaviorSubject

11.3 Subscribe()

11.4 Subscription

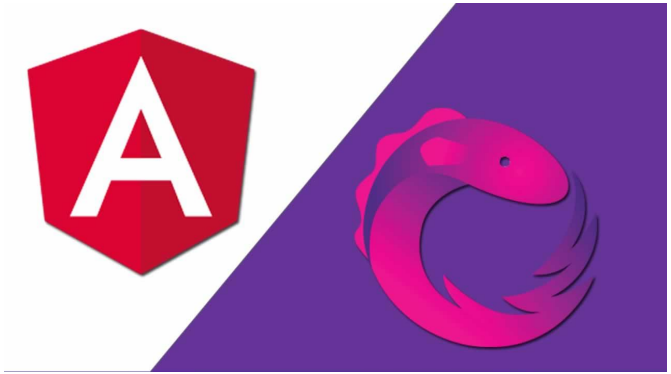
11.5 Opérateur RXJS

Table des matières

11.1 La librairie RXJS

Faisant partie du bundle de base d'Angular, cette librairie contient tout le nécessaire pour gérer les **Observable**. Ceux-ci sont le fondement de la programmation dite réactive et permettent en gros, de ne plus avoir à charger manuellement une valeur si elle est observée

Au delà de ça, elle permet de gérer des comportements asynchrones (appel d'api par exemple) et amène des outils de gestion de collection observable



https://www.learnrxjs.io/

Remarque : Ce chapitre ne couvre pas l'entièreté des possibilités de RXJS mais apporte l'essentiel pour la compréhension des observable

11.2 Subject et BehaviorSubject

Pour que plusieurs composants puissent se baser sur le même service et donc les mêmes valeurs. Il sera nécessaire d'utiliser la notion d'observable.

Nous verrons dans le chapitre suivant que les appels api (HttpClient) retourne d'office un observable. Mais dans le cas de propriétés ou d'objets gérés par l'application elle-même, il faut mettre en place soi-même cette notion.

C'est là qu'interviennent les Subject

```
mySubject : Subject<any> =
  new Subject<any>();
```

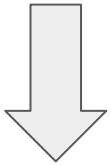
```
private property : boolean = false;

mySubject : BehaviorSubject<boolean> =
  new BehaviorSubject<boolean>(this.property);
```

Autre avantage non négligeables, nous accédons à la notion d'abstraction (plus d'accès direct à une propriétés) et pouvons dès lors utiliser l'encapsulation de manière efficace.

11.2 Subject et BehaviorSubject

```
this.mySubject.next(this.property)
```



La méthode next de mySubject , qu'il soit Subject où BehaviorSubject, enverra la valeur de property à tous les abonnés (voir slide suivant)

Les types Subject et BehaviorSubject permettent de d' émettre la valeur d'une propriétés aux différents composant abonné à ce subject.

La différence fondamentale entre les deux types est que le BehaviorSubject émet sa valeur par défaut lors de sa construction. Contrairement au Subject.

Pour les deux, à chaque nouvelle modification, il faudra émettre la valeur.

Le BehaviorSubject à également la particularité de garder sa dernière valeur émise en mémoire. Elle est accessible via la méthode GetValue()

11.3 Subscribe

Pour suivre l'évolution d'un observable, il sera nécessaire de s'y abonner via la méthode subscribe() disponible pour n'importe quel type observable

```
this._myService.mySubject.subscribe(
  (value : boolean) => this.property = value)
```

```
this.mySubject.subscribe({
  next : () => {/*réussite de l'observable*/},
  error : () => {/*Gestion des erreurs*/},
  complete : () => {/*Quoi qu'il arrive*/}
})
```

La méthode subscribe() attend jusqu'à trois lambas,

1) le paramètre d'entrée représente la valeur émise par l'observable. Et dont l'expression représente le traitement exécuté à chaque nouvelle émission de l'observable

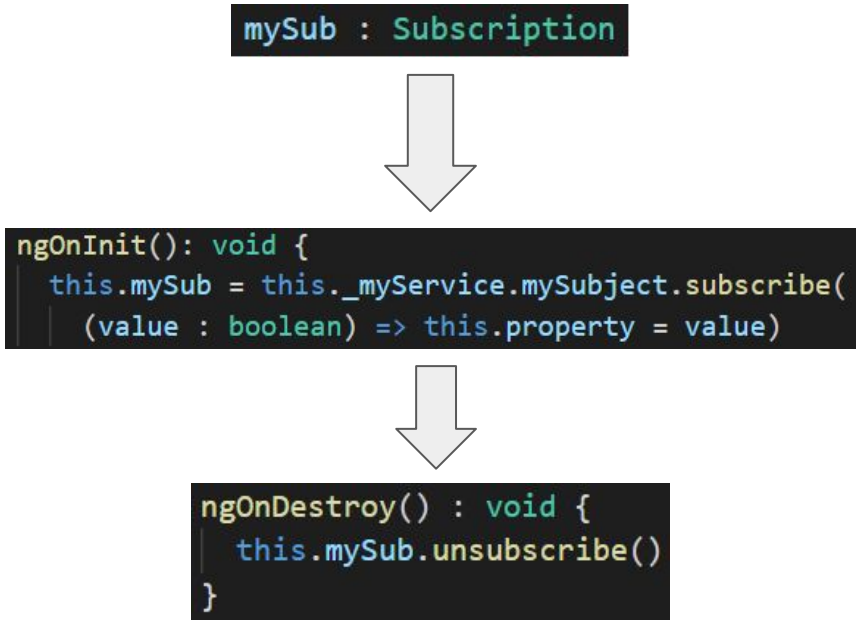
2) le paramètre d'entrée représente l'éventuel erreur retournée par l'observable et l'expression, le traitement à faire en cas d'erreur.

3) Représente l'action à effectuer qu'il y ai des erreurs ou pas

Nous pouvons comparer ça à un bloc "try catch finally"

11.4 Subscription

L'objet **Subscription** permet de stocker le statut de la méthode **subscribe()** . Ce qui veut dire qu'il est possible d'agir sur cette souscription pour s'en désabonner par exemple.



11.5 Opérateurs RXJS

Nous sommes capable de nous abonnés à des observables. Il temps d'apprendre à s'en servir.

Il est possible d'appliquer différents opérateurs à l'observable avant de déclarer la souscription.

Au travers de la méthode **.pipe()**

```
this._myService.mySubject
  .pipe(/*appliquer les oparéteurs*/)
  .subscribe(/*Gérer l'abonnement*/)
```

Ce cours ne couvre pas la liste des opérateurs.
La liste exhaustive se trouve ici :
https://www.learnrxjs.io/learn-rxjs/operators

12 HttpClient

HttpClient

12.1 Discussion API <-> Angular

12.2 HttpVerbs de base

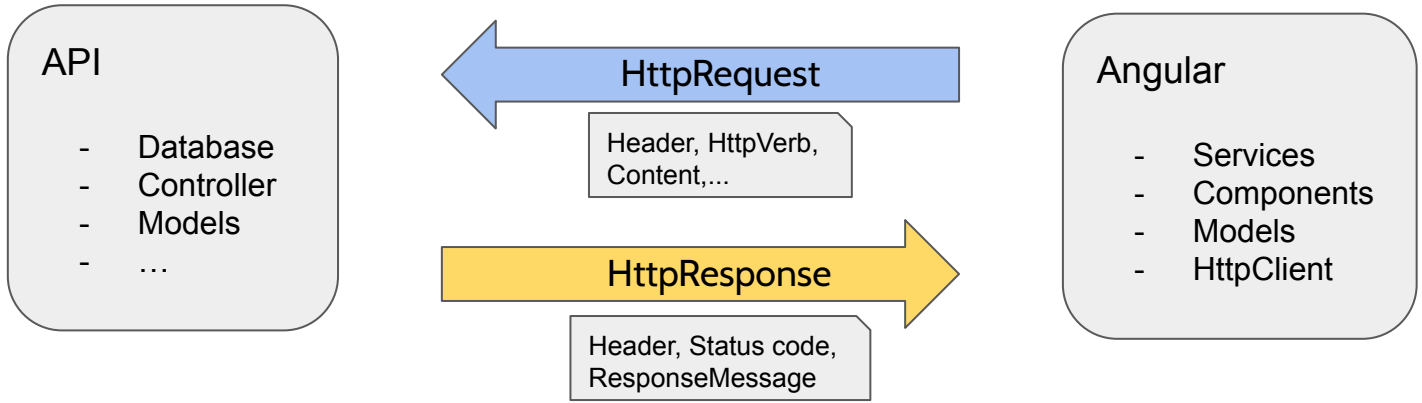
12.3 Resolver

12.4 Interceptor

Table des matières

12.1 Api <> Angular

La discussion entre API et Angular passe par la le module **HttpClient** (=> importer **HttpClientModule**)



Remarque : **Chaque action Http retourne un objet de type Observable<T>**

12.2 Http Verbs

Verbe	Utilité	Exemple Angular
Get	Récupération de données	`this._httpClient.get(this.url)`
Post	Envoi de données (possibilité d'en recevoir en retour)	`this._httpClient.post(this.url, {prop1 : "content"})`
Put	Modification de données	`this._httpClient.put(this.url, {myProp : "content"})`
Delete	Suppression de donnée	`this._httpClient.delete(this.url + id)`

Attention:
S'il y a objet en httpResponse, il est nécessaire de typer la méthode en ajoutant le type entre chevron (ex : get<T>)

Un paramètre optionnel supplémentaire permet de donner des infos de header (comme "Authorization", "Content-Type", ...)

12.3 Resolver

Le resolver permet de “résoudre un objet” durant la navigation.

Exemple :

- Chargement d’un objet venant d’un service
- Résoudre un appel API

Permet d’éviter les erreurs d’objet “undefined” lors d’appel asynchrone d’objet à envoyer au template visuel

On peut dire qu’il s’agit d’un “time-stop” durant la navigation pour loader un objet avant d’arriver à destination

12.3 Resolver

```
export class ResolverService implements Resolve<any>
```

1. Déclarer le service de résolution et son comportement

2. Définir la propriété "***resolve***" de la route pour y exposer le service de resolver et spécifier un nom pour l'objet résolu

```
{ path : 'resolved/:index',
  resolve : {monObjet :ObjectResolverService},
  component : ResolvedComponent }
```

```
this._activatedRoute.snapshot.data['monObjet']
```

3. Récupérer l'objet résolu grâce à **ActivatedRoute,** dans le composant cible

12.4 Interceptor

L'objet **interceptor** va, comme son nom l'indique, intercepter les requêtes Http et les retourner "modifiées". Pour, par exemple y inclure des informations de Header.

```
export class TokenInterceptor implements HttpInterceptor
```

```
intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>>
```

Le paramètre **req** correspond à la requête interceptée, quant au paramètre **next**, il fait référence à la requête une fois modifiée

12.4 Interceptor (suite)

Il faudra ensuite préciser au niveau des providers du module la classe d'intercepteur à utiliser.

provide : précise ce qui doit être gérer (ici l'interception HTTP).

useClass : l'instance de classe à utiliser

multi : (true/false par défaut). Définit s'il est possible d'avoir plusieurs instances de la classe en même temps

```
providers: [
  {provide: HTTP_INTERCEPTORS, useClass: TokenInterceptor, multi: true}
],
```

13 Publication

13 Build de l'application

Il est très facile de build son application pour déployer sur n'importe quel serveur pouvant héberger de l'html.

Il suffit de tapper la commande

ng build

Et d'y ajouter l'option **--prod** pour minifier le code.
Le build est alors déposer dans un répertoire /dist.
Copiez en le contenu sur votre hébergeur et le tour est joué.

Attention : Si vous ne déposez pas le projet à la racine de votre serveur, il faudra spécifier le chemin d'accès dans le fichier index.html à la ligne :

```
<base href="/">
```

en remplaçant le / par la route nécessaire

Merci pour votre attention.

CogniTIC

Building Together Better IT Solutions