

Cours de Angular

Plan

Cours 1

- **Presentation Angular**
- **Projet Angular**
 - **Installation des Outils**
 - **Architecture d'un projet Angular**
 - **Création d'un nouveau projet angular**
 - **Structure d'un projet Angular**
 - **Installation et configuration de quelques modules**
 - **bootstrap**

Cours 2 : Modules,Components et Routing

- **Projet Fil Rouge**
- **Modules**
 - **Définition**
 - **Types Module**
 - **Module à chargement automatique**
 - **Module Lazy Loading**
 - **Définition des modules du Projet**
 - **Génération des modules du projet**
- **Composants**
 - **Définition**
 - **Définition des composants du Projet**
 - **Génération des composants du projet**
 - **Intégration des Composants**
- **Navigabilité ou routing**
 - **Routes principales**
 - **Génération des modules de routing**
 - **Configuration des routes du projet**

Cours 3 : Typescript et Realisation des Modeles

A) TypeScript

- a) Introduction**
- b) Variables**
- c) Constante**
- d) Fonctions**
- e) Fonctions fléchées**
- f) POO**

Cours 4 : Le DataBinding et les Services

A) Directives de Structures

B) Le DataBinding

- i) Notion de Binding**
- ii) Property Binding**
- iii) Event Binding**
- iv) Two way Binding**

C) Notion Api Rest

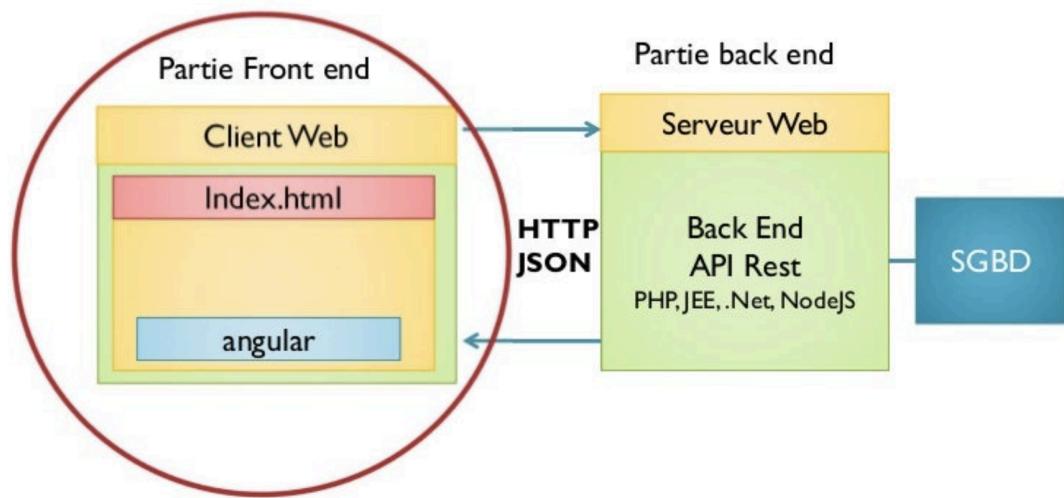
D) Les Services

- v) Définition**
- vi) Creation**
- vii) Injection de Dépendance dans les constructeurs**

A) Presentation Angular

Angular

- Angular Permet de créer la partie front end des applications web de type SPA (Single Page Application réactive)
- Une SPA est une application qui contient une seule page HTML (index.html) récupérée du serveur.
- Pour naviguer entre les différentes parties de cette application, Java Script est utilisé pour envoyer des requêtes http (AJAX) vers le serveur pour récupérer du contenu dynamique généralement au format JSON.
- Ce contenu JSON est ensuite affiché côté client au format HTML dans la même page.



Angular 1, 2, 4, 5

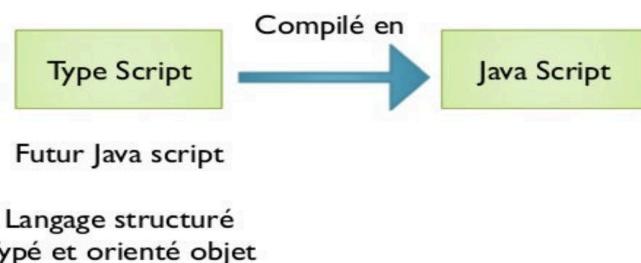
- **Angular 1 (Angular JS) :**
 - Première version de Angular qui est la plus populaire.
 - Elle est basé sur une architecture MVC coté client. Les applications Angular 1 sont écrite en Java Script.
- **Angular 2 (Angular) :**
 - Est une réécriture de Angular 1 qui est plus performante, mieux structurée et représente le futur de Angular.
 - Les applications de Angular2 sont écrites en Type Script qui est compilé et traduit en Java Script avant d'être exécuté par les Browsers Web.
 - Angular 2 est basée sur une programmation basée sur les Composants Web (Web Componenet)
- **Angular 4, 5, 6 sont de simples mises à jour de Angular 2 avec des améliorations au niveau performances.**

med@youssfi.net

NB : Actuellement on est à la version 15 de angular

Angular avec Type Script

- Pour développer une application Angular il est recommandé d'utiliser Type Script qui sera compilé et traduit en Java Script.
- Type Script est un langage de script structuré et orienté objet qui permet de simplifier le développement d'applications Java Script



med@youssfi.net

Type Script

- **TypeScript** est un langage de programmation libre et open source développé par Microsoft qui a pour but d'améliorer et de sécuriser la production de code JavaScript.
- C'est un sur-ensemble de JavaScript (c'est-à-dire que tout code JavaScript correct peut être utilisé avec TypeScript).
- Le code TypeScript est transcompilé en JavaScript, pouvant ainsi être interprété par n'importe quel navigateur web ou moteur JavaScript.
- Il a été cocréé par Anders Hejlsberg, principal inventeur de C#.
- TypeScript permet un typage statique optionnel des variables et des fonctions, la création de classes et d'interfaces, l'import de modules, tout en conservant l'approche non-constrictrice de JavaScript.
- Il supporte la spécification ECMAScript 6.

B) Avantages et Inconvénients

a) Avantages

Angular est un framework de développement d'applications Web open-source basé sur TypeScript. Il est conçu pour faciliter la création d'applications Web complexes en offrant une structure solide et des fonctionnalités avancées pour la gestion des données, la navigation, le routage et la mise en œuvre de l'interface utilisateur.

Il y a plusieurs raisons pour lesquelles les développeurs choisissent d'utiliser Angular pour leurs projets :

- Angular offre un environnement de développement structuré et cohérent qui permet aux développeurs de créer des applications Web plus rapidement et de manière plus fiable.
- Angular utilise TypeScript, ce qui permet de bénéficier des avantages de la programmation orientée objet et de la vérification statique du code pour éviter les erreurs courantes.
- Angular offre une série d'outils et de fonctionnalités avancées pour gérer les données et la logique de l'application, ainsi que pour mettre en œuvre des interfaces utilisateur complexes et réactives.
- Angular est un framework populaire et largement utilisé, ce qui signifie qu'il y a une forte communauté de développeurs et une grande quantité de ressources en ligne pour vous aider à apprendre et à utiliser Angular.

b) Inconvénients

Comme tous les frameworks et outils de développement, Angular a ses propres points faibles que les développeurs doivent prendre en compte lors de la sélection d'un framework pour leur projet. Voici quelques-uns des points faibles les plus courants d'Angular :

- Angular est un framework **complexe** qui peut être difficile à apprendre pour les développeurs débutants. Il y a une forte courbe d'apprentissage et il peut être difficile de maîtriser toutes les fonctionnalités avancées d'Angular.
- Angular est un framework volumineux qui peut entraîner des temps de chargement plus longs pour les applications et un impact sur les performances de l'application.
- Angular suit une approche très structurée pour la mise en œuvre des applications, ce qui peut être inflexible pour certains projets ou approches de développement.
- Angular est un framework en constante évolution, ce qui peut entraîner des changements de comportement entre les versions et des efforts de mise à jour importants pour les projets existants.

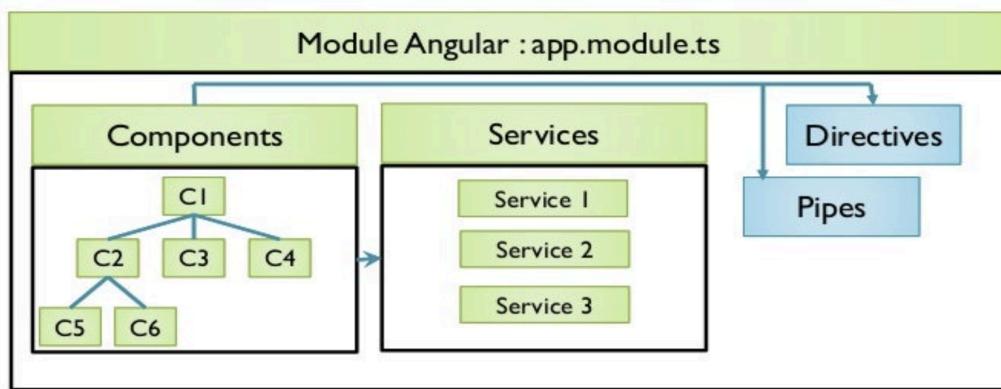
C) Cr ation d'un Projet Angular

Installation des outils

- Pour faciliter le d veloppement d'une application Angular, les outils suivant doivent  tre install s :
 - Node JS : <https://nodejs.org/en/download/>
 - Node JS installe l'outil npm (Node Package Manager) qui permet de t l charger et installer des biblioth ques Java Script.
 - Installer ensuite Angular CLI (Command Line Interface) qui vous permet de g n rer, compiler, tester et d ployer des projets angular (<https://cli.angular.io/>) :
 - **npm install -g @angular/cli**

Architecture de Angular

- Angular est un Framework pour créer la partie Front End des applications web en utilisant HTML et JavaScript ou TypeScript compilé en JavaScript.
- Une application Angular se compose de :
 - Un à plusieurs modules dont un est principal.
 - Chaque module peut inclure :
 - Des composants web : La partie visible de l'application Web (IHM)
 - Des services pour la logique applicative. Les composants peuvent utiliser les services via le principe de l'injection des dépendances.
 - Les directives : un composant peut utiliser des directives
 - Les pipes : utilisés pour formater l'affichage des données dans les composants.



Création d'un nouveau projet Angular

- Pour générer la structure d'un projet Angular, on utilise Angular CLI via sa commande `ng new` suivie des options `new` et le nom du projet.
 - `ng new FirstApp`
- Cette commande génère les différents fichiers requis par une application basique Angular et installe aussi toutes les dépendances requises par ce projet.

Exécuter un projet Angular

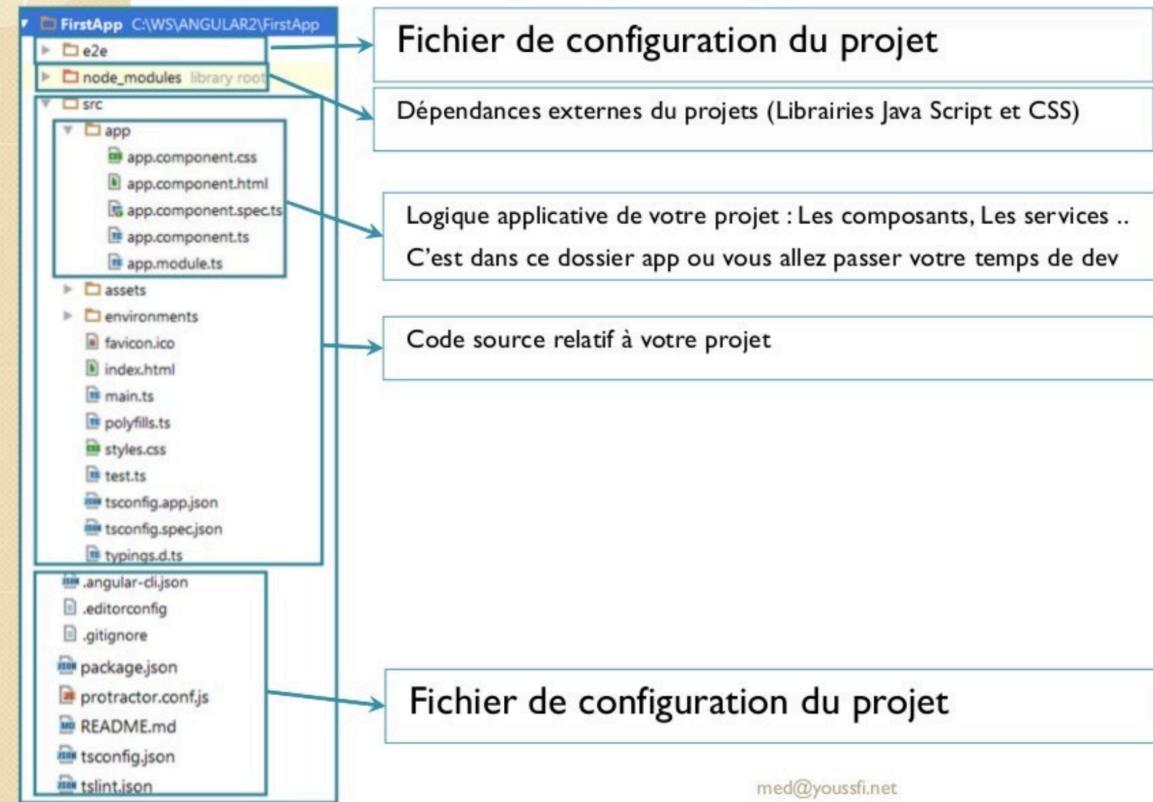
- Pour exécuter un projet Angular, on exécute la commande suivante à partir de la racine du projet
 - `ng serve`
- Cette commande compile le code source du projet pour transpiler le code TypeScript en Java Script et en même temps démarre un serveur Web local basé sur Node JS pour déployer l'application localement.
- Pour tester le projet généré, il suffit de lancer le Browser et taper l'url : `http://localhost:4200`
- Dans l'étape suivante, nous allons regarder la structure du projet généré par Angular CLI.



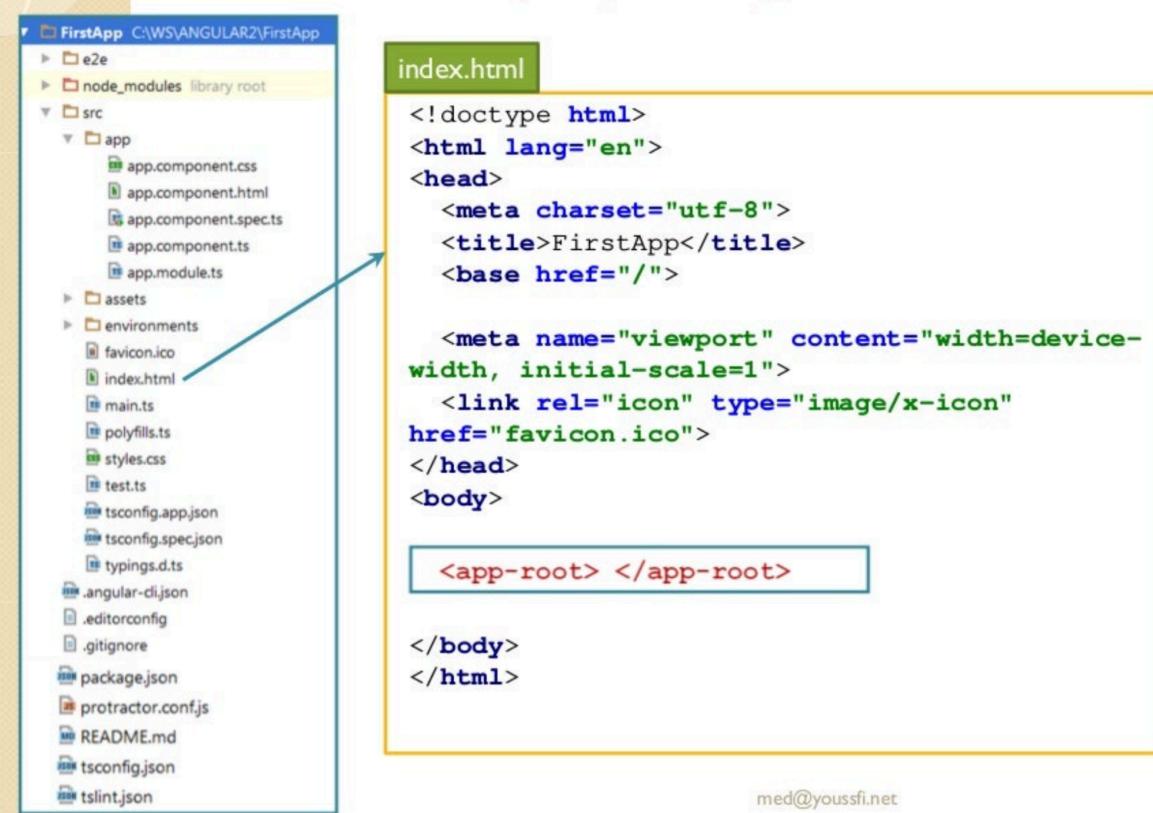
Edition du projet

- Plusieurs IDE professionnels peuvent être utilisés pour éditer le code:
 - Web Storm, PHP Storm
 - Visual Studio Code
 - Eclipse avec plugin Angular
- D'autres éditeurs classiques peuvent également être utilisés :
 - Atom
 - Sublime Text
 - Etc ...

Structure du projet Angular



Structure du projet Angular



Structure du projet Angular

```
main.ts
```

```
import { enableProdMode } from '@angular/core';
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

import { AppModule } from './app/app.module';
import { environment } from './environments/environment';

if (environment.production) {
  enableProdMode();
}

platformBrowserDynamic().bootstrapModule(AppModule);
```

med@youssfi.net

Structure du projet Angular

```
app.module.ts
```

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';

import { AppComponent } from './app.component';

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

med@youssfi.net

Structure du projet Angular

app.component.ts

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

app.component.html

```
<div style="text-align:center">
<h1>
  Welcome to {{title}}!!
</h1>
</div>
```

app.component.css

Les fichiers spec sont des tests unitaires pour vos fichiers source. La convention pour les applications Angular2 est d'avoir un fichier .spec.ts pour chaque fichier .ts. Ils sont exécutés à l'aide du framework de test javascript Jasmine via le programme de tâches Karma lorsque vous utilisez la commande '**ng test**'.

app.component.css (Screenshot of a browser window showing the output: 'Welcome to app!!')

Démarrage de l'application

- Le module racine est démarré dans le fichier main.ts
- Par défaut le module racine s'appelle AppModule

```
import { enableProdMode } from '@angular/core';
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

import { AppModule } from './app/app.module';
import { environment } from './environments/environment';

if (environment.production) {
  enableProdMode();
}

platformBrowserDynamic().bootstrapModule(AppModule);
```

Installation d'un framework css : Bootstrap

1) Installation bootstrap 5

a) Commande

```
npm i bootstrap@5.0
```

b) configuration Bootstrap

- Méthode 1: Dans angular.json

- Ajout du fichier bootstrap.min.css

```
"styles": [
```

```
  "src/scss/styles.scss",
  "./node_modules/bootstrap/dist/css/bootstrap.min.css",
],
```

- Méthode 2: Dans style.css

```
@import "~bootstrap/dist/css/bootstrap.css";
```

- Ajout du fichier bootstrap.min.js

```
"scripts": [
```

```
  "./node_modules/bootstrap/dist/js/bootstrap.min.js"
]
```

2) Installation popperjs

a) Commande

```
npm install -g @popperjs/core
```

b) Configuration dans angular.json

```
"scripts": [
```

```
  "./node_modules/@popperjs/core/dist/umd/popper.min.js",
  "./node_modules/bootstrap/dist/js/bootstrap.min.js"
]
```

Projet Fil Rouge

Mettre en place un projet de e-commerce, le projet offre les fonctionnalités suivantes :

En tant que Admin je peux :

- Ajout d'un Nouveau Produit
- Liste des produits
- Détail d'un produit
- Liste des produits par Catégorie
- Lister les commandes en cours
- Filtrer les Commande par client et par date
- Lister les Categories
- Ajout d'une Nouvelle Catégorie

En tant que Client je peux :

- Affichage des commandes d'un client
- Affichage du panier de commande
- Filtrer les Commande d'un client par date
- Enregistrement d'une nouvelle commande avec la possibilité de créer un compte
- Affichage du panier de commande
- Créer un compte de connexion

En tant que Visiteur je peux :

- Lister le catalogue de produit
- Lister les détails d'un produit
- Liste des produits par Catégorie

Un client ou un admin doit se connecter pour accéder au système.

Notion de Modules

a) Définition

Modules

- Les applications angulaires sont modulaires
- Angular possède son propre système de modularité appelé modules angulaires ou NgModules.
- Chaque application Angular possède au moins une classe de module angulaire: le module racine, appelé classiquement **AppModule**.
- Un module angulaire est une classe avec un décorateur **@NgModule**.
- Les décorateurs sont des fonctions qui modifient les classes JavaScript.
- Angular possède de nombreux décorateurs qui attachent des métadonnées aux classes pour configurer et donner le sens à ces classes.

src/app/app.module.ts

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
@NgModule({
  imports: [ BrowserModule ],
  providers: [ Logger ],
  declarations: [ AppComponent ],
  exports: [ AppComponent ],
  bootstrap: [ AppComponent ]
})
export class AppModule {}
```

@NgModule

- **@NgModule** est un décorateur qui prend en paramètre un objet javascript qui contient des métadonnées dont les propriétés décrivent le module. Les propriétés les plus importantes sont:
 - **declarations** : la classe représentant le module. Angular a trois types de classes de modules : **components**, **directives**, and **pipes**.
 - **exports** – Pour exporter les classes utilisables dans d'autres modules.
 - **imports** – Pour importer d'autres modules.
 - **providers** – Pour déclarer les fabriques de services.
 - **bootstrap** – Pour déclarer le composant Racine du module. Seul le module racine doit définir cette propriété.

src/app/app.module.ts

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
@NgModule({
  imports: [ BrowserModule ],
  providers: [ Logger ],
  declarations: [ AppComponent ],
  exports: [ AppComponent ],
  bootstrap: [ AppComponent ]
})
export class AppModule {}
```

NB:

Dans une application angular on peut utiliser seulement le module de démarrage **app.module.ts** générée par défaut lors de la création d'un projet angular mais la subdivision d'un projet en modules offre les avantages suivantes :

- Meilleur manière d'organiser les fonctionnalités d'un projet
- Facilité de Maintenance et de réutilisation des modules
- Implémentation du Lazy Loading
- Aide à la sécurisation de nos Pages

b) Types Module

Dans angular on distingue deux types de modules :

- Module à chargement automatique
- Module dont le chargement se fait à la demande ou Lazy Loading

i) Module à chargement automatique

Dans angular on appelle modules à chargement automatique, tous modules qui se chargent lors de la génération de l'application. Ces modules seront toujours chargés lors de l'initialisation d'une application angular ,qu'ils soient nécessaires ou pas.

Les étapes de Génération d'un **module à chargement automatique** :

- Etape 1 : Génération des modules
 - Commande : `ng g module <nom-module>`
 - Structuration du Module

Lors de la génération du module un fichier `nom-module.module.ts` est créé.

- Etape 2 : Génération des composants du module

`ng g component <nom-component>`

- Etape 3 : Exporter les components

Cette étape permet aux composants du module d'être accessibles dans les autres modules.

Ajouter le component dans le **tableau exports** dans le fichier `*.module.ts`

- Etape 4 : Importation du module créer dans le module de démarrage `app.module.ts`

Ajouter le **module** dans le tableau import de `app.module.ts`

ii) Module Lazy Loading

Pour les applications volumineuses avec de nombreuses routes, il est préférable d'utiliser les modules avec un chargement paresseux ou lazy.

Le Lazy module est un modèle de conception qui charge les modules selon les besoins c'est un chargement à la demande . Le chargement paresseux permet de réduire la taille initiale des lots, ce qui à son tour contribue à réduire les temps de chargement.

Les étapes de Génération d'un **module Lazy Loading** :

- **Etape 1 : Génération des modules**

- **Commande**

- `ng g module <nom-module> --route <nom-route> --module app.module`

- **Structuration du Module**

Lors de la génération du module les fichiers suivants sont créés :

- **nom-module.module.ts**
- **nom-routing.module.ts**
- **nom-component.ts**

- **Etape 2 : Génération les composants du module**

- `ng g component <nom-component>`

- **Etape 3 : Configuration de la Navigation**

NB: Lors de la génération d'un module lazy, angular ajoute dans le fichier **app.module.ts** la route de chargement de ce module.

C) Définition des modules du Projet

- **Module à chargement automatique**
 - **layout**
- **Module Lazy Loading**
 - **cmdes**
 - **products**
 - **securite**

d) Génération des modules du projet

i) Génération des modules à chargement automatique

- Etape 1 : Création du dossier **template** dans app
- Etape 2 : Génération
 - a) layout

ng g m layout

ii) Génération des modules Lazy Loading

- Etape 1 : création du dossier **front** dans app
- Etape 2 : Génération

a) products

ng g module products --route products --module app.module

b) commandes

ng g module commandes --route commandes --module app.module

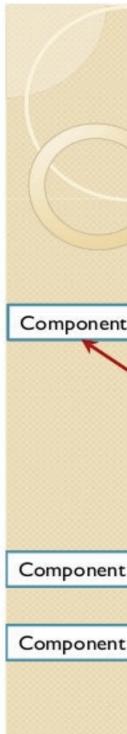
#Module Authentification

- Etape 1 : création du dossier **shared**/dans app
- Etape 2 : Génération
 - Securite

ng g module securite --route securite --module app.module

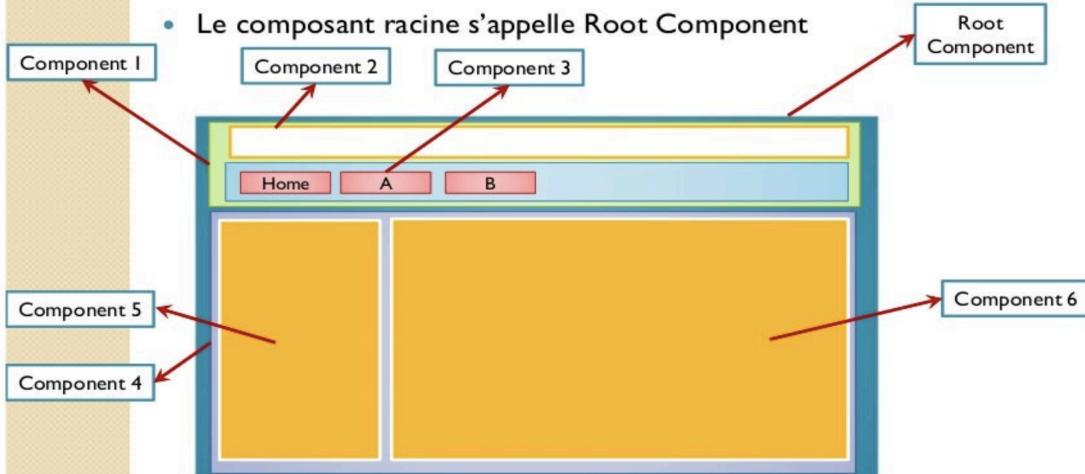
Notion de Composants

1.Définition



Components

- Les composants sont des éléments importants dans Angular.
- L'application est formée par un ensemble de composants.
- Chaque composant peut imbriquer d'autres composants définissant ainsi une structure hiérarchique.
- Le composant racine s'appelle Root Component



Components

• Chaque composant se compose principalement des éléments suivants:

- HTML Template : représentant sa vue
- Une classe représentant sa logique métier
- Une feuille de style CSS
- Un fichier spec sont des tests unitaires Ils sont exécutés à l'aide du framework de test javascript Jasmine via le programme de tâches Karma lorsque vous utilisez la commande `ng test`.

• Les composants sont facile à mettre à jour et à échanger entre les différentes parties des applications.

app.component.ts

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

app.component.html

```
<div style="text-align:center">
<h1>
  Welcome to {{title}}!!
</h1>
</div>
```

app.component.css

med@youssfi.net

Création de nouveaux composants

- Pour créer facilement des composants Angular, on peut utiliser à nouveau la commande `ng` comme suit :
 - `ng generate component NomComposant`
- Dans notre exemple, nous allons créer deux composants : `about` et `contacts`

The screenshot shows a development environment with two terminals and a file tree.

Terminal 1: Running `ng generate component about`. The output shows files being created in `src/app/about`:

```
+ C:\WS\ANGULAR2\FirstApp>ng generate component about
✖ installing component
  create src\app\about\about.component.css
  create src\app\about\about.component.html
  create src\app\about\about.component.spec.ts
  create src\app\about\about.component.ts
  update src\app\app.module.ts
```

Terminal 2: Running `ng generate component contacts`. The output shows files being created in `src/app/contacts`:

```
+ C:\WS\ANGULAR2\FirstApp>ng generate component contacts
✖ installing component
  create src\app\contacts\contacts.component.css
  create src\app\contacts\contacts.component.html
  create src\app\contacts\contacts.component.spec.ts
  create src\app\contacts\contacts.component.ts
  update src\app\app.module.ts
```

File Tree:

```
src
  app
    about
      about.component.css
      about.component.html
      about.component.spec.ts
      about.component.ts
    contacts
      contacts.component.css
      contacts.component.html
      contacts.component.spec.ts
      contacts.component.ts
    app.component.css
    app.component.html
    app.component.spec.ts
    app.component.ts
```

Structure du composant `about`

The screenshot shows the file structure of the `about` component and a browser preview.

File Structure:

```
src
  app
    about
      about.component.css
      about.component.html
      about.component.spec.ts
      about.component.ts
    contacts
      contacts.component.css
      contacts.component.html
      contacts.component.spec.ts
      contacts.component.ts
    app.component.css
    app.component.html
    app.component.spec.ts
    app.component.ts
```

about.component.ts:

```
import { Component, OnInit } from '@angular/core';
@Component({
  selector: 'app-about',
  templateUrl: './about.component.html',
  styleUrls: ['./about.component.css']
})
export class AboutComponent implements OnInit {
  constructor() { }
  ngOnInit() {
  }
}
```

about.component.html:

```
<p>
  about works!
</p>
```

about.component.css:

Browser Preview: A screenshot of a browser window showing the text "Welcome to app!!".

@Component

- Un composant est une classe qui possède le décorateur @Component
- Ce décorateur possède les propriétés suivantes :
 - **selector** : indique la déclaration qui permet d'insérer le composant dans le document HTML. Cette déclaration peut être :
 - Le nom de la balise associé à ce composant
 - selector :app-about
 - Dans ce cas le composant sera inséré par : <app-about></app-about>
 - Le nom de l'attribut associé à ce composant :
 - selector :[app-about]
 - Dans ce cas le composant sera inséré par : <div app-about></div>
 - Le nom de la classe associé à ce composant :
 - selector :.app-about
 - Dans ce cas le composant sera inséré par : <div class="app-about"></div>
 - **template ou templateUrl** :
 - **template** : permet de définir dans à l'intérieur du décorateur le code HTML représentant la vue du composant
 - **templateUrl** : permet d'associer un fichier externe HTML contenant la structure de la vue du composant
 - **styleUrls** : spécifier les feuilles de styles CSS associées à ce composant

Déclaration du composant

- Pour utiliser un composant, ce dernier doit être déclaré dans le module :

App.module.ts

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';

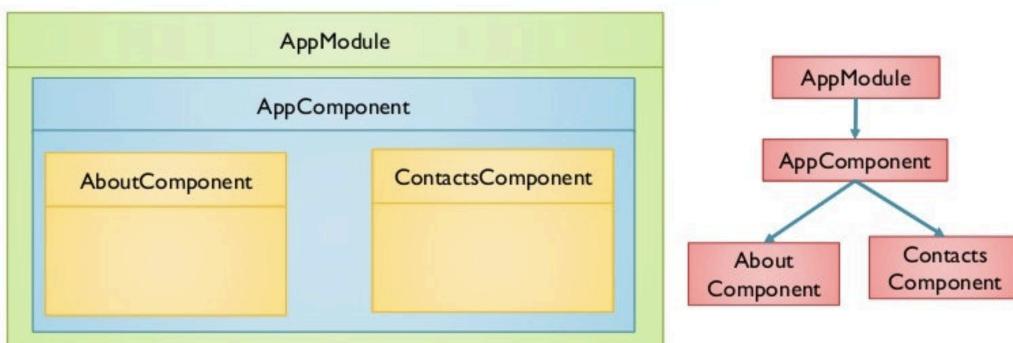
import { AppComponent } from './app.component';
import { AboutComponent } from './about/about.component';
import { ContactsComponent } from './contacts/contacts.component';

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

Utilisation composant

- Un composant peut être inséré dans n'importe que partie HTML de l'application en utilisant son selecteur associé.
- Dans cet exemple les deux composants générés sont insérés à l'intérieur du composant racine AppComponent

```
<div style="text-align:center">
  <h1>
    Welcome to {{title}}!!
  </h1>
  <app-about></app-about>
  <div app-contacts></div>
</div>
```



2.Définition des composants du Projet

a. Composants du module layout

- **nav** : Menu du template de base de la partie front de l'application
- **header** : Header du template de base de la partie front de l'application
- **footer** : Footer du template de base de la partie front de l'application

b. Module commandes

- **commandes** : Affichage des commandes d'un client
- **cart** : Affichage du panier de commande

c. Module products

- **catalogue** : Catalogue de produit
- **produit-detail** : Détail d'un produit

d. Module Securite

- **login** : Permet de se connecter
- **register** : Incription

e. Module App

- **page-not-found** : Permet de se connecter

3.Génération des composants du projet

a. Module base-template-front

- `ng g c nav`
- `ng g c header`
- `ng g c footer`

b. Module cmdes

- `ng g c commandes`
- `ng g c cart`

c. Module products

- `catalogue` : Catalogue de produit
- `produit-detail` : Détail d'un produit

d. Module Securite

- `login`: Permet de se connecter
- `register`: Incription

Chargement des Modules

1. Modules Chargement Automatique

- **Etape 1 : Importation du module `Layout` dans le module de démarrage `app.module.ts`**

Ajouter le module dans le tableau import de `app.module.ts`

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

- **Étape 2 : Exporter les composants du module**

Cette étape permet aux composants du module d'être accessibles dans les autres modules. Pour cela Ajouter les components dans le tableau exports dans le fichier `layout.module.ts`

```
@NgModule({
  declarations: [NavComponent, HeaderComponent, FooterComponent],
  imports: [CommonModule],
  exports: [NavComponent, HeaderComponent, FooterComponent],
})
export class LayoutModule {}
```

- **Étape 3 : Chargement des composants de du module Layout**

Les composants du module Layout seront chargés au chargement du composant de Démarrage (`app.component.ts`) .

```
<app-nav></app-nav>
<app-header></app-header>
<router-outlet></router-outlet>
<app-footer></app-footer>
```

NB :

Les composants du module layout doivent être chargés automatiquement en même temps que les autres composants donc on le charge dans `app.component.html` du module `app.module.ts`.

- `app.module.ts` : représente le module chargé par défaut par une application angular.
- `app.component.ts` : représente le composant qui est chargé par défaut par le module `app.module`

2. Modules Chargement Automatique

Pour le chargement des modules lazy, on utilisera le navigabilité ou le Routing .

Le Routing se fera à deux niveaux :

a. Dans app-routing.module.ts :

on définit le chargement du module en fonction d'un path dans le tableau de route.

```
const routes: Routes = [
  {path: 'products', loadChildren: () =>
    import('./front/products/products.module').then(m => m.ProductsModule) },
  { path: 'commandes', loadChildren: () =>
    import('./front/commandes/commandes.module').then(m => m.CommandesModule) },
  { path: 'securite', loadChildren: () =>
    import('./shared/securite/securite.module').then(m => m.SecuriteModule) },
  { path: 'erreur', loadChildren: () =>
    import('./shared/erreur/erreur.module').then(m => m.ErreurModule) },
];
```

NB: Le fichier **app.routing.module.ts** contient les chemins permettant de naviguer vers les modules à chargement Lazy .
Un objet route est une interface ayant les attributs suivants:

- **title?: string | Type<Resolve<string>> | ResolveFn<string>**
- **path?: string**
- **pathMatch?: 'prefix' | 'full'**
- **matcher?: UrlMatcher**
- **component?: Type<any>**
- **data?: Data**
- **resolve?: ResolveData**
- **children?: Routes**
- **loadChildren?: LoadChildren**

b. Dans le fichier routing des Modules Lazy

Ces fichiers de routing contiennent les chemins menant vers les composants de ce Modules.

Exemple : Le Routing du module Products =>

products-routing.module.ts

```
const routes: Routes = [
  { path: 'catalogue', component: CatalogueComponent },
  { path: 'detail/:id', component: ProduitDetailComponent },
];
```

NB : Le Routing du module permet de définir quel est le composant on devra afficher en fonction du **path** mais cela n'indique pas à Angular où injecter le composant dans la page.

Pour indiquer l'emplacement d'insertion du composant, il faut utiliser la directive **<router-outlet>** directement dans le "root component" .

```
<app-nav></app-nav>
<app-header></app-header>
<router-outlet></router-outlet>
<app-footer></app-footer>
```

Le **<router-outlet>** sera positionné au niveau du composant de démarrage du module principal.

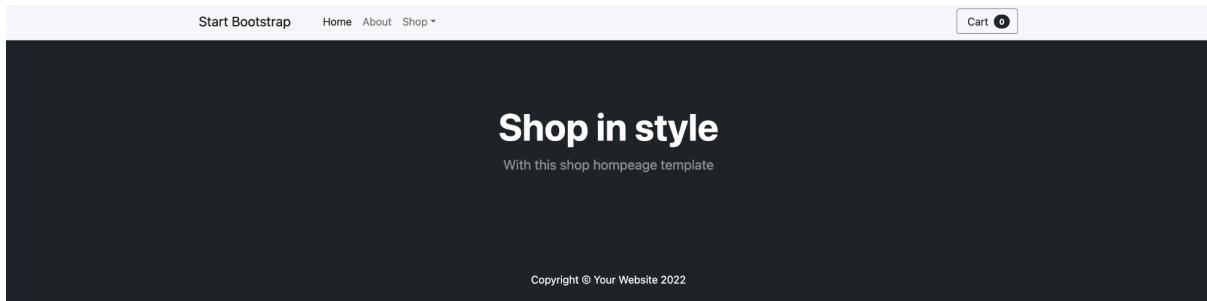
Intégration des Vues des composants Page

Téléchargement du Template

source : <https://github.com/startbootstrap/startbootstrap-shop-homepage>
<https://github.com/startbootstrap/startbootstrap-shop-item>

I. Intégration des composants du Module Layout

- A. Composant Nav
- B. Composant Header
- C. Composant Footer



II. Intégration des composants du Module Produit

- A. Routing du Module Produit

app.route.module.ts

```
const routes: Routes = [{  
  path: 'products', loadChildren: () =>  
    import('./front/products/products.module').then(m => m.ProductsModule) },  
];
```

- B. Catalogue

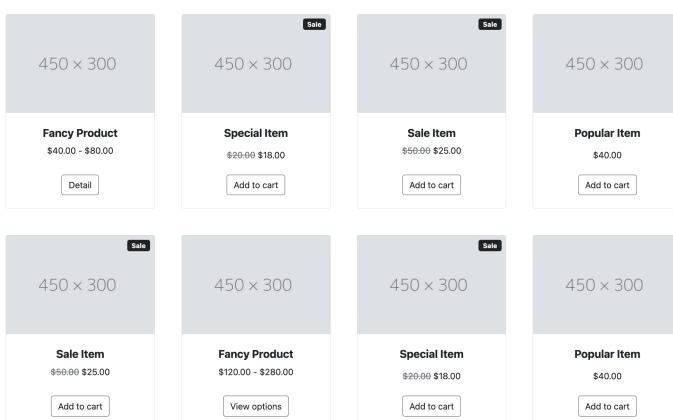
1. Routing

product.routing.module.ts

```
{ path: 'catalogue', component: CatalogueComponent },
```

2. Vue

Url : <http://localhost:4200/products/catalogue>



C. Détail un Produit

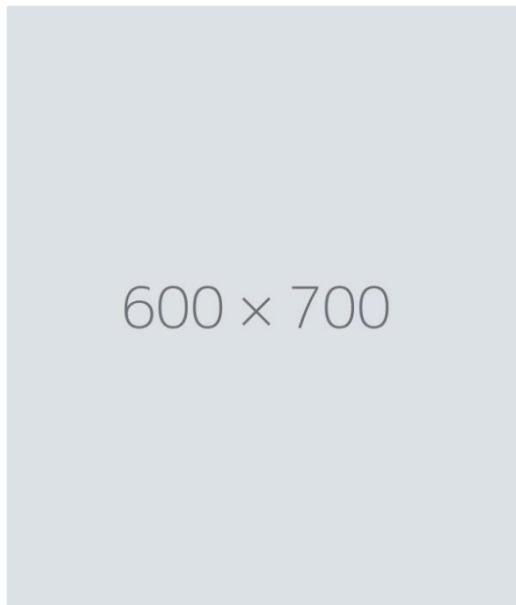
1. Routing

product.routing.module.ts

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

2. Vue

Url : <http://localhost:4200/products/detail/1>



SKU: BST-498

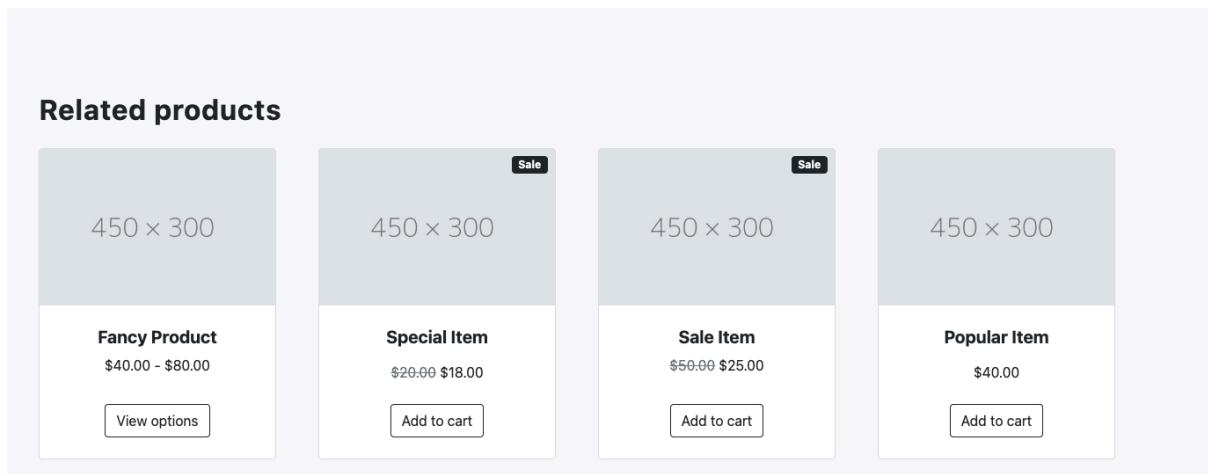
Shop item template

\$45.00 \$40.00

Lorem ipsum dolor sit amet consectetur adipisicing elit. Praesentium at dolorem quidem modi. Nam sequi consequatur obcaecati excepturi alias magni, accusamus eius blanditiis delectus ipsam minima ea iste laborum vero?

1

Add to cart



III. Intégration des composants du Module Commande

A. Routing du Module Commande

app.route.module.ts

```
{ path: 'commandes', loadChildren: () =>
import('./front/commandes/commandes.module').then(m => m.CommandesModule) },
```

B. Panier

1. Routing

commande.route.module.ts

```
{ path: 'panier', component: CartComponent },
```

2. Vue

<http://localhost:4200/commandes/panier>

Continue shopping

Shopping cart
You have 4 items in your cart

Sort by: [price](#)

	Iphone 11 pro 256GB, Navy Blue	2	\$900
	Samsung galaxy Note 10 256GB, Navy Blue	2	\$900
	Canon EOS M50 Onyx Black	1	\$1199
	MacBook Pro 1TB, Graphite	1	\$1799

Card details

Subtotal	\$4798.00
Shipping	\$20.00
Total(Incl. taxes)	\$4818.00

[\\$4818.00Checkout](#)

C. Commande

1. Routing

```
{ path: 'client/:id', component: CommandesComponent }
```

2. Vue

<http://localhost:4200/commandes/client/1>

Commande du Client

Shopping cart
You have 4 items in your cart

Sort by: [price](#)

	Iphone 11 pro 256GB, Navy Blue	2	\$900
	Samsung galaxy Note 10 256GB, Navy Blue	2	\$900
	Canon EOS M50 Onyx Black	1	\$1199
	MacBook Pro 1TB, Graphite	1	\$1799

Details Commande

Subtotal	\$4798.00
Shipping	\$20.00
Total(Incl. taxes)	\$4818.00

[\\$4818.00Checkout](#)

D. Template

```
<section class="h-50" style="background-color: #eee">
  <div class="container py-5 h-75">
    <div class="row d-flex justify-content-center align-items-center h-100">
      <div class="col">
        <div class="card">
          <div class="card-body p-4">
            <div class="row">
              <div class="col-lg-7">
                <h5 class="mb-3">
                  <a href="#" class="text-body">
                    ><i class="fas fa-long-arrow-alt-left me-2"></i>Commande du
                    Client
                  </a>
                </h5>
                <hr />
                <div
                  class="d-flex justify-content-between align-items-center mb-4"
                >
                  <div>
                    <p class="mb-1">Shopping cart</p>
                    <p class="mb-0">You have 4 items in your cart</p>
                  </div>
                  <div>
                    <p class="mb-0">
                      <span class="text-muted">Sort by:</span>
                      <a href="#" class="text-body">
                        >price <i class="fas fa-angle-down mt-1"></i>
                      </a>
                    </p>
                  </div>
                </div>
                <div class="card mb-3">
                  <div class="card-body">
                    <div class="d-flex justify-content-between">
                      <div class="d-flex flex-row align-items-center">
                        <div class="ms-3">
                          <h5>Iphone 11 pro</h5>
                          <p class="small mb-0">256GB, Navy Blue</p>
                        </div>
                      </div>
                      <div class="d-flex flex-row align-items-center">
                        <div style="width: 50px">
                          <h5 class="fw-normal mb-0">2</h5>
                        </div>
                      </div>
                    </div>
                  </div>
                </div>
              </div>
            </div>
          </div>
        </div>
      </div>
    </div>
  </div>
</section>
```

```
<div style="width: 80px">
    <h5 class="mb-0">$900</h5>
</div>
<a href="#" style="color: #cecece"><i class="fas fa-trash-alt"></i></a>
</div>
</div>
</div>
<div class="card mb-3">
    <div class="card-body">
        <div class="d-flex justify-content-between">
            <div class="d-flex flex-row align-items-center">
                <div class="ms-3">
                    <h5>Samsung Galaxy Note 10</h5>
                    <p class="small mb-0">256GB, Navy Blue</p>
                </div>
            </div>
            <div class="d-flex flex-row align-items-center">
                <div style="width: 50px">
                    <h5 class="fw-normal mb-0">2</h5>
                </div>
                <div style="width: 80px">
                    <h5 class="mb-0">$900</h5>
                </div>
                <a href="#" style="color: #cecece"><i class="fas fa-trash-alt"></i></a>
            </div>
        </div>
    </div>
</div>
<div class="card mb-3">
    <div class="card-body">
        <div class="d-flex justify-content-between">
            <div class="d-flex flex-row align-items-center">
                <div class="ms-3">
                    <h5>Canon EOS M50</h5>
                    <p class="small mb-0">Onyx Black</p>
                </div>
            </div>
            <div class="d-flex flex-row align-items-center">
                <div style="width: 50px">
                    <h5 class="fw-normal mb-0">1</h5>
                </div>
            </div>
        </div>
    </div>
</div>
```

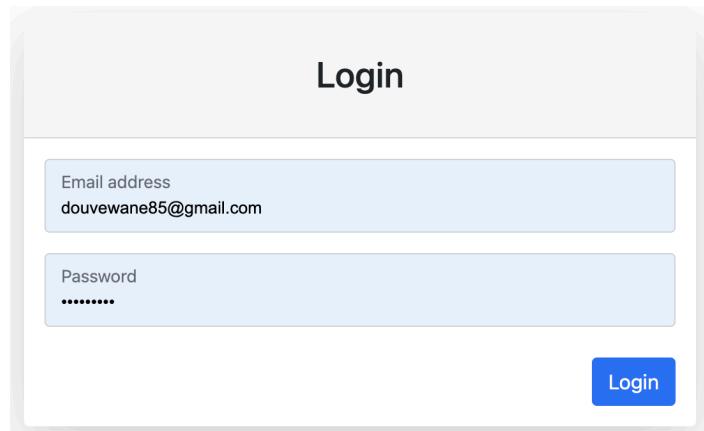
```
        </div>
        <div style="width: 80px">
            <h5 class="mb-0">$1199</h5>
        </div>
        <a href="#" style="color: #cecece">
            <i class="fas fa-trash-alt"></i>
        </a>
    </div>
</div>
<div class="card mb-3 mb-lg-0">
    <div class="card-body">
        <div class="d-flex justify-content-between">
            <div class="d-flex flex-row align-items-center">
                <div class="ms-3">
                    <h5>MacBook Pro</h5>
                    <p class="small mb-0">1TB, Graphite</p>
                </div>
            </div>
            <div class="d-flex flex-row align-items-center">
                <div style="width: 50px">
                    <h5 class="fw-normal mb-0">1</h5>
                </div>
                <div style="width: 80px">
                    <h5 class="mb-0">$1799</h5>
                </div>
                <a href="#" style="color: #cecece">
                    <i class="fas fa-trash-alt"></i>
                </a>
            </div>
        </div>
    </div>
</div>
<div class="col-lg-5">
    <div class="card bg-light text-dark rounded-3 h-100">
        <div class="card-body">
            <div
                class="d-flex justify-content-between align-items-center mb-4"
            >
                <h5 class="mb-0">Details Commande</h5>
                  
    import('./shared/securite/securite.module').then((m) => m.SecuriteModule),  
},
```

`securite.route.module.ts`

```
const routes: Routes = [  
  { path: 'login', component: LoginComponent },  
  { path: 'register', component: RegisterComponent },  
];
```

2. Composant de Connexion

Url: <http://localhost:4200/securite/login>



3. Composant d'inscription

Url: <http://localhost:4200/securite/register>

4. Template

```
<div id="layoutAuthentication">  
  <div id="layoutAuthentication_content">  
    <main>  
      <div class="container mb-5">  
        <div class="row justify-content-center">  
          <div class="col-lg-5">  
            <div class="card shadow-lg border-0 rounded-lg mt-5">  
              <div class="card-header">  
                <h3 class="text-center font-weight-light my-4">Login</h3>  
              </div>  
              <div class="card-body">
```

```
<form>

<div class="form-floating mb-3">
    <input
        class="form-control"
        id="inputEmail"
        type="email"
        placeholder="name@example.com"
    />
    <label for="inputEmail">Email address</label>
</div>
<div class="form-floating mb-3">
    <input
        class="form-control"
        id="inputPassword"
        type="password"
        placeholder="Password"
    />
    <label for="inputPassword">Password</label>
</div>

<div
    class="d-flex align-items-center justify-content-between mt-4
mb-0">
    >
        <a class="small" href="#"></a>
        <a class="btn btn-primary" href="index.html">Login</a>
    </div>
</form>
</div>
</div>
</div>
</div>
</main>
</div>
</div>
```

IV. Navigabilité

A. Route de Redirection

`app.route.module.ts`

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

NB : `PathMatch` est un attribut de l'interface route qui permet de mapper la route à l'url . Il est par défaut à `prefix`. On le met `PathMatch` à `full` lorsqu'on fait une redirection.

B. Page Inexistante

1. Crédation du Composant Page-not-found

Dans le Module App : `ng g c page-not-found`

2. Routing

`app.route.module.ts`

```
{ path: '**', component: PageNotFoundComponent },
```

C. Lien sur les vues

1. Module RouterModule

Le Module `RouterModule` fournit des directives et des méthodes pour la navigation dans l'application à partir des vues définies dans une application.

2. Ajout du Module RouterModule

```
@NgModule({
  declarations: [
    ----
  ],
  imports: [
    CommonModule,
    RouterModule
  ],
  exports: [
    -----
  ]
})
```

3. La Directive routerLink

En utilisant des liens natifs ``, le navigateur va produire une requête HTTP GET vers le serveur et recharger toute l'application.

Pour éviter ce problème, le module `RouterModule` d'Angular fournit la directive `routerLink` qui permet d'intercepter l'événement click sur les liens et de charger la vue d'un composant sans recharger toute l'application.

Exemple :

```
<a routerLink="/search">Search</a>
```

Remarque

La directive `routerLink` génère tout de même l'attribut `href` pour faciliter la compréhension de la page par les "browsers" ou "moteurs de recherche".

4. La Directive RouterLinkActive

La directive `RouterLinkActive` est appliquée parallèlement à la directive `RouterLink`. Cette directive permet d'appliquer des classes CSS lorsque la route est active.

iii) Route avec Paramètre

Au niveau des vues on peut construire des routes avec des paramètres :

```
<a routerLink="/books/123"> routeName</a>
```

ou

```
<a [routerLink]="'/books', 123">routeName</a>
```

iv) Route avec Paramètre optionnels

Il est également possible de passer des paramètres optionnels par "`query string`" via l'Input `queryParams`.

```
<a routerLink=" '/search'"  
[queryParams]="{keywords: 'eXtreme Programming'}">  
    eXtreme Programming Books  
</a>
```

Path de la Route :

`/search?keywords=eXtreme Programming`

5. Les Liens du Projet

a) Composant de Navigation

```
<a  
    class="nav-link"  
    aria-current="page"  
    routerLink="/products/catalogue"  
    routerLinkActive="active"  
>Catalogue</a>  
>
```

```
<a  
    class="nav-link"  
    routerLink="/commandes/client/1"  
    routerLinkActive="active"  
>Mes Commande</a>  
>
```

```
<a class="btn btn-outline-dark" routerLink="/commandes/panier">  
    <i class="bi-cart-fill me-1"></i>  
    Panier  
    <span class="badge bg-dark text-white ms-1 rounded-pill">0</span>  
</a>
```

```
<ul class="navbar-nav mb-2 mb-lg-0 ms-lg-4">  
    <li class="nav-item">  
        <a  
            class="nav-link"  
            routerLink="/securite/login"  
            [queryParams]={{page:'login'}}  
            routerLinkActive="active"  
>Se Connecter</a>  
    </li>  
    <li class="nav-item">  
        <a  
            class="nav-link"  
            routerLink="/securite/register"  
            routerLinkActive="active"  
>S'inscrire</a>  
    </li>  
</ul>
```

b) Composant Catalogue

```
<a  
    class="btn btn-outline-dark mt-auto"  
    [routerLink]=["/products/detail", '1']"  
    >Detail</a  
>
```

c) Valider le Panier

```
<a  
    type="button"  
    class="btn btn-info btn-block btn-lg"  
    routerLink="/securite/login"  
    [queryParams] = { page: 'panier' }"  
    >  
    <div class="d-flex justify-content-between">  
        <span>$4818.00</span>  
        <span class="ml-2"  
            >Terminer La Commande  
            <i class="fas fa-long-arrow-alt-right ms-2"></i>  
        </span>  
    </div>  
</a>
```

V. Récupération des Paramètres au niveau du Component

On peut récupérer le paramètre d'une route au niveau du composant chargé par ce path en utilisant le Service **ActivatedRoute**.

A. Paramètre de Route

Exemple :

```
#product-detail.component.ts  
export class ProductsFrontDetailsComponent implements OnInit {  
  
    //Injection de Dépendance de ActivatedRoute dans le constructeur  
  
    constructor(private route: ActivatedRoute) { }  
  
    ngOnInit(): void {  
  
        alert(this.route.snapshot.params['id'])
```

```
}
```

B. Query String

```
this.route.queryParams.subscribe(params => {  
  alert(params['page']);  
});
```

VI. Routage Dynamique

La classe **Router** offre deux méthodes pour naviguer dans les classes de composants : **Routeur.navigate** et **Router.navigateByUrl**. Les deux méthodes renvoient une **promesse** qui se résout :

- **true** si la navigation est réussie, null s'il n'y a pas de navigation,
- **false** si la navigation échoue ou est complètement rejetée s'il y a une erreur.

Pour utiliser l'une ou l'autre méthode, vous devez d'abord vous assurer que la Router classe est injectée dans la classe de composants.

#*.ts, Importation de la Classe Router

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

```
export class AppComponent {  
  
    constructor(private router: Router) {}  
  
}  
  
}
```

Maintenant, vous pouvez utiliser **Router.navigate** ou **Router.navigateByUrl**.

1. **Router.navigate**

#Méthode 1

```
nomFonction() {this.router.navigate(['/users'])}
```

#Méthode 2

```
nomFonction() {  
  
    this.router.navigate(['/users'])  
  
    .then(nav => {  
  
        console.log(nav); // true if navigation is successful  
  
    }, err => {  
  
        console.log(err) // when there's an error  
  
    })  
}
```

2. Router.navigateByUrl.

```
this.router.navigateByUrl(`/products/detail/${id}`);
```

Cours 3 :

TypeScript

I. Langage TypeScript

A) TypeScript

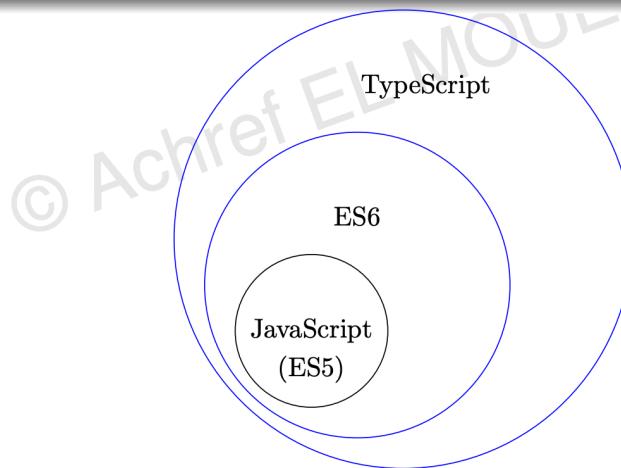
a) Présentation

TypeScript

- langage de programmation
 - procédural et orienté-objet
 - supportant le typage statique, dynamique et générique
- open-source
- créé par Anders Hejlsberg (inventeur de C#) de **MicroSoft**

TypeScript : sur-couche de ES6 ajoutant

- typage statique
- meilleure gestion de module (à ne pas confondre avec les modules **Angular**)



b) Framework utilisant TypeScript

Frameworks Frontend utilisant TypeScript

- **Angular** depuis la version 2
- **React** (voir <https://www.typescriptlang.org/docs/handbook/react.html>) et **Vue.js** (voir <https://fr.vuejs.org/v2/guide/typescript.html>) qui intègrent un support **TypeScript**

Frameworks Backend utilisant TypeScript

- **ExpressJS** avec l'extension **TypeScript Node Starter** (voir <https://github.com/microsoft/TypeScript-Node-Starter>)
- **nest** Framework backend de **TypeScript**

c) Librairie utilisant TypeScript

Librairie écrite en TypeScript

RxJS (voir <https://github.com/ReactiveX/rxjs>)

API écrite en TypeScript

Firebase (voir <https://github.com/firebase/>)

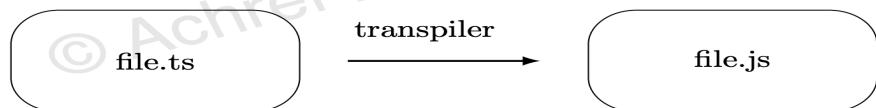
Deno

Le nouveau projet de Ryan Dahl pour remplacer **NodeJS** écrit en **TypeScript**

d) Transpilation

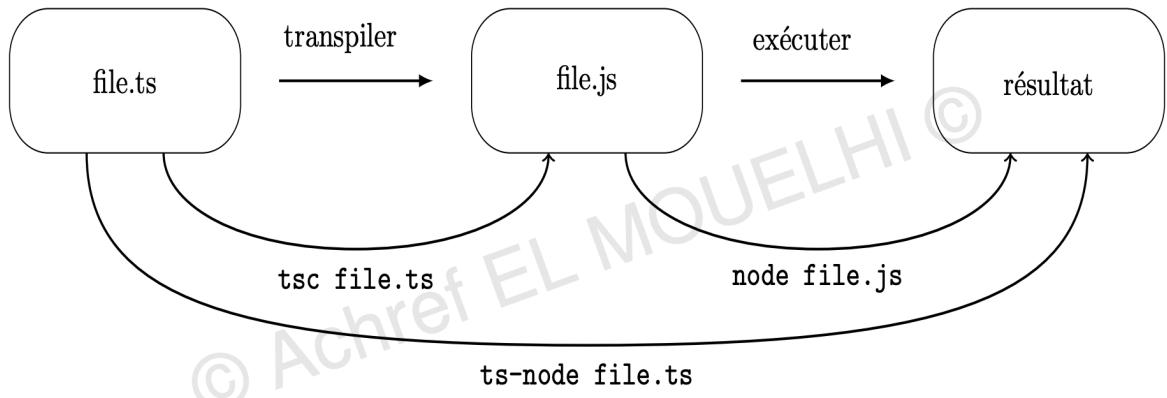
Le navigateur ne comprend pas TypeScript

Il faut le transcompiler (ou transpiler) en **JavaScript**



Ok

Comment va t-on procéder dans ce cours ?



Pour consulter la liste d'options pour la commande `tsc`

e) Outils pour utiliser exécuter ` TypeScript

De quoi on a besoin ?

- **Node.js** pour exécuter la commande `node`
- **TypeScript** pour exécuter la commande `tsc`
- **ts-node** pour exécuter la commande `ts-node`

Pour Node.js, il faut

- aller sur <https://nodejs.org/en/>
- choisir la dernière version, télécharger et installer

Les règles de nommage

- Pour les classes et interfaces : **Pascal case**
- Pour les variables, fonctions et méthodes : **Camel case**
- Pour les fichiers : **Kebab case**

Pour TypeScript, il faut

- ouvrir une console (invite de commandes)
- lancer la commande `npm install -g typescript`
- vérifier la version avec la commande `tsc -v`

Pour plus de détails

<https://wprock.fr/blog/conventions-nommage-programmation/>

Pour ts-node, il faut

- lancer la commande `npm install -g ts-node`
- vérifier la version avec la commande `ts-node -v`

B) Les Variables en TypeScript

Trois modes de typage en **TypeScript**

- Statique : le développeur précise le type à la déclaration de la variable
- Dynamique : le type d'une variable est déterminé à l'exécution (comme en JS)
- Générique : il permet au développeur de paramétriser un type complexe

Quels types pour les variables en **TypeScript** ?

- `number` pour les nombres (entiers, réels, binaires, décimaux, hexadécimaux...)
- `string` pour les chaînes de caractère
- `boolean` pour les booléens
- `symbol` pour les symboles [**ES6**]
- `array` pour les tableaux non-statiques (taille variable)
- `tuple` pour les tableaux statiques (taille et type fixes)
- `object` pour les objets
- `any` pour les variables pouvant changer de type dans le programme
- `enum` pour les énumérations (**tableau de constantes**)

C) Variables

a) Déclaration

Déclarer une variable

```
var nomVariable: typeVariable;
```

Exemple

```
var x: number;
```

Initialiser une variable

```
x = 2;
```

Déclarer et initialiser une variable

```
var x: number = 2;
```

b) String

Pour les chaînes de caractères, on peut faire

```
var str1: string = "wick";
var str2: string = 'john';
```

On peut aussi utiliser template strings

```
var str3: string = `Bonjour ${str2} ${str1}
Que pensez-vous de TypeScript ?
`;
console.log(str3);
// affiche Bonjour john wick
Que pensez-vous de TypeScript ?
```

L'équivalent de faire

```
var str3: string = "Bonjour " + str2 + " " + str1 + "\nQue
pensez-vous de TypeScript ?";
```

c) Tableau

Une première déclaration de tableau

```
var list: number[] = [1, 2, 3];
console.log(list);
// affiche [ 1, 2, 3 ]
```

Une deuxième déclaration

```
var list: Array<number> = new Array(1, 2, 3);
console.log(list);
// affiche [ 1, 2, 3 ]
```

Ou encore plus simple

```
var list: Array<number> = [1, 2, 3];
console.log(list);
// affiche [ 1, 2, 3 ]
```

d) Tuple

Pour les tuples, on initialise toutes les valeurs à la déclaration

```
var t: [number, string, string] = [ 100, "wick", 'john' ];
```

Pour accéder à un élément d'un tuple en lecture ou en écriture

```
console.log(t[0]);
// affiche 100

t[2] = "travolta";
console.log(t);
// affiche [ 100, 'wick', 'travolta' ]
```

Cependant, ceci génère une erreur

```
t = [100, 200, 'john'];
```

Avec TypeScript 3.0, on peut rendre certains éléments de tuple optionnels

```
var t: [number, string?, string?] = [100];
console.log(t);
// affiche [ 100 ]
console.log(t[1]);
// affiche undefined
```

Pour ajouter un élément

```
t[1] = 'wick';
```

Ceci génère une erreur

```
t[2] = 100;
```

Et cette instruction aussi car on dépasse la taille du tuple

```
t[3] = 100;
```



e) Any

Exemple avec any

chrome://bookmarks

```
var x: any;
x = "bonjour";
x = 5;
console.log(x);
// affiche 5;
```

Une variable de type any peut être affectée à n'importe quel autre type de variable

```
var x: any;
x = "bonjour";
x = 5;
var y: number = x;
```



f) unknown

Le type `unknown` (TypeScript 3.0) fonctionne comme `any` mais ne peut être affecté qu'à une variable de type `unknown` ou `any`

```
var x: unknown;
x = "bonjour";
x = 5;
console.log(x);
// affiche 5;
```

Ceci génère donc une erreur

```
var x: unknown;
x = "bonjour";
x = 5;
var y: number = x;
```

g) Enumeration

Déclarons une énumération (dans `file.ts`)

```
enum mois { JANVIER, FEVRIER, MARS, AVRIL, MAI, JUIN, JUILLET,
AOUT, SEPTEMBRE, OCTOBRE, NOVEMBRE, DECEMBRE };
```

L'indice du premier élément est 0

```
console.log(mois.AVRIL)
// affiche 3
```

Pour modifier l'indice du premier élément

```
enum mois { JANVIER = 1, FEVRIER, MARS, AVRIL, MAI, JUIN,
JUILLET, AOUT, SEPTEMBRE, OCTOBRE, NOVEMBRE, DECEMBRE };
```

En affichant maintenant, le résultat est

```
console.log(mois.AVRIL)
// affiche 4
```

h) Déclaration d'un Objet

Pour déclarer un objet

```
var obj: {  
    nom: string;  
    numero: number;  
};
```

On peut initialiser les attributs de cet objet

```
obj = {  
    nom: 'wick',  
    numero: 100  
};  
  
console.log(obj);  
// affiche { nom: 'wick', numero: 100 }  
  
console.log(typeof obj);  
// affiche object
```

On peut modifier les valeurs d'un objet ainsi

```
obj.nom = 'abruzzi';  
obj['numero'] = 200;  
  
console.log(obj);  
// affiche { nom: 'abruzzi', numero: 200 }
```

Ceci est une erreur

```
obj.nom = 125;
```

i) Union des Types

Union de type

Il est possible d'autoriser plusieurs types de valeurs pour une variable

WariCustomAuthenticatorAuthenticator.php
todo: check the credentials inside F:\wamp64\...

Déclarer une variable acceptant plusieurs types de valeur

```
var y: number | boolean | string;
```

affecter des valeurs de type différent

```
y = 2;  
y = "bonjour";  
y = false;
```

Ceci génère une erreur

```
y = [2, 5];
```

j) Croisement de Type

Croisement de type

Il est possible qu'une variable ait les propriétés de plusieurs types différents

Déclarer une variable ayant les propriétés de plusieurs types différents

```
var enseignant: {  
    nom: string;  
    salaire: number;  
};  
  
var etudiant: {  
    niveau: string;  
};  
  
var doctorant: typeof etudiant & typeof enseignant = {  
    nom: 'wick',  
    salaire: 1700,  
    niveau: 'master'  
};  
console.log(doctorant);  
// affiche { nom: 'wick', salaire: 1700, niveau: 'master' }
```

k) Opérateur ??

L'opérateur ?? permet d'éviter d'affecter la valeur null ou undefined à une variable

```
var obj = { nom: null, prenom: 'john' };
let nom: string = obj.nom ?? 'doe';
console.log(nom);
// affiche doe
```

C'est équivalent à

```
var obj = { nom: null, prenom: 'john' };
let nom: string = (obj.nom !== null && obj.nom !==
undefined) ? obj.nom : 'doe';
console.log(nom);
// affiche doe
```

I) Opérateur de destruction

Déstructuration (ES6)

permet d'extraire les données d'un objet ou un tableau dans des variables.

Exemple

```
var personne = { nom: 'wick', prenom: 'john' };
var { nom, prenom } = personne;

console.log(nom, prenom);
// affiche wick john
```

- **Constante**

D) Fonctions

a) Déclaration et appel

Déclarer une fonction

```
function nomFonction([les paramètres]){
    les instructions de la fonction
}
```

Exemple

```
function somme(a: number, b: number): number {
    return a + b;
}
```

Appeler une fonction

```
let resultat: number = somme (1, 3);
console.log(resultat);
// affiche 4
```

Une fonction qui ne retourne rien a le type **void**

```
function direBonjour(): void {
    console.log("bonjour");
}
```

Une fonction qui n'atteint jamais sa fin a le type **never**

```
function boucleInfinie(): never {
    while (true) {

    }
}
```

b) Fonction avec argument par défaut

Il est possible d'attribuer une valeur par défaut aux paramètres d'une fonction

```
function division(x: number, y: number = 1) : number
{
    return x / y;
}

console.log(division(10));
// affiche 10

console.log(division(10, 2));
// affiche 5
```

c) Fonction avec argument optionnels

Il est possible de rendre certains paramètres d'une fonction optionnels

```
function division(x: number, y?: number): number {
    if(y)
        return x / y;
    return x;
}

console.log(division(10));
// affiche 10

console.log(division(10, 2));
// affiche 5
```

d) Fonction avec un nombre argument indéfini

Il est possible de définir une fonction prenant un nombre indéfini de paramètres

```
function somme(x: number, ...tab: number[]): number {
    for (let elt of tab)
        x += elt;
    return x;
}

console.log(somme(10));
// affiche 10

console.log(somme(10, 5));
// affiche 15

console.log(somme(10, 1, 6));
// affiche 17
```

e) Paramètres à plusieurs types autorisés

Il est possible d'autoriser plusieurs types pour un paramètre

```
function stringOrNumber(param1: string | number,
    param2: number): number {
    if (typeof param1 == "string")
        return param1.length + param2;
    return param1 + param2;
}

console.log(stringOrNumber("bonjour", 3));
// affiche 10

console.log(stringOrNumber(5, 3));
// affiche 8
```

E) Fonctions fléchées

Fonctions fléchées : pourquoi ?

- Simplicité d'écriture du code ⇒ meilleure lisibilité
- Pas de binding avec les objets prédéfinis : arguments, this...
- ...

Bookmarks
chrome://bookmarks

Il est possible de déclarer une fonction en utilisant les expressions fléchées

```
let nomFonction = ([les paramètres]): typeValeurRetour => {
    les instructions de la fonction
}
```

Exemple

```
let somme = (a: number, b: number): number => { return a + b; }
```

Ou en plus simple

```
let somme = (a: number, b: number): number => a + b;
```

Appeler une fonction fléchée

```
let resultat: number = somme (1, 3);
```

Cas d'une fonction fléchée à un seul paramètre

```
let carre = (a: number): number => a * a;
console.log(carre(2)); // affiche 4
```

Sans typage, la fonction peut être écrite ainsi

```
let carre = a => a * a;
console.log(carre(2)); // affiche 4
```

Déclaration d'une fonction fléchée sans paramètre

```
let sayHello = (): void => console.log('Hello');
sayHello(); // affiche Hello
```

Remarque

- Il est déconseillé d'utiliser les fonctions fléchées dans un objet
- Le mot-clé `this` est inutilisable dans les fonctions fléchées

Sans les fonctions fléchées

```
let obj = {
  nom: 'wick',
  afficherNom: function() {
    console.log(this.nom)
  }
}
obj.afficherNom();
// affiche wick
```

Avec les fonctions fléchées

```
let obj = {
  nom: 'wick',
  afficherNom: () => {
    console.log(this.nom)
  }
}
obj.afficherNom();
// affiche undefined
```

Les fonctions fléchées sont utilisées pour réaliser les opérations suivant sur les tableaux

- `forEach()` : pour parcourir un tableau
- `map()` : pour appliquer une fonction sur les éléments d'un tableau
- `filter()` : pour filtrer les éléments d'un tableau selon un critère défini sous forme d'une fonction anonyme ou fléchée
- `reduce()` : pour réduire tous les éléments d'un tableau en un seul selon une règle définie dans une fonction anonyme ou fléchée
- `some()` : pour vérifier s'il existe au moins un élément qui respect une condition
- `every()` : pour vérifier si tous les éléments respectent une condition
- ...

Utiliser `forEach` pour afficher le contenu d'un tableau

```
var tab = [2, 3, 5];
tab.forEach_elt => console.log_elt);
// affiche 2 3 5
```

Dans `forEach`, on peut aussi appeler une fonction `afficher`

```
tab.forEach_elt => afficher_elt);
function afficher(value: number) {
    console.log(value);
}
// affiche 2 3 5
```

On peut simplifier l'écriture précédente en utilisant les callback

```
tab.forEach(afficher);
function afficher(value: number) {
    console.log(value);
}
// affiche 2 3 5
```

On peut aussi utiliser `filter` pour filtrer des éléments

```
tab.map_elt => elt + 3)
    .filter_elt => elt > 5)
    .forEach_elt => console.log_elt);
// affiche 6 8
```

Remarque

Attention, selon l'ordre d'appel de ces méthodes, le résultat peut changer.

Exemple avec `reduce` : permet de réduire les éléments d'un tableau en une seule valeur

```
var tab = [2, 3, 5];
var somme = tab.map_elt => elt + 3)
    .filter_elt => elt > 5)
    .reduce((sum, elt) => sum + elt);

console.log(somme);
// affiche 14
```

Si on a plusieurs instructions, on doit ajouter les accolades

```
var tab = [2, 3, 5];
var somme = tab.map_elt => elt + 3)
    .filter_elt => elt > 5)
    .reduce((sum, elt) => {
        return sum + elt;
    })
    console.log(somme);
// affiche 14
```

Remarques

- Le premier paramètre de `reduce` correspond au résultat de l'itération précédente
- Le deuxième correspond à l'élément du tableau de l'itération courante
- Le premier paramètre est initialisé par la valeur du premier élément du tableau
- On peut changer la valeur initiale du premier paramètre en l'ajoutant à la fin de la méthode

Dans cet exemple, on initialise le premier paramètre de `reduce` par la valeur 0

```
var somme = tab.map(elt => elt + 3)
    .filter(elt => elt > 5)
    .reduce((sum, elt) => sum + elt, 0);

console.log(somme);
// affiche 14
```

Dans cet exemple, on vérifie s'il existe un élément pair dans le tableau, après modification

```
var tab = [2, 3, 5];

let result = tab.map(elt => elt + 3)
    .some(elt => elt % 2 == 0);

console.log(result);
// affiche true
```

Dans cet exemple, on vérifie si tous les éléments du tableau sont pairs, après modification

```
var tab = [2, 3, 5];

let result = tab.map(elt => elt + 3)
    .every(elt => elt % 2 == 0);

console.log(result);
// affiche false
```

Remarque

Attention, selon l'ordre d'appel de ces méthodes, le résultat peut changer.

Exemple avec `reduce` : permet de réduire les éléments d'un tableau en une seule valeur

```
var tab = [2, 3, 5];
var somme = tab.map(elt => elt + 3)
    .filter(elt => elt > 5)
    .reduce((sum, elt) => sum + elt);

console.log(somme);
// affiche 14
```

TypeScript

Considérons les deux objets suivants

```
let obj = { nom: 'wick', prenom: 'john' };
let obj2 = obj;
```

Modifier l'un ⇒ modifier l'autre

```
obj2.nom = 'abruZZI';
console.log(obj);
// affiche { nom: 'abruZZI', prenom: 'john' }

console.log(obj2);
// affiche { nom: 'abruZZI', prenom: 'john' }
```

F) Notion de Module

Module

- Introduit dans ES6
- Un fichier pouvant contenir des variables ; fonctions, classes, interfaces...

Propriétés

- Il est possible d'utiliser des éléments définis dans un autre fichier : une variable, une fonction, une classe, une interface...
- Pour cela, il faut l'importer là où on a besoin de l'utiliser
- Pour importer un élément, il faut l'exporter dans le fichier source
- En transpilant le fichier contenant les `import`, les fichiers contenant les éléments importés seront aussi transpilés.

Pour exporter les deux fonctions somme et produit de fonction.ts

```
export function somme(a: number = 0, b: number = 0) {
    return a + b;
}

export function produit(a: number = 0, b: number = 1) {
    return a * b;
}
```

Ou aussi

```
function somme(a:number = 0, b:number = 0) {
    return a + b;
}

function produit(a: number = 0, b: number = 1) {
    return a * b;
}
export { somme, produit };
```

Pour importer et utiliser une fonction

```
import { somme } from './fonctions';

console.log(somme(2, 5));
// affiche 7
```

Pour importer plusieurs éléments

```
import { somme, produit } from './fonctions';

console.log(somme(2, 5));
// affiche 7

console.log(produit(2, 5));
// affiche 10
```

On peut aussi utiliser des alias

```
import { somme as s, produit as p } from './fonctions';

console.log(s(2, 5));
// affiche 7

console.log(p(2, 5));
// affiche 10
```

Ou aussi

```
import * as f from './fonctions';

console.log(f.somme(2, 5));
// affiche 7

console.log(f.produit(2, 5));
// affiche 10
```

Les alias peuvent être attribués à l'exportation

```
function somme(a:number = 0, b:number = 0) {
    return a + b;
}

function produit(a: number = 0, b: number = 1) {
    return a * b;
}
export { produit as p, somme as s } ;
```

Pour importer

```
import * as f from './fonctions';

console.log(f.s(2, 5));
// affiche 7

console.log(f.p(2, 5));
// affiche 10
```

On peut aussi utiliser le export default (un seul par fichier)

```
export default function somme(a: number = 0, b: number = 0) {
    return a + b;
}

export function produit(a: number = 0, b: number = 1) {
    return a * b;
}
```

Pour importer, pas besoin de { } pour les éléments exporter par défaut

```
import somme from './fonctions';
import { produit } from './fonctions';

console.log(somme(2, 5));
// affiche 7

console.log(produit(2, 5));
// affiche 10
```

G) La POO avec TypeScript

a) Présentation

Qu'est ce qu'une classe en POO ?

- Ça correspond à un plan, un moule, une usine...
- C'est une description abstraite d'un type d'objets
- Elle représente un ensemble d'objets ayant les mêmes propriétés statiques (attributs) et dynamiques (méthodes)

Instance ?

- Une instance correspond à un objet créé à partir d'une classe (via le constructeur)
- L'instanciation : création d'un objet d'une classe
- instance ≡ objet

b) Définition d'une classe

Considérons la classe Personne définie dans personne.ts

```
export class Personne {  
    num: number;  
    nom: string;  
    prenom: string;  
}
```

Avant de compiler, vérifiez dans tsconfig.json les propriétés suivantes

- "target": "es6"
- "strictPropertyInitialization": false

En TypeScript

- Toute classe a un constructeur par défaut sans paramètre.
- Par défaut, la visibilité des attributs est public.

Hypothèse

Si on voulait créer un objet de la classe Personne avec les valeurs 1, wick et john

Étape 1 : Commençons par importer la classe Personne dans file.ts

```
import { Personne } from './personne';
```

Étape 2 : déclarons un objet (objet non créé)

```
let personne: Personne;
```

Étape 3 : créons l'objet (instanciation) de type Personne (objet créé)

```
personne = new Personne();
```

On peut faire déclaration + instanciation

```
let personne: Personne = new Personne();
```

Affectons les valeurs aux différents attributs

```
personne.num = 1;  
personne.nom = "wick";  
personne.prenom = "john";
```

Pour être sûr que les valeurs ont bien été affectées aux attributs, on affiche

```
console.log(personne);  
// affiche Personne { num: 1, nom: 'wick', prenom: 'john' }
```

#Les Setters

Hypothèse

Bookmarks
chrome://bookmarks

Supposant que l'on n'accepte pas de valeur négative pour l'attribut `num` de la classe `Personne`

Démarche

- ① Bloquer l'accès direct aux attributs (mettre la visibilité à `private`)
- ② Définir des méthodes publiques qui contrôlent l'affectation de valeurs aux attributs (les `setter`)

Convention

- Mettre la visibilité `private` ou `protected` pour tous les attributs
- Mettre la visibilité `public` pour toutes les méthodes

Mettons la visibilité `private` pour tous les attributs de la classe `Personne`

```
export class Personne {  
    private num: number;  
    private nom: string;  
    private prenom: string;  
}
```

Dans le fichier `file.ts`, les trois lignes suivantes sont soulignées en rouge

```
personne.num = 1;  
personne.nom = "wick";  
personne.prenom = "john";
```

Explication

Les attributs sont privés, donc aucun accès direct n'est autorisé

Solution : les setters

Des méthodes qui contrôlent l'affectation de valeurs aux attributs

Conventions TypeScript

- Le setter est une méthode déclarée avec le mot-clé set.
- Il porte le nom de l'attribut.
- On l'utilise comme un attribut.
- Pour éviter l'ambiguïté, on préfixe l'attribut par un underscore.

Nouveau contenu de la classe Personne après ajout des setters

```
export class Personne {  
    private _num: number;  
    private _nom: string;  
    private _prenom: string;  
  
    public set num(_num : number) {  
        this._num = (_num >= 0 ? _num : 0);  
    }  
    public set nom(_nom: string) {  
        this._nom = _nom;  
    }  
    public set prenom(_prenom: string) {  
        this._prenom = _prenom;  
    }  
}
```

Pour tester, rien à changer dans file.ts

```
import { Personne } from './personne';  
  
let personne: Personne = new Personne();  
personne.num = 1;  
personne.nom = "wick";  
personne.prenom = "john";  
  
console.log(personne);
```

Le résultat est :

```
Personne { _num: 1, _nom: 'wick', _prenom: 'john' }
```

Testons avec une valeur négative pour l'attribut `numero`

```
import { Personne } from './personne';

let personne: Personne = new Personne();
personne.num = -1;
personne.nom = "wick";
personne.prenom = "john";

console.log(personne);
```

Le résultat est :

```
Personne { _num: 0, _nom: 'wick', _prenom: 'john' }
```

#Getters

Question

Comment récupérer les attributs (privés) de la classe `Personne` ?

Démarche

Définir des méthodes qui retournent les valeurs des attributs (les `getter`)

Conventions TypeScript

- Le getter est une méthode déclarée avec le mot-clé `get`.
- Il porte le nom de l'attribut.
- On l'utilise comme un attribut.

Ajoutons les getters dans la classe `Personne`

```
public get num(): number {
    return this._num;
}

public get nom(): string {
    return this._nom;
}

public get prenom(): string {
    return this._prenom;
}
```

Pour tester

```
import { Personne } from './personne';

let personne: Personne = new Personne();
personne.num = 1;
personne.nom = "wick";
personne.prenom = "john";

console.log(personne.num);
// affiche 1

console.log(personne.nom);
// affiche wick

console.log(personne.prenom);
// affiche john
```

Remarques

- Par défaut, toute classe **TypeScript** a un constructeur par défaut sans paramètre.
- Pour simplifier la création d'objets, on peut définir un nouveau constructeur qui prend en paramètre plusieurs attributs de la classe.

Les constructeurs avec **TypeScript**

- On le déclare avec le mot-clé `constructor`.
- Il peut contenir la visibilité des attributs si on veut simplifier la déclaration.

#constructeur avec arguments

Le constructeur de la classe Personne prenant trois paramètres

```
public constructor(_num: number, _nom: string, _prenom: string) {  
    this._num = _num;  
    this._nom = _nom;  
    this._prenom = _prenom;  
}
```

Pour préserver la cohérence, il faut que le constructeur contrôle la valeur de l'attribut num

```
public constructor(_num: number, _nom: string, _prenom: string) {  
    this._num = (_num >= 0 ? _num : 0);  
    this._nom = _nom;  
    this._prenom = _prenom;  
}
```

Dans file.ts, la ligne suivante est soulignée en rouge

```
let personne: Personne = new Personne();
```

Explication

Le constructeur par défaut a été écrasé (il n'existe plus)

Comment faire ?

- TypeScript n'autorise pas la présence de plusieurs constructeurs (la surcharge)
- On peut utiliser soit les valeurs par défaut, soit les paramètres optionnels

#constructeur avec arguments optionnels

Le nouveau constructeur avec les paramètres optionnels

```
public constructor(_num?: number, _nom?: string,  
_prenom?: string) {  
    if (_num)  
        this._num = _num;  
    if (_nom)  
        this._nom = _nom;  
    if (_prenom)  
        this._prenom = _prenom;  
}
```

Pour tester

```
import { Personne } from './personne';  
  
let personne: Personne = new Personne();  
personne.num = -1;  
personne.nom = "wick";  
personne.prenom = "john";  
console.log(personne);  
  
let personne2: Personne = new Personne(2, 'bob', 'mike');  
console.log(personne2);
```

En exécutant, le résultat est :

```
Personne { _num: 0, _nom: 'wick', _prenom: 'john' }  
Personne { _num: 2, _nom: 'bob', _prenom: 'mike' }
```

#Fusionner la déclaration des arguments dans le constructeur

TypeScript nous offre la possibilité de fusionner la déclaration des attributs et le constructeur

```
public constructor(private _num?: number,  
                  private _nom?: string,  
                  private _prenom?: string) {  
}
```

En exécutant, le résultat est le même

```
Personne { _num: 0, _nom: 'wick', _prenom: 'john' }  
Personne { _num: 2, _nom: 'bob', _prenom: 'mike' }
```

c) Heritage

L'héritage, quand ?

- Lorsque deux ou plusieurs classes partagent plusieurs attributs (et méthodes)
- Lorsqu'une Classe1 est (**une sorte de**) Classe2

Forme générale

```
class ClasseFille extends ClasseMère  
{  
    // code  
};
```

Particularité du langage **TypeScript**

- Une classe ne peut hériter que d'une seule classe
- L'héritage multiple est donc non-autorisé.

#Exemple

Préparons la classe Enseignant

```
import { Personne } from "./personne";  
  
export class Enseignant extends Personne {  
  
}
```

Préparons la classe Etudiant

```
import { Personne } from "./personne";  
  
export class Etudiant extends Personne {  
  
}
```

extends est le mot-clé à utiliser pour définir une relation d'héritage entre deux classes



On peut créer un objet de la classe Enseignant **ainsi**

```
let enseignant: Enseignant = new Enseignant(3, "green", "jonas", 1700);
```

Ou ainsi

```
let enseignant: Personne = new Enseignant(3, "green", "jonas", 1700);
```

Ceci est faux

```
let enseignant: Enseignant = new Personne(3, "green", "jonas");
```

d) Définition d'une Interface

i) Approche 1

En TypeScript

- Une classe ne peut hériter que d'une seule classe
- Mais elle peut hériter de plusieurs interfaces

Une interface

- déclarée avec le mot-clé `interface`
- comme une classe complètement abstraite (impossible de l'instancier) dont : toutes les méthodes sont abstraites
- un protocole, un contrat : toute classe qui hérite d'une interface doit implémenter toutes ses méthodes

#Définition

Définissons l'interface `IMiseEnForme` dans `i-mise-en-forme.ts`

```
export interface IMiseEnForme {
    afficherNomMajuscule(): void;
    afficherPrenomMajuscule() : void;
}
```

Pour hériter d'une interface, on utilise le mot-clé `implements`

```
export abstract class Personne implements IMiseEnForme {
    ...
}
```

Remarque

- Une interface peut hériter de plusieurs autres interfaces (mais pas d'une classe)
- Pour cela, il faut utiliser le mot-clé `extends` et pas `implements` car une interface n'implémente jamais de méthodes.

ii) Approche 2

Une deuxième utilisation

- En TypeScript, une interface peut être utilisée comme une classe **Model** de plusieurs autres interfaces (mais pas d'une classe)
- Elle contient des attributs (qui sont par définition publiques) et des méthodes (abstraites)

Exemple

```
export interface Person {  
    num: number;  
    nom: string;  
    prenom: string;  
}
```

Impossible d'instancier cette interface avec l'opérateur `new`, mais on peut utiliser les objets JavaScript

```
let person: Person = {  
    num: 1000,  
    nom: 'turing',  
    prenom: 'alan'  
};  
console.log(person);  
// affiche { num: 1000, nom: 'turing', prenom: 'alan' }
```

On peut rendre les attributs optionnels

```
export interface Person {  
    num?: number;  
    nom?: string;  
    prenom?: string;  
}
```

II. Réalisation des Modèles du Projet

A. Détermination des Modèles du Projet

B. Génération des Modèles

1. Créer un dossier models dans shared

2. Génération des interfaces

a) Commande : `ng g i <nom_modele>`

b) Interface Categorie : `ng g i categories`

#categories.ts

```
export interface Categories {  
    id?: number,  
    name: string  
}
```

c) Interface Produit : `ng g i products`

#products.ts

```
import { Categories } from './categories';  
  
export interface Products {  
    id?: number,  
    name: string,  
    description?: string,  
    oldPrice: number,  
    newPrice: number  
    qteStock: number  
    qteSeuil: number  
    isSolde: boolean  
    note?: number  
    categorie: Categories  
    pathImg: string,  
  
}
```

d) Interface User : `ng g i users`

#users.ts

```
export interface Users {  
    id?: number,  
    nomComplet: string,  
    login: string,  
    password: string,  
    role: string,
```

```
    token:string,  
    avatar:string,  
  
}
```

e) Interface Commande : ng g i commandes

#commandes.ts

```
import { Products } from './products';  
import { Users } from './users';  
export interface Commandes {  
  id?:number  
  date:string,  
  mntTotal:number  
  isPayed:boolean  
  isLivred:boolean  
  client : Users  
  products :Products[]  
  
}
```

f) Interface Commande : ng g i panier

#panier.ts

```
import { Products } from './products';  
import { Users } from './users';  
export interface Panier {  
  client?: Users  
  products:Products[]  
  
}
```

Cours 4

DataBinding , Directives

de Structures et

les Services

A.Data Binding

Le databinding est un mécanisme qui permet la synchronisation instantanément de la vue (HTML Template) et les données de l'application(fichier Ts) .

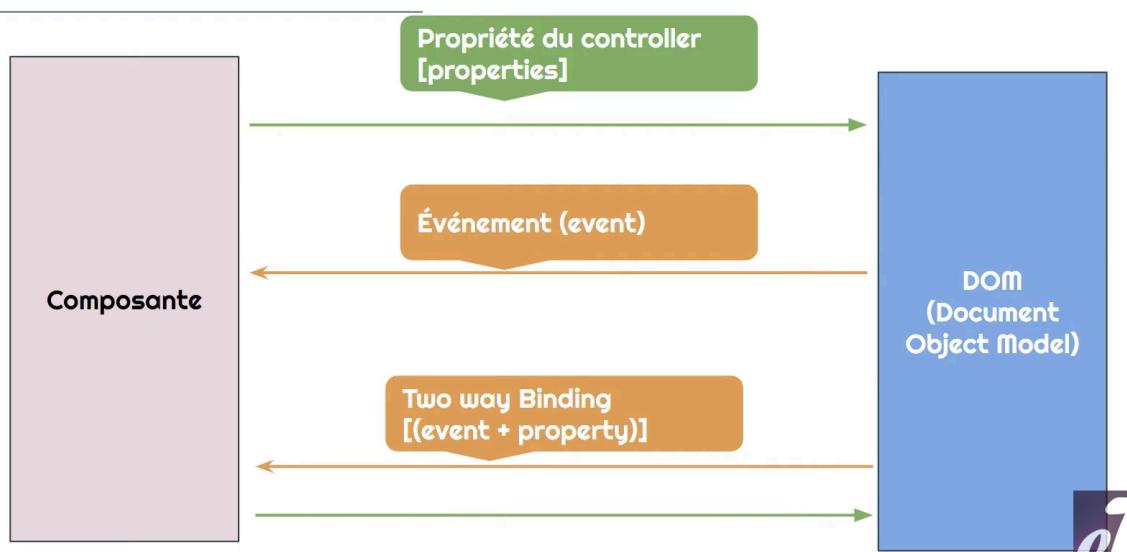
Le databinding permet de faire la communication entre le component(.ts) et son template(.html). Angular propose plusieurs types de databinding qui seront utilisés selon le besoin.

On distingue plusieurs types de databinding. Nous allons regrouper ces types dans des catégories selon le sens du flux des données.

Il y a trois sens pour le flux:

- Du component vers la template(Donnée)
- ou inversement de la template vers le component (Événements)
- et finalement un Databinding dans le deux sens

Binding



I. Binding du Composant vers le Template

a. Interpolation

Comme de nombreux langages de "templating", Angular utilise la syntaxe "**double curly braces**" ({{}}) pour l'interpolation . L'interpolation permet d'afficher un **attribut public** de la classe du fichier ***.ts** dans la vue du component ***.html**.

Binding (One-Way)



#AppComponent

app.component.ts

```
export class AppComponent {  
  title = 'demo-ecom';  
}
```

app.component.html

```
<h2>Nom de Mon Projet {{title}}</h2>
```

NB :

L'interpolation ne doit être utilisée pour contrôler les attributs d'un élément par exemple `</balise>.
```

### Exemple

#### #AppComponent

#### app.component.ts

```
path_img: string = "https://dummyimage.com/450x300/dee2e6/6c757d.jpg"
```

#### app.component.html

```

```

### Exemple :

- Afficher la valeur d'une variable dans un champ de texte

#### #AppComponent

#### app.component.ts

```
value: string = "Ma Valeur"
```

### ■ Interpolation

#### app.component.html

```
<input type="text" name="" id="" value="{{value}}">
```

### ■ Property Binding

#### app.component.html

```
<input type="text" name="" id="" [value]="value">
```

- Désactiver un dans un champ de texte

## #AppComponent

### app.component.ts

```
isDisabled:boolean=true;
```

#### ■ Interpolation

### app.component.html

```
<input type="text" name="" id="" disabled="{{isDisabled}}>
```

#### ■ Property Binding

### .app.component.html

```
<input type="text" name="" id="" [disabled]="isDisabled">
```

## II. Binding du Template vers le Composant ou Event Binding

Event Binding permet d'envoyer des données du template vers le composant, cet envoi se fait à l'aide des événements du HTML.

### Syntaxe Utilisation

```
<balise(event)="MaFonction()"> Click me! </balise>
```

- MaFonction() doit être définie dans le component

NB : On peut aussi utiliser le préfixe "on-" juste avant le nom de l'événement à utiliser. Donc la syntaxe devient comme suivant:

```
<balise on-event="MaFonction()"> Click me! </balise>
```

## Exemple :

Récupérer une valeur saisie dans un champ input de type texte puis l'afficher dans un span

### #AppComponent

#### app.component.ts

```
value:string="Ma Valeur"
getValue(event:any) {
 this.value=event.target.value;
}
```

#### app.component.html

```
<input type="text" name="" id="" (keyup)="getValue($event)"
>{{value}}
```

### III. Binding dans les deux sens Two Data Binding

Le Binding Bidirectionnel permet de faire une synchronisation instantanée entre la template et la valeur de la propriété du composant. Cette synchronisation est dans les deux sens. Ce type Binding utilise la directive **ngModel** se trouvant dans **FormsModule**.

Le data binding Bidirectionnel est la combinaison des deux Bindings précédents :

```
[propertyBinding] + (eventBinding) = [(twoWayBinding)]
```

#### Syntaxe d'utilisation:

Importer le **FormsModule** dans le **App.module.ts** qui permettra d'utiliser la directive **ngModel**.

## #App.module.ts

```
imports: [FormsModule],
```

```
<balise [(ngModel)]="variable"> </balise>
```

### Exemple 1 :

Récupérer une valeur saisie dans un champ input de type texte puis l'afficher dans un span

## #AppComponent

### app.component.ts

```
value:string="Ma Valeur"
```

### app.component.html

```
<input type="text" name="" id="" [(ngModel)]="value">
{{value}}
```

### Exemple 2 :

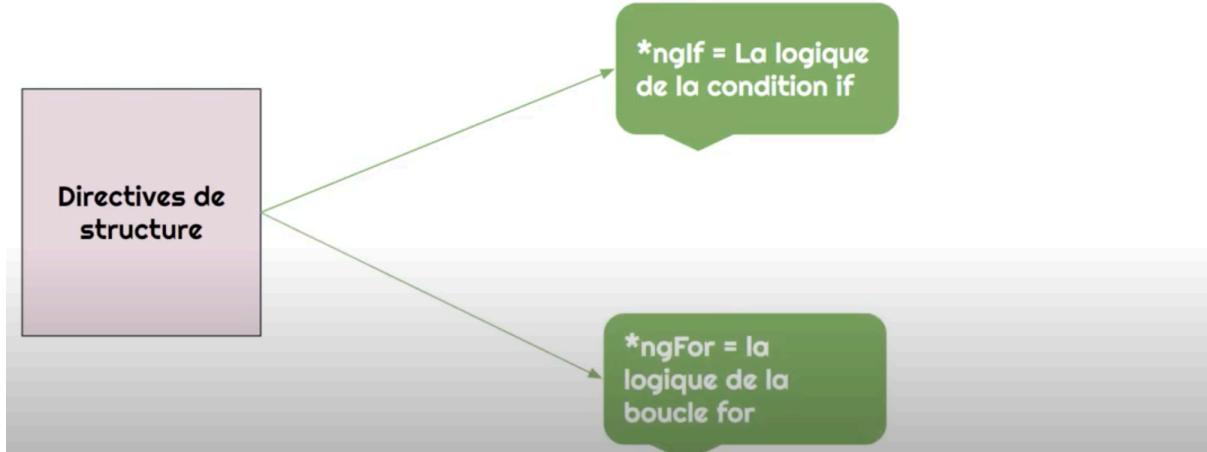
## B.Directives de Structure

Les directives sont des classes permettant d'enrichir et de modifier les vues (\*.html) par simple **ajout d'attributs HTML sur le template ou de balises**. On distingue deux types de directives : les **Directives de Structure** et les **Directives Attributs**.

Les **Directives de Structure** permettent de manipuler le DOM(Data Object Model) en **ajoutant** ou en **retirant** des éléments ou balises.

Les Directives de Structures natives les plus utilisées sont **ngIf** et **ngFor**.

# Directive



## 3. Directives de Structure **ngIf**

La directive structurelle **ngIf** permet d'injecter ou de retirer les éléments dans le DOM.

### Approche 1

#AppComponent

app.component.ts

```
export class AppComponent {
 title = 'e-shop';
 trouve: boolean=false;
}
```

app.component.html

```
<h2 *ngIf="trouve">Nom de Mon Projet {{title}}</h2>
```

### Approche 2

#AppComponent

app.component.html

```
<ng-template [ngIf]="trouve" >
 <h2 >Nom de Mon Projet {{title}}</h2>
</ng-template>
```

## NB :

Le tag **ng-template** est un élément de modèle que Angular utilisé avec les directives structurelles (**\*ngIf**, **\*ngFor**, **[ngSwitch]** et les directives personnalisées). Ces éléments de modèle ne fonctionnent qu'en présence de directives structurelles, qui nous aident à définir un modèle qui ne rend rien par lui-même, mais les rend conditionnellement au DOM. Il nous aide à créer des modèles dynamiques qui peuvent être personnalisés et configurés.

**Remarques :** Pour implémenter le **if..else**, la directive **ngIf** nous donne deux syntaxes :

## #AppComponent

### app.component.html

#### Syntaxe 1

```
<div *ngIf="trouve; else elseBlock">
 Contenu affiché lorsque la condition est vraie
</div>
<ng-template #elseBlock>
 Contenu affiche lorsque la condition est fausse
</ng-template>
```

#### Syntaxe 2

```
<ng-container *ngIf="trouve; then thenBlock else elseBlock"></ng-container>
<ng-template #thenBlock>
 Contenu affiche lorsque la condition est vraie
</ng-template>
<ng-template #elseBlock>
 Contenu affiche lorsque la condition est fausse
</ng-template>
```

## 4. Directives de Structure **ngFor**

La directive structurelle **ngFor** permet de boucler sur un array et d'injecter les éléments dans le DOM.

## #AppComponent

### app.component.ts

```
export class AppComponent {
 title = 'e-shop';
 trouve:boolean=true;
 numbers:number[]=[1,2,4,5,6,7]
}
```

### app.component.html

```
<ng-container *ngIf="numbers.length>0">
 <li *ngFor="let nbre of numbers">{{nbre}}
</ng-container>
```

#### NB :

- Le tag **ng-container** est une directive extrêmement simple qui vous permet de regrouper des éléments dans un modèle qui n'interfère pas avec les styles ou la mise en page car Angular ne le met pas dans le DOM. Ceci est utile si vous ne voulez pas de div supplémentaire sur DOM.

**Exemple :**

```
<div *ngIf="details" *ngFor="let info of details">
 {{ info.content }}
</div>
```

Ce code génère une erreur car impossible d'avoir plusieurs directives de structures sur un élément.

#### Correction

```
<ng-container *ngIf="details">
 <div *ngFor="let info of details">
 {{ info.content }}
</div>
</ng-container>
```

## C.Les Services

### a. Concepts

#### Service

**Classe à responsabilité particulière**



Un service est une classe Type Script composée d'attributs et de méthodes, dont l'instanciation est gérée par Angular.

Un service est, sauf cas spécifique, **un singleton** (design pattern) : une seule instance de l'objet est utilisée à travers toute l'application.

Une fois instancié, il est injectable dans n'importe lequel de vos composants ou dans un autre service.

**Les Services permettent de :**

- réutiliser du code entre différents composants
- faciliter l'échange des données
- centraliser les appels de service
- séparer les responsabilités visuelles (component) et fonctionnelles/techniques (service)

**Les Cas d'utilisation d'un Service**

Communication avec une API :

- Communication avec une base de données
- Implémentation d'un cache d'objets
- Gestion de la session utilisateur côté client
- 

## b. Générer le Service

**ng g s <nom\_service>**

Cette commande effectue 4 actions :

- 1) Créer la Classe Typescript
- 2) Ajouter le décorateur **@Injectable**, pour que le service puisse faire objet d'une injection de dépendance dans un composant ou dans un autre service
- 3) Déclaration du Service dans un module , composant à travers Injector

## c. L'injecteur de dépendances d'Angular

Il existe plusieurs injecteurs de dépendance dans Angular qui interviennent à différents niveaux :

- **L'injecteur root** : Il s'agit de l'injecteur principal, accessible à travers tous les modules, services, components, .etc de votre application.

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

- **L'injecteur par module** : lors de la création d'un service la propriété **providedIn** permet de spécifier le module qui sera chargé de l'injection. Le service ne sera alors pas accessible en dehors du module (sauf si import du module)

```
@Injectable({
 providedIn: NomModule
})
```

- L'injecteur par composant : permet de spécifier quels sont les services à injecter pour un composant, en précisant la propriété **providers** au niveau du composant .

#### #Dans le Service

```
@Injectable() //pas de providedIn
```

#### #Dans le Composant

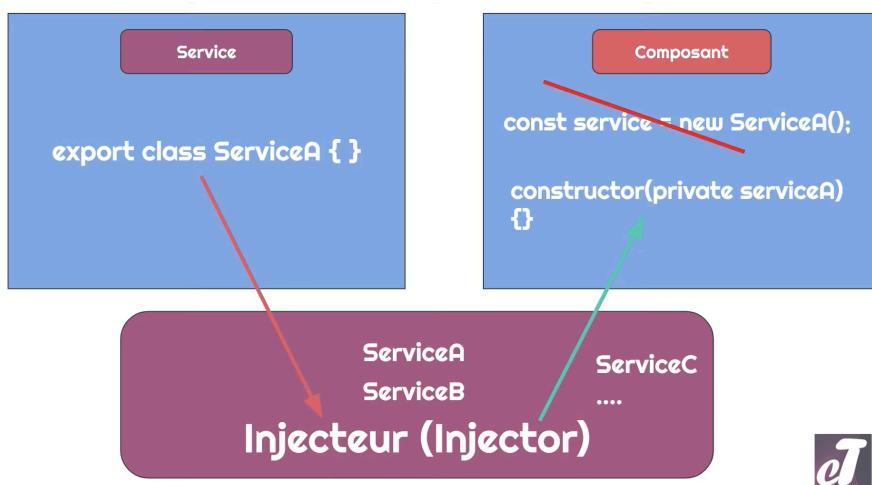
```
@Component({
 providers: [ProductsServiceService],
})

export class AppComponent {}
```

**NB :** Les services à injecter ne seront instanciés qu'à l'instanciation du composant. Notez que dans ce cas, **les services ne sont plus des singletons**. Car il peut exister plusieurs instances au sein d'une même exécution.

## d. Utilisation d'un Service

Un service peut être injecté dans un autre service, ou dans un composant, en l'ajoutant en paramètre du constructeur .



**Exemple :**

```
constructor(private prodServ: ProductsServiceService) {
}
}
```

# Cours 5

## RxJs :Les Observables

### A.RxJs et Observable

#### a. RxJs (Reactive Extension for JavaScript)

La programmation réactive est un paradigme de programmation asynchrone concerné par les flux de données et la propagation du changement . RxJS (**\*R\*eactive E\*x\*tensions pour \*J\*ava\*S\*cript**) est une bibliothèque de programmation réactive utilisant des observables qui facilite la composition de code asynchrone ou basé sur le rappel.

RxJS fournit une implémentation du type **Observable**. La bibliothèque fournit également des fonctions utilitaires pour **créer et travailler** avec des observables. Ces fonctions utilitaires peuvent être utilisées pour :

- Conversion du code existant pour les opérations asynchrones en observables
- Itérer sur les valeurs d'un flux
- Mappage de valeurs à différents types
- Filtrage des flux
- Composer plusieurs flux

NB : RxJS 6 est une dépendance obligatoire à partir d'Angular 6

## b. Les Observables

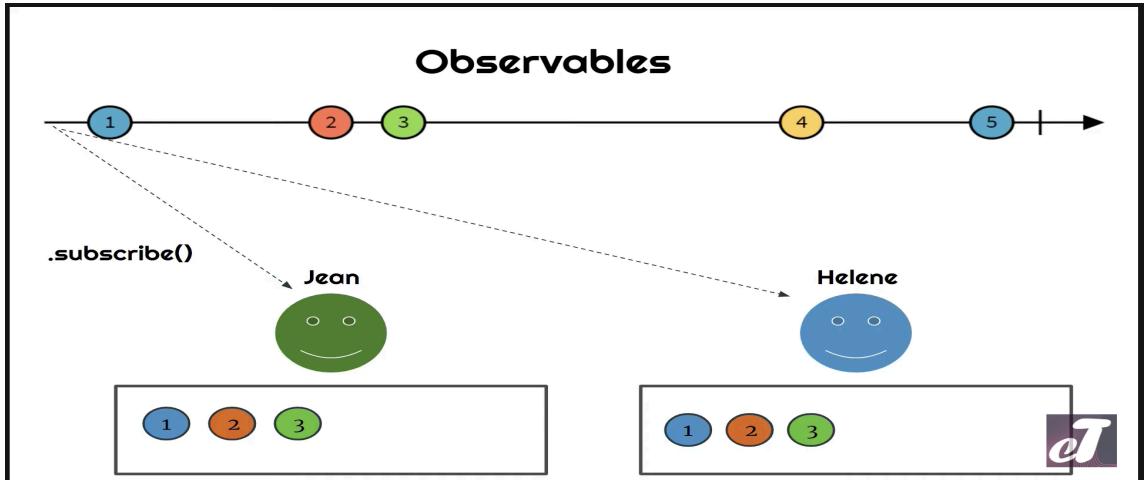
### i. Définition

Les Observables est un nouveau type primitif qui sert de modèle pour la façon dont nous voulons créer des flux, nous y abonner, réagir à de nouvelles valeurs et combiner des flux pour en créer de nouveaux.

**NB :** Les observables pourraient devenir une partie essentielle du langage JavaScript à l'avenir, nous pouvons donc considérer RxJS comme un espace réservé pour le moment où cela arrivera.

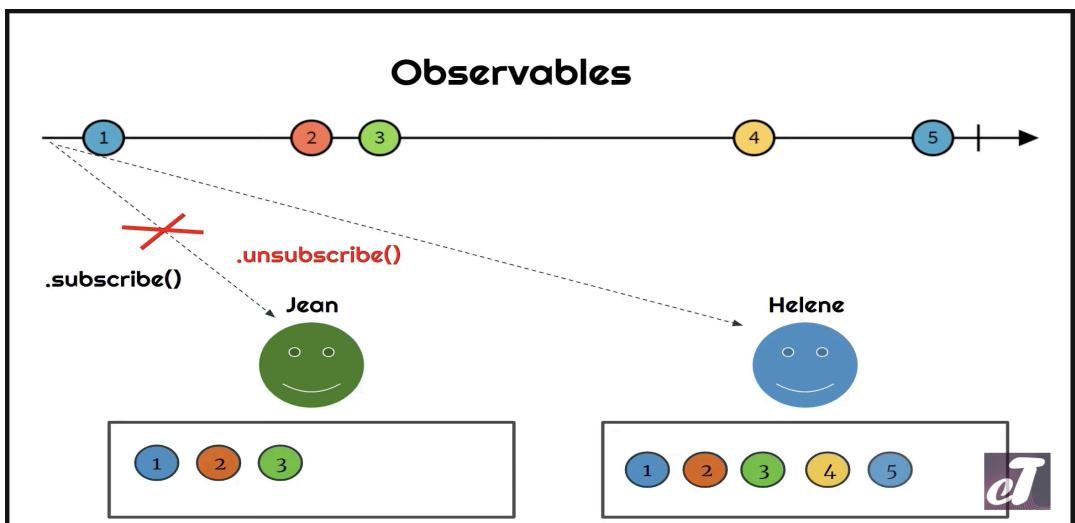
### ii. Abonnement ou souscription(subscribe)

Un abonnement est simplement un moyen de démarrer une "exécution observable" de fournir des valeurs ou des événements à un observateur de cette exécution. S'abonner à un Observable, c'est comme appeler une fonction, fournir des rappels où les données seront livrées.



### iii. Désabonnement (unsubscribe)

Un **désabonnement** est un moyen d'arrêter une "**exécution observable**" donc **l'observateur** ne reçoit plus des valeurs ou des événements de cet observable .



## c. Les Opérateurs de RxJs

RxJS est surtout utile pour ses opérateurs , même si l'Observable en est la base. Les opérateurs sont les éléments essentiels qui permettent de composer facilement du code asynchrone complexe de manière déclarative.

Les opérateurs sont des fonctions . Il existe deux types d'opérateurs les opérateurs de création, les Pipeable opérateur et l'opérateur pipe()

#### a. Les opérateurs de création

Ce sont des fonctions autonomes qui peuvent être appelées pour créer un nouvel Observable.

Parmi ces opérateurs on peut citer :

- `of(1, 2, 3)` crée un observable qui émettra 1, 2 et 3, l'un après l'autre.
- `interval(1000)`: crée un observable qui émettra des valeurs entre 1 et 1000 de manière alternative les uns après les autres.

NB : Convention de Nommage des Observables

Le framework Angular n'applique pas de convention de dénomination pour les observables, mais vous verrez souvent des observables nommés avec un signe "\$"

Exemple : Crédit Observable

```
private obsvJour$: Observable<string>=of("Lundi","Mardi","Mercredi");

private obsvNumbers1$: Observable<number>=range(1,6);

private obsvNumbers2$: Observable<number>=interval(6);
```

```
private obsvArrNumber$: Observable<number[]>=of([1,3,5,6]);

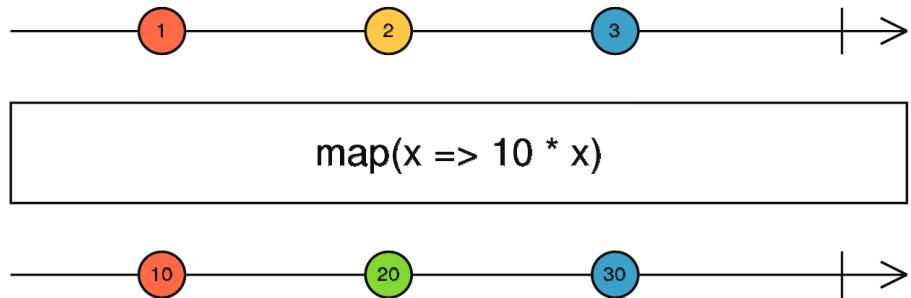
private obsvArrProduts$: Observable<Products[]>=of([]);
```

### b. Les opérateurs Pipeable

Pipeable Operator est une fonction qui prend un Observable comme entrée et renvoie un autre Observable. C'est une opération pure, c'est-à-dire l'observable de base reste inchangé.

Parmi ces opérateurs on peut citer :

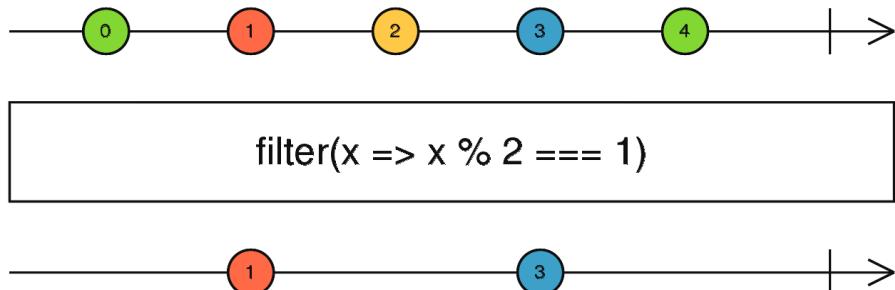
- **map()** : reçoit un observable en entrée et renvoie un autre Observable dont les valeurs ont subi la même modification



**Exemple :** Créez un nouvel observable qui ajoute +2 à chaque valeur de l'observable obsvNumbers1

```
obsvNumbers1$.pipe(map(nbre=>nbre+2))
```

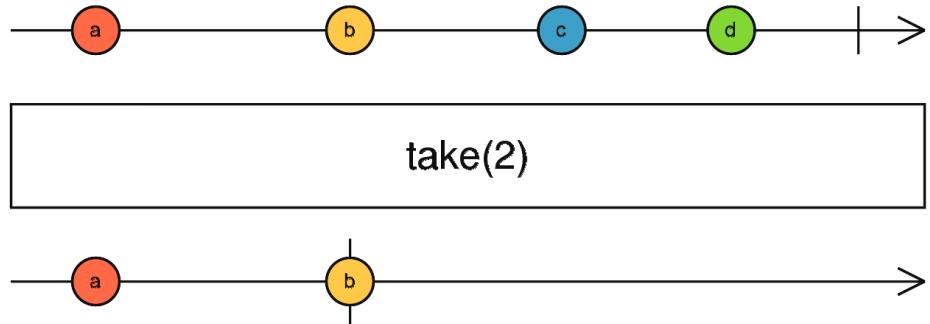
- **filter()** : reçoit un observable en entrée et renvoie un autre Observable qui vérifie le critère de filtre.



**Exemple :**Créer un nouvel observable qui contient les valeurs paires de l'observable `obsvNumbers1`

```
obsvNumbers1$.pipe(filter(nbre=>nbre%2==0))
```

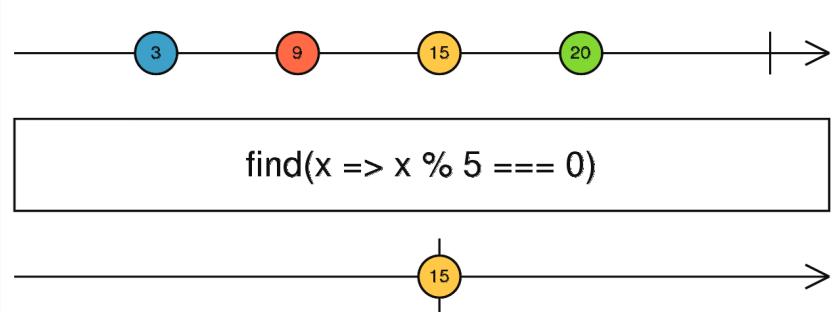
- **take(n)** : reçoit un observable en entrée et renvoie un autre Observable qui contient les n premières valeurs.



**Exemple :**Créer un nouvel observable qui contient 2 premiers Jours de l'observable obsvJour\$.

```
obsvJour$.pipe(take(2))
```

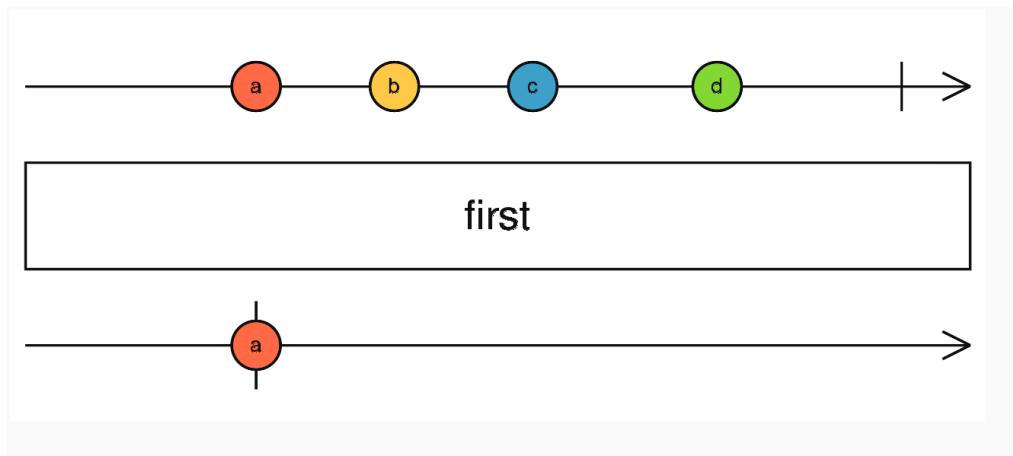
- **find()**:Créer un nouvel observable qui contient la première valeur qui vérifie le critère de filtre



**Exemple :**Créer un nouvel observable qui contient la premiers valeurs paires de l'observable obsvNumbers1

- **first()**: Créer un nouvel observable qui contient le premier Jour de l'observable obsvJour\$.

```
obsvJour$.pipe(first())
```



Source : <https://runebook.dev/fr/docs/rxjs/-index-#operators>

### c. l'opérateur pipe()

Les **opérateurs pipeables** sont des fonctions, ils pourraient donc être utilisés comme des fonctions ordinaires : `operator() (observableInstance)`.

Mais en pratique, ils ont tendance à être nombreux et à devenir rapidement illisibles :

`operator2() (operator1() (observableInstance))`

Pour cette raison, les Observables ont une méthode appelée `.pipe()` qui accomplit la même chose tout en étant beaucoup plus facile à lire :

`observableInstance.pipe(operator1(), operator2())`

### d. Exercice Application :

1. Dans la classe `ProductsServiceService` définir :

- un observable qui contient un tableau de nombres
- une fonction `getNumber()` qui retourne un observable de tableau de nombres

- une fonction `getNumberPair()` qui retourne observable de tableau de nombres

2. Dans la classe `AppComponent` définir :

- Injecter le Service `ProductsServiceService` dans le constructeur
- Implémenter l'interface `OnInit` et redéfinir la méthode `ngOnInit()` en faisant l'abonnement aux observables retournés par les méthodes `getNumber()` et `getNumberPair()` de `Product Service Service`. Ces valeurs retournée seront stockées dans des attributs de cette classe.

3. Dans la vu `app.component.html` définir :

- Faire un `oneWayBinding` pour récupérer les valeurs des attributs définies dans `AppComponent` puis les afficher.

```
export class ProductsServiceService {
 private source$: Observable<number[]>;
 constructor() {
 this.source$=of([1,3,5,6]);
 }
 getNumber () :Observable<number[]>{
 return this.source$;
 }
 getNumberPair () :Observable<number[]>{
 return this.source$.pipe(
 map(x=>{
 return x.filter(y=>y%2==0)
 })
)
 }
}
```

## #App.module.ts

```
constructor(private prodServ:ProductsServiceService) {
}

ngOnInit(): void {
 this.prodServ.getNumber()
 .subscribe(arg => this.numbers = arg)
}
```

## #App.component.html

```
<ng-container *ngIf="numbers.length>0">

 <li *ngFor="let nbre of numbers">{{nbre}}

</ng-container>
```

## B. Projet : Réalisation des Fonctionnalités

### a. Affichage du Catalogue de Produit

#### i. Service Product

##### 1. Génération du Service

##### ng g s product

##### 2. Définition de Observable du tableau Product

```
private produits$: Observable<Products[]>=of([
 {
 id:1,
 name:"Produit 1",
 description:"Description Produit 1",
 oldPrice: 50000,
 newPrice: 0,
 qteStock:15,
 qteSeuil:2,
 isSolde:false,
 note: 4,
 categorie:{
 id:1,
 name:"Categorie 1"
 },
 pathImg:'https://dummyimage.com/450x300/dee2e6/6c757d.jpg',
 },
 {
 id:2,
 name:"Produit 2",
 description:"Description Produit 1",
 oldPrice: 50000,
 newPrice: 30000,
 qteStock:15,
 qteSeuil:2,
 isSolde:true,
 note: 0,
 categorie:{
 id:1,
 name:"Categorie 1"
 },
 pathImg:'https://dummyimage.com/450x300/dee2e6/6c757d.jpg',
 }
]);
```

### 3. Définir la méthode `getProducts()`

```
getProducts(): Observable<Products[]>{
 return this.products$;
}
```

#### ii. Dans le component `products-front-list.component.ts`

1. Injecter le service dans le constructeur du component

```
constructor(private productsServ:ProductsServiceService) { }
```

#### 2. Implémenter l'interface `OnInit` et redéfinir la méthode `ngOnInit()`

```
export class ProductsFrontListComponent implements OnInit {
```

#### 3. Faire l'abonnement à l'observable retourne la méthode `getProducts()` puis récupérer la liste des produits

```
public products :Products=[];

ngOnInit(): void {
 this.productsServ.getProducts().subscribe(
 products=>this.products=products
)
}
```

#### iii. Dans le template `products-front-list.component.html`

1. Faire un `oneWayDataBinding` pour récupérer les valeurs des attributs définies dans le component puis les afficher

```
<section class="py-5">

 <div class="container px-4 px-lg-5 mt-5">

 <div class="row gx-4 gx-lg-5 row-cols-1 row-cols-md-3 row-cols-xl-4
justify-content-center">

 <ng-container *ngIf="products.length>0" >
```

```
<div class="col mb-5" *ngFor="let product of products">

 <div class="card h-100">

 <!-- Sale badge-->

 <div *ngIf="product.isSolde" class="badge bg-dark text-white position-absolute" style="top: 0.5rem; right: 0.5rem">Solde</div>

 <!-- Product image-->

 <!-- Product details-->

 <div class="card-body p-4">

 <div class="text-center">

 <!-- Product name-->

 <h5 class="fw-bolder">{{product.name}}</h5>

 <!-- Product price-->

 <ng-container *ngIf="product.isSolde; then isSolded else notSolded" ></ng-container>

 <ng-template #notSolded >

 {{product.oldPrice}} CFA

 </ng-template>

 <ng-template #isSolded >

 {{product.oldPrice}} CFA

 {{product.newPrice}} CFA

 </ng-template>

 </div>

 </div>

 <!-- Product actions-->
```

```

<div class="card-footer p-2 pt-0 border-top-0 bg-transparent row">

 <div class="text-center col-6"><a class="btn btn-sm
btn-outline-info" href="#">Panier</div>

 <div class="text-center col-6"><a class="btn btn-sm
btn-outline-success" routerLink="/products-front/details/{{product.id}}"
routerLinkActive="active">Details

 </div>

</div>

```

## b. Filtrer les produits par catégorie

### i. Dans le **ProductService**

#### 1. Définir la méthode **getProductByCategorie()**

```

getProductsByCategorie(id: number): Observable<Products[]>{
 return this.products$.pipe(
 map(products=>{
 return products.filter(product=>product.categorie.id==id)
 })
)
}

```

### ii. Dans le component **products-front-list.component.ts**

1. Faire l'abonnement à l'observable  
retourne la méthode  
**getProductByCategorie()** puis récupérer la  
liste des produits

## c. Affichage des Catégories dans le Menu

### i. Dans le service **CategorieService**

#### 1. Définir Observable de Categories

```
private categories:Observable<Categories[]>=of(
 [
 {
 id:1,
 name:"Categorie 1"
 },
 {
 id:2,
 name:"Categorie 2"
 }
]
)
```

#### 2. Définir la méthode **getCategories()**

```
getCategories():Observable<Categories[]>{
 return this.categories;
}
```

### ii. Dans le menu.component

#### 1. Dans .ts

##### #Souscription à Observable de Catégories

```
ngOnInit(): void {

 this.catService.getCategories().subscribe(
 categories=>this.categories=categories
)
}
```

#### 2. Dans .html

##### #Affichage des Catégories

```
<li *ngFor="let categorie of categories">
 {{categorie.name}}

```

## d. Filtrer les Produits une Catégorie

### i. Dans le menu.component

#### 1. Dans le fichier .ts

#Ajouter la fonction de chargement de la vue

```
async onLoadView(id:any){
 await this.router.navigateByUrl('.', { skipLocationChange: true });
 this.router.navigateByUrl(`/products-front/categorie/${id}`)
}
```

#### 2. Dans le fichier .html

#Activer l'événement

```
<li *ngFor="let categorie of categories"><a class="dropdown-item"
 (click)="onLoadView(categorie.id)">
 {{categorie.name}}

```

## e. Affichage du Détail un Catalogue d'un Produit

### i. Dans le ProductService

#### 1. Définir la méthode `getProductById()`

```
getProductById(id:number):Observable<Products[]>{
 return this.products$.pipe(
 map(products=>{
 return products.filter(product=>product.id==id)
 })
)
}
```

### ii. Dans le component `products-front-details.component.ts`

#### 1. Injecter le service dans le constructeur du component

```
constructor(private route:ActivatedRoute,
 private productsServ:ProductsServiceService) { }
```

#### 2. Faire l'abonnement à l'observable retourne la méthode `getProductById()` puis récupérer le produit

#### 3. Implémenter l'interface OnInit et redéfinir la méthode `ngOnInit()`

```
public product :Products| null=null;
public products :Products[]=[];
```

```

ngOnInit(): void {
 const id:number=this.route.snapshot.params['id'];
 this.productsServ.getProductById(id).subscribe(
 product=>this.product = product[0]
)
 this.productsServ.getProducts().subscribe(
 product=>this.products = product
)
}

```

### iii. Dans le template products-front-details.component.html

1. Faire un oneDataWayBinding pour récupérer les valeurs des attributs définies dans le component puis les afficher

```

<section class="py-5">
 <div class="container px-4 px-lg-5 my-5">
 <div class="row gx-4 gx-lg-5 align-items-center">
 <div class="col-md-6"></div>
 <div class="col-md-6">
 <div class="small mb-1">{{product?.categorie?.name}}</div>
 <h1 class="display-5 fw-bolder">{{product?.name}}</h1>
 <div class="fs-5 mb-5">
 <ng-container *ngIf="product?.isSolde; then isSolded else
notSolded" ></ng-container>
 <ng-template #notSolded >
 {{product?.oldPrice}} CFA
 </ng-template>
 <ng-template #isSolded >
 <span class="text-muted
text-decoration-line-through">{{product?.oldPrice}} CFA
 {{product?.newPrice}} CFA
 </ng-template>
 </div>
 <p class="lead">{{product?.description}}</p>
 <div class="d-flex">
 <input class="form-control text-center me-3" id="inputQuantity"
type="num" value="1" style="max-width: 3rem" />
 <button class="btn btn-outline-dark flex-shrink-0"
type="button">
 <i class="bi-cart-fill me-1"></i>
 </button>
 </div>
 </div>
 </div>
 </div>
</section>

```

```

 Ajouter Panier
 </button>
</div>
</div>
</div>
</div>
</section>
<!-- Related items section-->
<section class="py-5 bg-light">
<div class="container px-4 px-lg-5 mt-5">
 <h2 class="fw-bolder mb-4">Produits Associes</h2>
 <div class="row gx-4 gx-lg-5 row-cols-2 row-cols-md-3 row-cols-xl-4 justify-content-start">
 <ng-container *ngIf="products.length>0" >
 <div class="col mb-5" *ngFor="let product of products">
 <div class="card h-100">
 <!-- Sale badge-->
 <div *ngIf="product.isSolde" class="badge bg-dark text-white position-absolute" style="top: 0.5rem; right: 0.5rem">Solde</div>
 <!-- Product image-->

 <!-- Product details-->
 <div class="card-body p-4">
 <div class="text-center">
 <!-- Product name-->
 <h5 class="fw-bolder">{{product.name}}</h5>
 <!-- Product price-->
 <ng-container *ngIf="product.isSolde; then isSolded else notSolded" ></ng-container>
 <ng-template #notSolded >
 {{product.oldPrice}} CFA
 </ng-template>
 <ng-template #isSolded >
 {{product.oldPrice}} CFA
 {{product.newPrice}} CFA
 </ng-template>
 </div>
 </div>
 <!-- Product actions-->
 <div class="card-footer p-2 pt-0 border-top-0 bg-transparent row ">
</pre>

```

```

 <div class="text-center col-6"><a class="btn btn-sm
btn-outline-info" href="#">Panier</div>
 <div class="text-center col-6"><a class="btn btn-sm
btn-outline-success" href="#">Details</div>
 </div>
 </div>
 </ng-container>

</div>
</div>
</section>

```

#### iv. Rechargement des détails à produit associé

#Dans le Template

```

<div class="text-center col-6"><a class="btn btn-sm btn-outline-success"
(click)="reloadView(product.id)"
routerLinkActive="active">Details
</div>

```

#Dans le Ts

```

async reloadView(idProduct: any) {
 await this.router.navigateByUrl('.', { skipLocationChange: true });
 this.router.navigateByUrl(`products-front/details/${idProduct}`)
}

```

## C. Les Subject

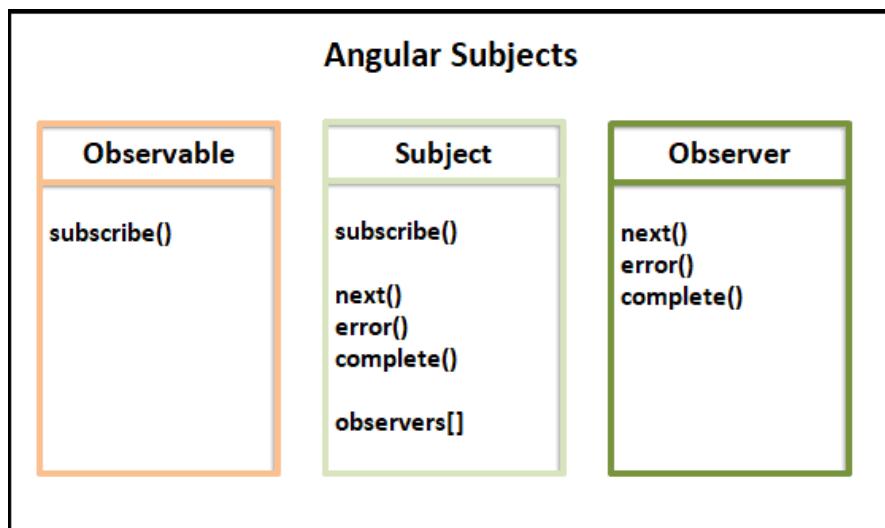
### i. Définition

Un **subject** RxJS est un type spécial d'observable qui permet aux valeurs d'être multi diffusées à de nombreux observateurs. Alors que les Observables simples sont unicast (chaque Observateur abonné possède une exécution indépendante de l'Observable), les Sujets sont multicast.

### ii. Utilisation

Les subject sont aussi des observateurs car ils peuvent s'abonner à un autre observable et en tirer de la valeur, qu'il diffusera à tous ses abonnés.

Fondamentalement, un subject peut agir à la fois comme **observable** et comme **observateur**.



### iii. Exercice Application

1. Dans la classe **ProductsServiceService** définir :

- Créer un Subjet de string
- Définir la méthode `envoyerMessage()`
- Définir la méthode `recevoirMessage()` qui retourne un observable

```
#ProductsServiceService
```

```
private subject$ = new Subject<string>();

envoyerMessage(sms:string):void{
 this.subject$.next(sms);
}

recevoirMessage():Observable<string>{
 return this.subject$.asObservable();
}
```

## 2. Envoie du Message

- a. Dans la classe **AppComponent** définir :
  - o Injecter le Service **ProductsServiceService** dans le constructeur
  - o Créer la méthode **OnEnvoieMessage()** qui est déclenché par l'événement **click**. Cette Méthode appelle **envoyerMessage()** de **Product Service Service**
- b. Dans la vu **app.component.html**, définir un **input[type="text"]** pour la saisie du message et un bouton qui déclenche l'événement **click**;

```
#app.component.html
```

```
<input type="text" #sms>
<button (click)="onEnvoie(sms.value)">Envoyer Message</button>
```

```
#app.component.ts
```

```
constructor(private proServ:ProductsServiceService) {}

onEnvoie(sms:string) {

 this.proServ.envoyerMessage(sms)

}
```

## 3. Réception du Message dans **HeaderComponent**

- a. Injecter le Service **ProductsServiceService** dans le constructeur
- b. Dans le **ngOnInit**, récupérer le message puis l'afficher dans la vue.

```
#header.component.ts

constructor(private productsServ:ProductsServiceService) { }

sms:string="";

ngOnInit(): void {

 this.productsServ.recevoirMessage().subscribe(sms => {

 this.sms=sms;

 });

}
```

```
#header.html
```

```
 {{sms}}
```

## D.Les BehaviorSubject

### a. Définition

Un **BehaviorSubject** a obligatoirement une valeur par défaut. Il sauvegarde la dernière valeur qu'il a émis et l'envoie aux observateurs lors de leur subscribe (si vous ne l'aviez pas remarqué, un observateur ne récupère pas les événements passés mais uniquement les nouveaux)

## E. Ajout dans le Panier

### i. Service Product

#### 1. Génération du Service

**ng g s panier**

#### 2. Le Model Panier

```
export interface Panier {
 client?: Users,
 products:Products[],
 total:number
}
```

### 3. Dans PanierService.ts

```
panier$:BehaviorSubject<Panier>=new BehaviorSubject<Panier>({
 products:[],
 total:0
});

getCard():Observable<Panier>{
 return this.panier$.asObservable();
}
clearCard(){
 this.panier$.next({
 products:[]
 })
}
addToCard(product:Products,qteCmde:number=1):void{
 product.qteCmde=qteCmde;
 let exist=false;
 this.getCard().pipe(take(1)).subscribe(
 panier=> {
 for (const prod of panier.products) {
 if(prod.id==product.id) {
 exist=true
 if(prod.qteCmde) prod.qteCmde+= qteCmde
 }
 }
 let total:number=panier.total
 var mnt:number=product.oldPrice??0
 if(product.isSolde){
 mnt=product.newPrice??0
 }
 total+=mnt*qteCmde
 const arr:Products[] = !exist?[...panier.products,product]:panier.products
 const panierElt:Panier={
 products:arr,
 total:total
 }
 this.panier$.next(panierElt);
 }
)
}
```

#### 4. Dans products-front-list.component.ts

```
onAddCard(product: Products) {
 this.panierService.addToCard(product)
}
```

#### 5. Dans products-front-list.component.html

```
<div class="text-center col-6"><button class="btn btn-sm btn-outline-info" (click)="onAddCard(product)">Panier</button></div>
```

#### 6. Dans menu.component.ts

```
constructor(private panierService: PanierService) { }
panier: Products[] = [];
ngOnInit(): void {
 this.panierService.getCard().subscribe(
 products => this.panier = products.products
)
}
```

#### 7. Dans menu.component.html

```
Panier
{{this.panier.length}}
```

## F. Ajout Panier a partir des Détails

#### 1. Dans detail.component.html

```
<div class="d-flex">
 <input class="form-control text-center me-3" id="inputQuantity" #qteCom type="num" value="1" style="max-width: 3rem"
 (keyup)="OnQteDisponible(qteCom.value)" />
 <button [disabled]="erreurQteComd" class="btn btn-outline-dark flex-shrink-0" type="button" (click)="onAddCard(product, qteCom.value)">
 <i class="bi-cart-fill me-1"></i>
 Ajouter Panier
 </button>


```

```

 </div>
 <p *ngIf="erreurQteComd" class="text-danger mt-2" >Qte non
disponible</p>

```

## 2. Dans detail.component.ts

```

onAddCard(product: any, qteCom: any=0) {
 this.panierServ.addToCard(product, qteCom)
}

OnQteDisponible(value: any) {
 let qteStock=this.product?.qteStock??0;
 this.erreurQteComd=qteStock<value || value=="" ? true:false;
}

```

# G. Affichage du Panier

## 1. Dans le component card

### a. Dans le .ts

```

panier:Panier={
 products:[],
 total:0
};

constructor(private panierServ:PanierService) { }

ngOnInit(): void {
 this.panierServ.getCard().subscribe(
 panier=> this.panier=panier
)
}

```

### b. Dans le .html

```

<section class="h-50 " style="background-color: #eee;">
 <div class="container py-5 h-75">
 <div class="row d-flex justify-content-center align-items-center h-100">
 <div class="col">
 <div class="card">
 <div class="card-body p-4">
 <div class="row">
 <div class="col-lg-7">
 <h5 class="mb-3"><i
 class="fas fa-long-arrow-alt-left me-2"></i>Continuer votre
Achat</h5>
 <hr>
 <div class="d-flex justify-content-between align-items-center
mb-4">

```

```

<div>
 <p class="mb-1">Panier Achat</p>
 <p class="mb-0">Vous avez {{panier.products.length}} Produit
dans votre Panier</p>
</div>
</div>
<div class="card mb-3" *ngFor="let product of panier.products ">
 <div class="card-body">
 <div class="d-flex justify-content-between">
 <div class="d-flex flex-row align-items-center">
 <div>

 </div>
 <div class="ms-3">
 <h5>{{product.name}}</h5>
 <p class="small mb-0">{{product.description}}</p>
 </div>
 </div>
 <div class="d-flex flex-row align-items-center">
 <div style="width: 50px;">
 <h5 class="fw-normal mb-0">{{product.qteCmde}}</h5>
 </div>
 <div style="width: 80px;">

<ng-container *ngIf="product.isSolde;then solded else
notSolded" ></ng-container>
 <ng-template #notSolded>
 {{product.oldPrice}}
 </ng-template>
 <ng-template #solded>
 {{product.newPrice}}
 </ng-template>
 </div>
 <i class="fas
fa-trash-alt"></i>
 </div>
 </div>
 </div>
 </div>
</div>

```

```

 </div>
 <div class="col-lg-5">
 <div class="card bg-light text-dark rounded-3 h-100">
 <div class="card-body">
 <div class="d-flex justify-content-between align-items-center mb-4">
 <h5 class="mb-0">Total Commande</h5>

 </div>

 <div class="d-flex justify-content-between">
 <p class="mb-2">Montant Hors Taxe</p>
 <p class="mb-2">{{panier.total}} CFA</p>
 </div>
 <div class="d-flex justify-content-between">
 <p class="mb-2">Taxes</p>
 <p class="mb-2">{{panier.total*0.18}} CFA</p>
 </div>
 <div class="d-flex justify-content-between mb-4">
 <p class="mb-2">Total</p>
 <p class="mb-2">{{panier.total+panier.total*0.18}} CFA</p>
 </div>
 <button type="button" class="btn btn-info btn-block btn-lg">
 <div class="d-flex justify-content-between">
 <a
 class="text-decoration-none"
 routerLink="/login"
 >
 Terminer la Commande<i class="fas fa-long-arrow-alt-right ms-2"></i>

 </div>
 </button>
 </div>
 </div>
 </div>
 </div>
 </div>

```

```
</div>
</div>
</section>
```

## Désactivation du bouton Terminer Commande

### #card.component.ts

```
disabledBtn:boolean=true;
```

```
ngOnInit(): void {
 this.panierServ.getCard().subscribe(
 panier=> {
 this.panier=panier
 this.disabledBtn= this.panier.products.length==0
 return this.panier;
 }
)
}
```

### #card.component.html

```
<button type="button" class="btn btn-info btn-block btn-lg"
[disabled]="disabledBtn">
 <div class="d-flex justify-content-between">
 <a
 class="text-decoration-none"
 routerLink="/login"
 >
 Terminer la Commande<i
class="fas fa-long-arrow-alt-right ms-2"></i>

 </div>
</button>
```

## H. Affichage des Commandes du Client

### I. Affichage des Détails d'une Commande

# **Cours 6 :**

# **Notion d'Api et les**

# **Formulaires**

# I. Api Rest

## A. Notion de Json

Le JavaScript Object Notation (JSON) est un format standard utilisé pour représenter des données structurées de façon semblable aux objets Javascript. Il est habituellement utilisé pour structurer et transmettre des données sur des applications web (par exemple, envoyer des données depuis un serveur vers un client afin de les afficher sur une page web ou vice versa).

Le JSON se présente sous la forme d'une chaîne de caractères -utile lorsque vous souhaitez transmettre les données sur un réseau. Il a donc besoin d'être converti en un objet JavaScript natif lorsque vous souhaitez accéder aux données.

{

```
date:"12-01-2020",

products:[

{

id:1,

qteComd:2000,

prix:3000

},

{

id:1,

qteComd:2000,

prix:3000
```

```
 }
```

```
]
```

```
}
```

## a. Les Fonctions de Conversion

### i. `JSON.stringify(data)`

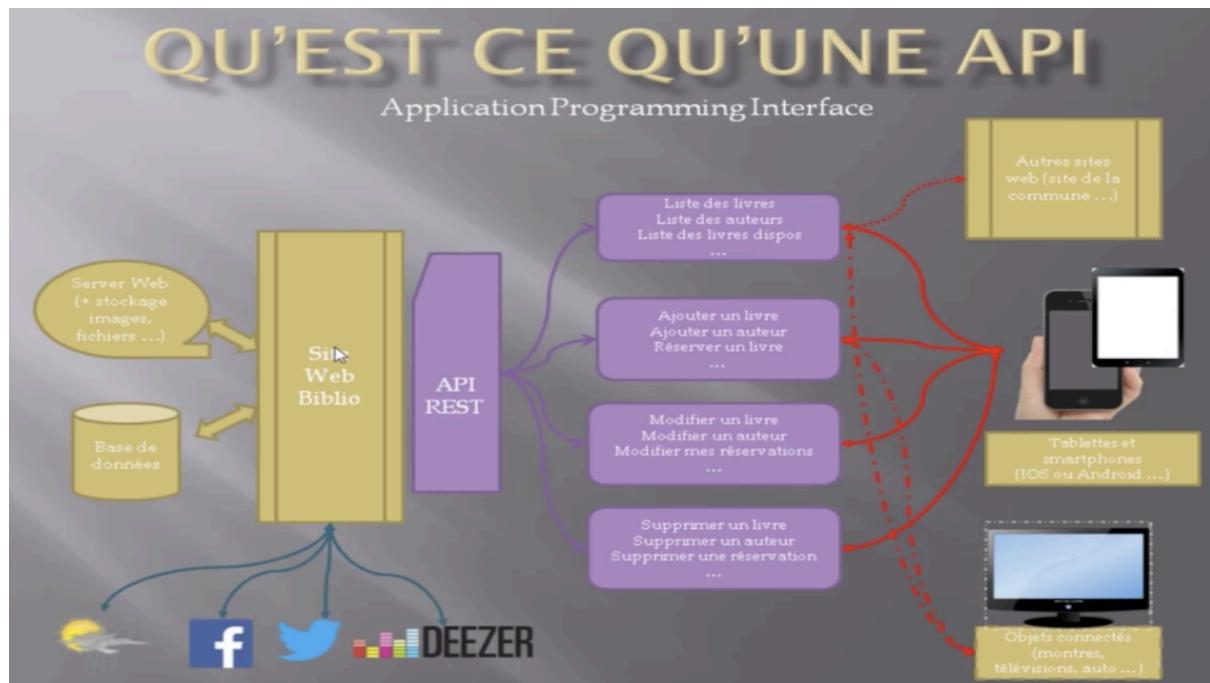
Cette fonction `JSON.stringify(data)` permet de sérialiser un objet c'est à dire le transformer en Json.

### ii. `JSON.parse(data)`

Cette fonction `JSON.parse(data)` permet de convertir du JSON en Objet.

```
{
```

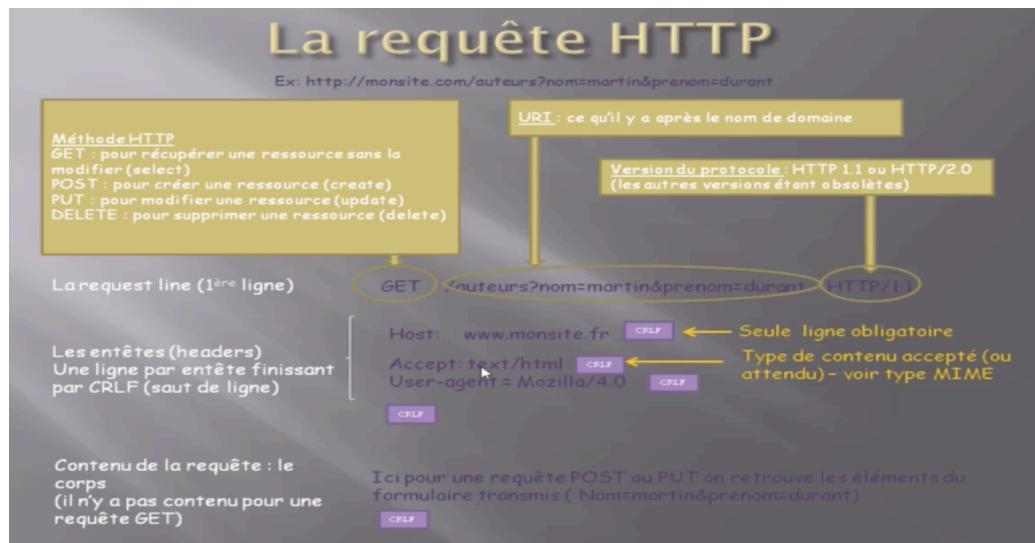
## B.Notion Api Rest



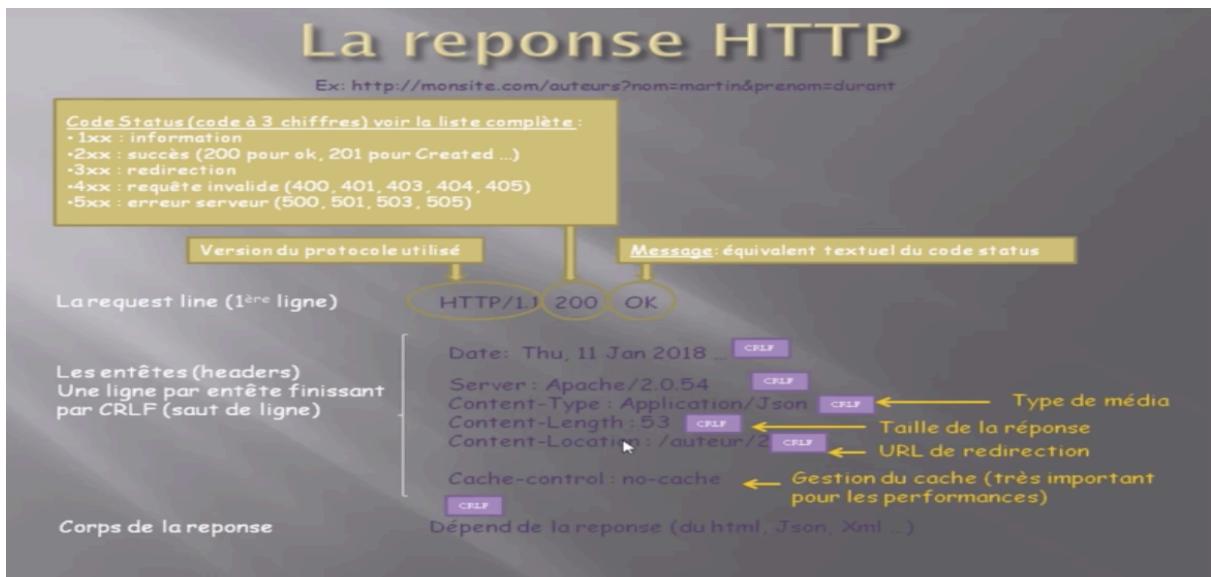
## C.Raisons et Intérêts des API

- Besoins de multiplier les points de consommations (Site web, Smartphone, Objets connectés)**
- Besoin de centraliser la couche métier (accès à la base de données, Stockage fichiers, accès par authentification)**

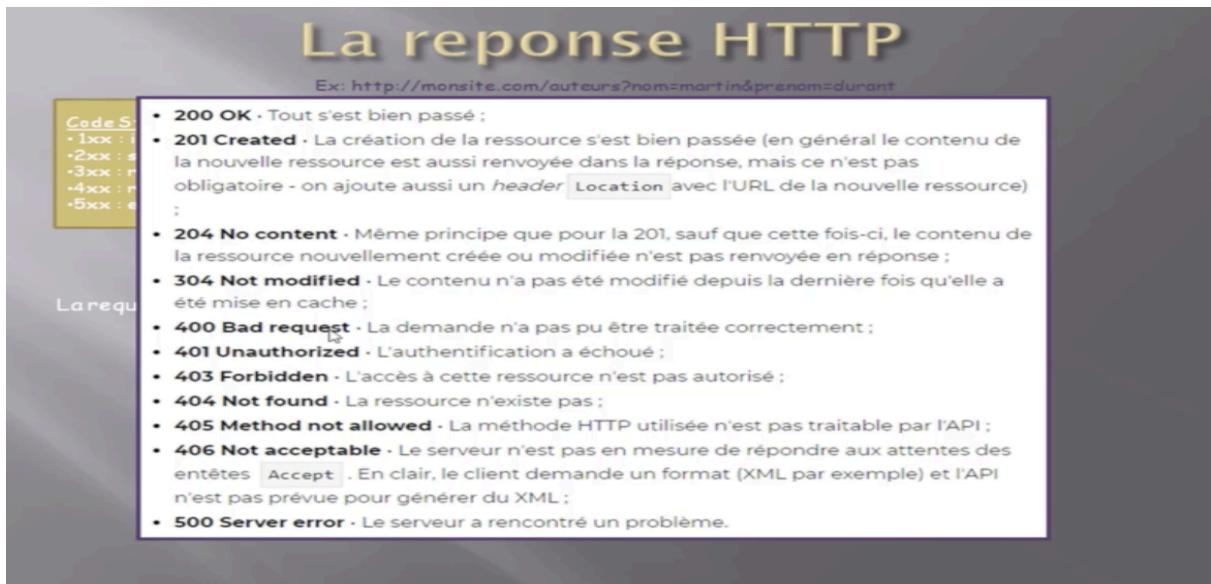
## D.La Requête HTTP



## E.La Réponse HTTP



## F. Les Codes Erreurs



## G. Architecture Rest

### Architecture REST

Representational State Transfert

Les 6 contraintes :

1. **Architecture client/serveur** : c'est le cas du protocole HTTP
2. **Stateless (sans état)** : on ne conserve pas de contexte entre les requêtes donc pas de variable de session par exemple. L'utilisateur doit donc être authentifié à chaque requête !
3. **Cacheable** : on doit pouvoir mettre en cache la ressource pour s'en resservir pour des requêtes similaires consécutives.
4. **Layered system (système scindée en couche)** : le client n'a pas à savoir comment est générée la ressource
5. **Uniform Interface** : contrainte au niveau de la ressource qui doit :
  - Posséder un identifiant unique (combinaison Méthode - URI unique) :  
Ex : ces deux requêtes sont différentes
    - > GET /auteurs (va récupérer la liste des auteurs )
    - > POST /auteurs (va créer un auteur)
  - Avoir une représentation : il faut choisir la manière de formater la réponse et s'y tenir
  - être auto-décrise : il s'agit simplement de préciser le format de la réponse (Json, Xml, CSV ...) grâce au content-type dans le header.

## H. Architecture Rest : Modèle de Maturité de Richardson

### Modèle de maturité de Richardson

Glory of REST

Level 3: Hypermedia Controls

Level 2: HTTP Verbs

Level 1: Resources

Level 0: The Swamp of POX

Permet d'évaluer son API et son degré d'adhésion aux normes REST  
Plus les recommandations sont respectées, plus l'API est dite « RESTful »

### a. Level 1

### Modèle de maturité de Richardson

Glory of REST

Level 3: Hypermedia Controls

Level 2: HTTP Verbs

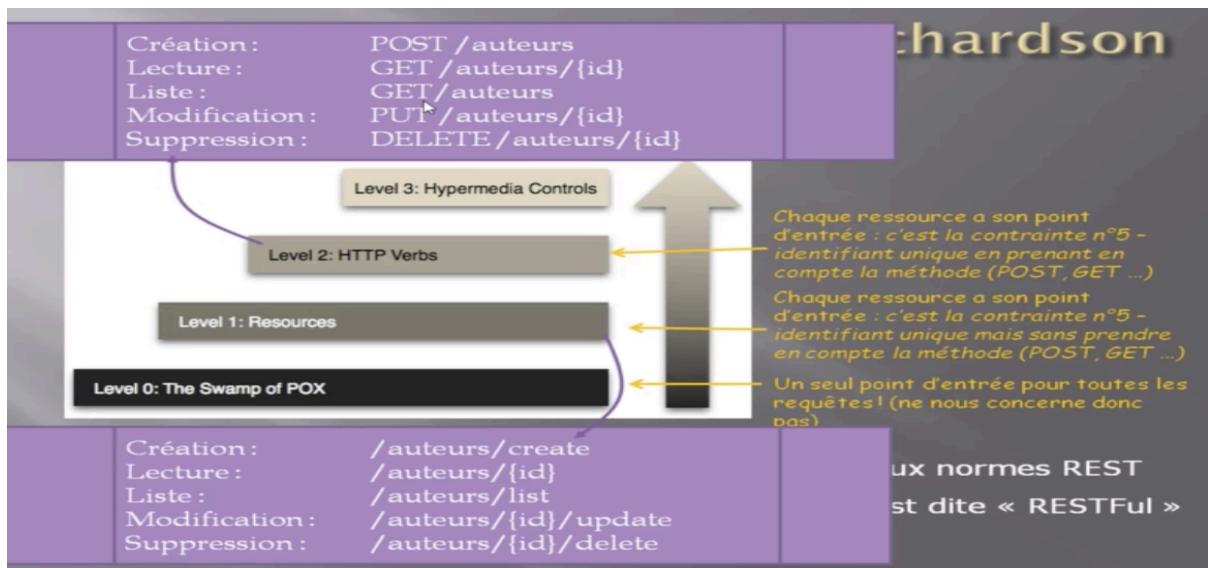
Level 1: Resources

Level 0: The Swamp of POX

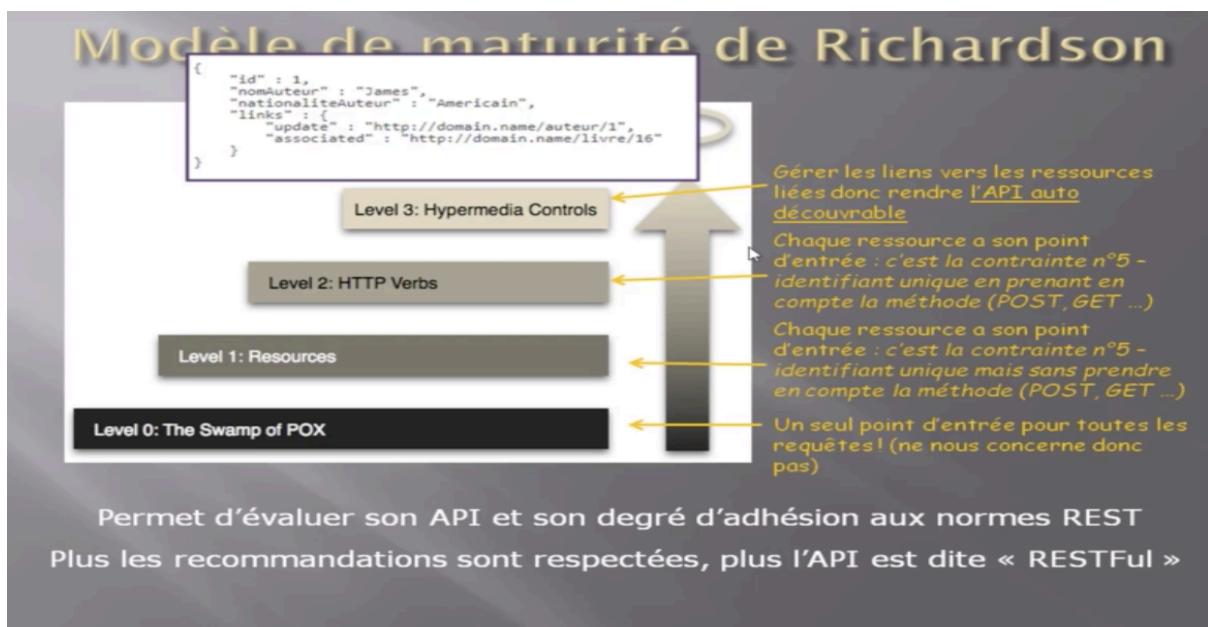
Un seul point d'entrée pour toutes les requêtes! (ne nous concerne donc pas)

Permet d'évaluer son API et son degré d'adhésion aux normes REST  
Plus les recommandations sont respectées, plus l'API est dite « RESTful »

## b. Level 2



## c. Level 3



## I. Utilisation l'API json-server

### a. Installation de l'API json-server

```
npm install --save json-server
npm install --save concurrently
```

### b. Configuration

```
#Dans package.json
"start": "concurrently \"ng serve\" \"json-server --watch db.json\""
#Lancer le Server
npm start
```

### c. PostMan

- i. Installation
- ii. Test Api

### d. Documentation Json Server

<https://github.com/typicode/json-server>

## J. Le Module HttpClientModule

Le module client HttpClient d'angular est introduit depuis sa version 4.3 ,pour permettre de faire des requêtes HTTP comme **GET**, **POST**, **PUT**, **DELETE** au serveur principal.

Cette nouvelle API est disponible en package **@angular/common/http**.

Le client HTTP utilise le RxJs Observables. La réponse du HttpClient est observable.

### a. Étape d'utilisation de HttpClient

- i. Importer le module HttpClient dans le module racine

```
#app.module.ts
```

```
import { HttpClientModule } from '@angular/common/http';
imports: [
 HttpClientModule,
]
```

ii. Injectez le service HttpClient dans le constructeur de votre service.

#### #categorie.service.ts

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

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

iii. Appelez les méthodes

1. **HttpClient.get** : récupérer une ressource sans le modifier(select)
2. **HttpClient.post**: créer une nouvelle ressource(create)
3. **HttpClient.put**: modifier une ressource
4. **HttpClient.delete**: supprimer une ressource

## b. Implémentation de l'Api avec Spring Boot

### i. Catalogue et Détail

#### Pour le Catalogue

```
public class ProductDto {
 private Long id;
 private String libelle;
 private double oldPrice;
 private double newPrice;
 private String image;
 private boolean saled;
 private String categorie;
 private double qteStock;
 private double qteComd;

 public ProductDto(Article article) {
 id=article.getId();
 }
```

```

 libelle=article.getLibelle();
 newPrice=article.getNouveauPrice();
 oldPrice=article.getAncienPrice();
 categorie =article.getCategorie().getLibelle();
 saled=article.isPromo();
 image= article.getPhoto();
 qteStock=article.getQteStock();

 }
}

```

## Pour le Détail

```

public class ProductDetailDto {
 ProductDto product;
 List<ProductDto> products;
}

```

## ii. Dans le Controller Article Catalogue

```

@CrossOrigin(origins = "http://localhost:4200")
@RestController
@RequestMapping("/api/articles")
public class ArticleRestController {
 @Autowired
 ArticleRepository articleRepository;
 @GetMapping("")
 @ResponseBody
 public ResponseEntity<List<ProductDto>>
 getCatalogue(@RequestParam(defaultValue = "0") double
prix,@RequestParam(defaultValue = "0") boolean promo){

 List<ProductDto> articles=
 articleRepository.findAll()
 .stream()
 .map(article -> new
ProductDto(article))
 .collect(Collectors.toList());
 if(articles.isEmpty()){

```

```

 throw new NoDataFoundException();
 }
 return new ResponseEntity<>(articles,
HttpStatus.OK);
}

```

## Détail Catalogue

```

@GetMapping("/{id}")
@ResponseBody
public ResponseEntity<ProductDetailDto>
getDetailProduit(@PathVariable("id")Long id){
 Article article= articleRepository.findById(id)
 .orElseThrow(() -> new
ResourceNotFoundException(String.format("Article avec cet Id
%d n'existe pas", id)));
 ProductDto productDto=new ProductDto(article);
 return new ResponseEntity<>(new
ProductDetailDto(productDto,null), HttpStatus.OK);
}

```

### iii. Gestion des Erreurs

```

public class ResourceNotFoundException extends RuntimeException{
 public ResourceNotFoundException() {
 }
 public ResourceNotFoundException(String message) {
 super(message);
 }
}

```

```

public class NoDataFoundException extends RuntimeException{
 public NoDataFoundException() {
 super("Pas de Donnee");
 }
}

```

```

@ControllerAdvice
@ResponseBody

```

```

public class RestResponseEntityExceptionHandler extends
 ResponseEntityExceptionHandler {

 @ExceptionHandler({ AccessDeniedException.class })
 public ResponseEntity<Object> handleAccessDeniedException(
 Exception ex, WebRequest request) {
 Map<String, Object> body = new LinkedHashMap<>();
 body.put("timestamp", LocalDateTime.now());
 body.put("status", HttpStatus.FORBIDDEN.value());
 body.put("message", "Vous n'avez pas access a cette
ressource");
 return new ResponseEntity<Object>(
 body, new HttpHeaders(), HttpStatus.FORBIDDEN);
 }

 @ExceptionHandler({ ResourceNotFoundException.class })
 public ResponseEntity<Object> ProductDtoNotFoundException(
 Exception ex, WebRequest request) {
 Map<String, Object> body = new LinkedHashMap<>();
 body.put("timestamp", LocalDateTime.now());
 body.put("status", HttpStatus.NOT_FOUND.value());
 body.put("message", ex.getMessage());

 return new ResponseEntity<Object>(
 body, HttpStatus.NOT_FOUND);
 }

 @ExceptionHandler(NoDataFoundException.class)
 public ResponseEntity<Object> handleNoDataFoundException(
 NoDataFoundException ex, WebRequest request) {

 Map<String, Object> body = new LinkedHashMap<>();
 body.put("timestamp", LocalDateTime.now());
 body.put("status", HttpStatus.NOT_FOUND.value());
 body.put("message", ex.getMessage());

 return new ResponseEntity<>(body,
HttpStatus.NOT_FOUND);
 }

 @Override

```

```

protected ResponseEntity<Object>
handleMethodArgumentNotValid(MethodArgumentNotValidException ex, HttpHeaders headers, HttpStatusCode status,
WebRequest request) {
 Map<String, Object> body = new LinkedHashMap<>();
 body.put("timestamp", LocalDate.now());
 body.put("status", status.value());

 List<String> errors = ex.getBindingResult()
 .getFieldErrors()
 .stream()
 .map(x -> x.getDefaultMessage())
 .collect(Collectors.toList());

 body.put("errors", errors);
 return new ResponseEntity<>(body,
HttpStatus.BAD_REQUEST);
}
}

```

## c. Implémentation des Services

```

#categorie.service.ts
getCategories(): Observable<Categories[]>{
 return this.http.get<Categories[]>(`${this.URL_API}/categories`)
}

addCategorie(categorie: Categories): Observable<Categories>{
 return
this.http.post<Categories>(`${this.URL_API}/categories`, categorie)
}

upCategorie(categorie: Categories): Observable<Categories>{
 return
this.http.put<Categories>(`${this.URL_API}/categories/${categorie.id}`, categorie)
}

```

```
deleteCategorie(id:number):Observable<Categories>{
 return this.http.delete<Categories>(`${this.URL_API}/categories/${id}`)
}
```

## #product.service.ts

### #environement.ts

```
export const environment = {
 production: false,
 api:"http://localhost:3000"
};
```

## #product.service.ts

```
constructor(private http:HttpClient){

}

private URL_API:string=environment.api;
getProducts(): Observable<Products[]>{
 return
this.http.get<Products[]>(`${this.URL_API}/products`)
}

getProductById(id:number):Observable<Products>{
 return
this.http.get<Products>(`${this.URL_API}/products/${id}`);
}

getProductsByCategorie(id:number):Observable<Products[]>{
```

```

 return
 this.http.get<Products[]>(`${this.URL_API}/products?categories
_id=${id}`);
}

```

## d. Gestion du Cors

### i. Dans le Controller

```
@CrossOrigin(origins = "http://localhost:4200")
```

### ii. Dans Manière Globale

```

@Bean
public CorsFilter corsFilter() {
 CorsConfiguration corsConfiguration = new
CorsConfiguration();
 corsConfiguration.setAllowCredentials(true);

 corsConfiguration.setAllowedOrigins((Arrays.asList("http://loca
lhost:4200"));
 corsConfiguration.setAllowedHeaders((Arrays.asList("Origin",
"Access-Control-Allow-Origin", "Content-Type",
"Accept", "Authorization", "Origin, Accept",
"X-Requested-With",
"Access-Control-Request-Method",
"Access-Control-Request-Headers"));
 corsConfiguration.setExposedHeaders((Arrays.asList("Origin",
"Content-Type", "Accept", "Authorization",
"Access-Control-Allow-Origin",
"Access-Control-Allow-Origin",
"Access-Control-Allow-Credentials"));
 corsConfiguration.setAllowedMethods((Arrays.asList("GET",
"POST", "PUT", "DELETE", "OPTIONS")));
 UrlBasedCorsConfigurationSource
urlBasedCorsConfigurationSource = new
UrlBasedCorsConfigurationSource();

 urlBasedCorsConfigurationSource.registerCorsConfiguration("/**",
corsConfiguration);

 return new CorsFilter(urlBasedCorsConfigurationSource);
}

```

## e. Gestion des Commandes

### i. Dto

```
@Data
@Getter
@Setter
@Builder
@NoArgsConstructor
@AllArgsConstructor
public class ClientDto {

 private Long id;
 private String nomComplet;
 private String login;
 private String password;
 private String telephone;
 private List<String> roles;

 public ClientDto(Client client) {
 id=client.getId();
 nomComplet= client.getNomComplet();
 telephone= client.getTelephone();
 login=client.getUsername();
 }

}
```

```
public class CommandeDto {
 private Long id;
 private ClientDto client;
 private List<ProductDto> products;
 private double totalHtt;

 private Date date;

 public CommandeDto(Commande commande) {

 id=commande.getId();
 products=commande
 .getLigneCommandes()
 .stream()
 .map(ligneCommande->new ProductDto(ligneCommande))
 .collect(Collectors.toList());
 date =commande.getDateComd();
 totalHtt=commande.getMontant();

 }

 public Commande toEntity(){
 return new Commande(null
 ,date, totalHtt,
```

```

 EtatCommande.Encours
 ,new Client(client.getId()) ,
 null);
 }
}

```

```

public class CommandeClientDto {
 ClientDto client ;
 List<CommandeDto> commandes;
}

```

```

public ProductDto(LigneCommande ligneCommande) {
 id=ligneCommande.getArticle().getId();
 libelle=ligneCommande.getArticle().getLibelle();
 newPrice=ligneCommande.getArticle().getNouveauPrice();
 oldPrice=ligneCommande.getArticle().getAncienPrice();
 categorie
 =ligneCommande.getArticle().getCategorie().getLibelle();
 saled=ligneCommande.getArticle().isPromo();
 image= ligneCommande.getArticle().getPhoto();
 qteStock=ligneCommande.getArticle().getQteStock();
 qteComd=ligneCommande.getQuantite();
 montant=ligneCommande.getQuantite() *
 ligneCommande.getArticle().getNouveauPrice();
}

```

## ii. Dans le Service

```

public interface ICommandeServiceRest {
 CommandeDto saveCommande(CommandeDto commande);
 CommandeClientDto getCommandeByClient(Long id);
}

```

```

@Service
public class CommandeServiceRest implements ICommandeServiceRest{

 @Autowired
 private CommandeRepository commandeRepository;
 @Autowired
 LigneCommandeRepository ligneCommandeRepository;
 @Autowired
 ClientRepository clientRepository;
 @Override
 public CommandeDto saveCommande(CommandeDto commandeDto) {
 Commande commande=commandeDto.toEntity();
 try {
 commandeRepository.save(commande);
 commandeDto.getProducts().stream().forEach(product->{
 ligneCommandeRepository.save(
 new LigneCommande(
 null,product.getNewPrice(),
 (int) product.getQteComd(),
 (int)product.getMontant(),
 commande,
 new Article(product.getId())));
 });
 commandeDto.setId(commande.getId());
 return commandeDto;
 }catch (Exception ex){
 throw new ResourceNotFoundException("Erreur Insertion de
commande");
 }
 }

 @Override
 public CommandeClientDto getCommandeByClient(Long id) {

 Optional<Client> optionalClient=clientRepository.findById(id);
 if(optionalClient.isPresent()){
 List<CommandeDto> commandeDtoList=
commandeRepository.getByClient(optionalClient.get())
 .stream()
 .map(commande -> new CommandeDto(commande))
 .collect(Collectors.toList());
 ;
 return new CommandeClientDto(new
ClientDto(optionalClient.get()),commandeDtoList);
 }
 throw new ResourceNotFoundException(String.format("Ce Client avec
cet Id %d n'existe pas", id));
 }
}

```

### iii. Dans le Controller

```
@RestController
@RequestMapping("/api/commandes")
@CrossOrigin(origins = "http://localhost:4200",methods =
{RequestMethod.GET,RequestMethod.POST})
public class CommandeRestController {
 @Autowired
 ICommandeServiceRest commandeService;
 @PostMapping(path = "",consumes = "application/json", produces =
"application/json")
 @ResponseBody
 public ResponseEntity<CommandeDto> addCommande(@RequestBody
CommandeDto commande) {
 commande= commandeService.saveCommande(commande);
 return new ResponseEntity<>(commande, HttpStatus.CREATED);
 }

 @GetMapping("/client/{id}")
 @ResponseBody
 public ResponseEntity<CommandeClientDto>
getCommandeByClient(@PathVariable("id")Long id){
 CommandeClientDto
commandeClient=commandeService.getCommandeByClient(id);
 return new ResponseEntity<>(commandeClient, HttpStatus.OK);
}
}
```

### iv. Dans le Service ,Angular

```
export class CommandeService {
 commandes: Commande[] = [];

 private api: string = `${environment.API}/commandes`;
 constructor(private http: HttpClient) {}

 addCommande(commande: Commande): Observable<Commande> {
 // this.commandes.push(commande);
 console.log(JSON.stringify(commande));
 return this.http.post<Commande>(this.api, commande);
 }

 commandeByClient(client: Client): Observable<CommandeClient> {
 ///commandes/client/2
 return this.http.get<CommandeClient>(` ${this.api}/client/${client.id}`);
 }
}
```

## V. Dans le Composant ,Angular

```
this.cmdeservice.commandeByClient(client).subscribe(data=>{
 this.panier=data
}) ;
```

## vi. Gestion des États Affichage

### 1. Model DataState

```
export interface DataState<T> {
 state: State;
 value?: T;
 error?: HttpErrorResponse | Error;
}

export enum State{
 Loading ,Load,Error
}
```

## 2. Dans le Composant Catalogue

```
produits$!: Observable<DataState<Produit[]>>;
readonly State = State;
ngOnInit(): void {
 this.produits$ = this.produitService.catalogue().pipe(
 delay(2000),
 switchMap((data) =>
 of(data).pipe(
 map((value) => {
 return { state: State.Load, value };
 })
)
),
 startWith({ state: State.Loading }),
 catchError((err) => {
 alert('ko');
 return of({ state: State.Error, error: err.message });
 })
);
}
```

### 3. Dans la Vue

```
<ng-container *ngIf="produits$ | async as produits$"> </ng-container>
```

#### Etat en Chargement

```
<div
 class="alert alert-success"
 role="alert"
 *ngIf="produits$.state == State.Loading"
>
 Loading
</div>
```

#### Etat Erreur

```
<div
 class="alert alert-danger"
 role="alert"
 *ngIf="produits$.state == State.Error"
>
 {{ produits$.error }}
</div>
```

#### Etat Loaded

```
<section class="py-5" *ngIf="produits$.state == State.Load"> </section>
```

## II. Local Storage

Le **LocalStorage** est un stockage dans le navigateur qui alloue de l'espace pour chaque domaine, il peut stocker jusqu'à 5 Mo de données et toutes les transactions avec le stockage local sont synchronisées.

Le stockage local est une API immuable, ce qui signifie que chaque interaction avec l'objet ne le modifie pas.

L'API n'a que quatre méthodes.

- `setItem(clé : chaîne, données : chaîne | JSON) : void`
- `getItem(clé : chaîne) : chaîne | JSON | null`
- `removeItem(key: string): undefined`
- `clear() : undefined`

NB : Les valeurs stockées dans le `LocalStorage` doivent être stockées en chaîne. Maintenant lorsque l'on veut stocker des objets il faudra les sérialiser.

## vii. Connexion et Inscription

### viii. Connexion

#### 1. Back-end

```
@RestController
@RequestMapping("/api/security")
@CrossOrigin(origins = "http://localhost:4200",methods =
{RequestMethod.GET,RequestMethod.POST})
public class SecuriteRestController {
 @Autowired
 SecurityService service;
 @Autowired
 PasswordEncoder passwordEncoder;

 @PostMapping ("/login")
 @ResponseBody
 public ResponseEntity<ClientDto> login(@RequestBody UserConnect
userConnect) {
 Client user=(Client)
service.getUserByUsername(userConnect.login);
 if(user!=null){
 List<String> roles =user.getRoles()
 .stream()
 .map(appRole -> appRole.getRoleName())
 .collect(Collectors.toList());
 ClientDto clientDto=new ClientDto(user);
 clientDto.setRoles(roles);
 }
 }
}
```

```

 return new ResponseEntity<>(clientDto, HttpStatus.OK);
 }
 throw new ResourceNotFoundException(String.format("Ce Login n'existe pas"));
}

}

class UserConnect{
 public String login;
 public String password;
}

```

## 2. Front-end

### a. Dans le Service

```

login(login: string, password: string): Observable<Client> {
 return this.http
 .post<Client>(`${this.api}/login`, { login: login, password: password })
 ;
}

readonly USER_KEY = 'auth-user';

removeUser(): void {
 this.isConnect$.next(false);
 localStorage.removeItem(this.USER_KEY);
}

saveUser(user: Client): void {
 this.userConnnect = user;
 localStorage.setItem(this.USER_KEY, JSON.stringify(user));
}

getUser(): Client | null {
 const user = localStorage.getItem(this.USER_KEY);
 if (user) {
 this.isConnect$.next(true);
 return JSON.parse(user);
 }
 return null;
}

```

### b. Dans le Composant

```

errorMessage!: string;
isNotLogin: boolean = true;

OnConnexion() {
 this.securiteService
 .login(this.user.login, this.user.password)
}

```

```
.pipe()
.subscribe({
 next: (data) => {
 this.isNotLogin = true;
 this.securiteService.saveUser(data);
 if (this.page == 'panier') {
 this.addCommande(data);
 }
 this.router.navigateByUrl(`commandes/client/${data?.id}`);
 },
 error: (err) => {
 this.isNotLogin = false;
 this.errorMessage = err.error.message;
 },
}) ;
}
```

### III. Les Formulaires

Il existe deux manières d'implémenter les formulaires avec Angular.

- **Template-driven Forms**
- **Reactive Forms**
- **Template-driven Forms**

Ces modèles de formulaire s'appuient sur les directives du modèle pour créer et manipuler le modèle objet sous-jacent. Ils sont utiles pour ajouter un formulaire simple à une application, tel qu'un formulaire d'inscription à une liste de diffusion. Ils sont simples à ajouter à une application, mais ils n'évoluent pas aussi bien que les formulaires réactifs. Si vous avez des

exigences de formulaire très basiques et une logique qui peuvent être gérées uniquement dans le modèle, les formulaires basés sur des modèles pourraient être une bonne solution.

## 1. Utilisation

Cette approche se base essentiellement sur deux directives **ngForm** et **ngModel**.

### a. ngForm

Elle est utilisée dans votre balise `<form #formVariabl="ngForm"></form>` et crée une instance de **FormGroup** pour ce formulaire. Cette directive exporte les valeurs saisies et les états du formulaire vers le composant. Cette directive devient active par défaut sur toutes les balises `<form>` dès que vous importez **FormsModule**.

On distingue les attributs suivants :

- **formVariabl.form.value** qui retourne un objet formé de `{name:value}`. Sachant `name` est la valeur d'un `name` d'un champ du formulaire et `value` la valeur de ce champ.
- **formVariabl.form.errors** qui retourne un tableau erreurs des champs invalides du formulaire .
- **formVariabl.form.valid** qui retourne `true` si le formulaire est valide sinon `false`.

- `formVariabl.form.invalid` qui retourne `true` si le formulaire est invalide sinon `false`.

On distingue les événements suivants :

- `(ng Submit)="formSubmit(formVariabl)":appelle la méthode formSubmit() du composant à la soumission du formulaire en passant comme paramètre la formVariabl.`

## b. **ngModel**

Cette directive est utilisée à l'intérieur de la balise HTML `<input ngModel=""></input>`. Elle est utilisée pour exposer la valeur de cet input ou même d'un autre champ. Donc son rôle est d'enregistrer ce champ en utilisant l'attribut "name". "ngModel" permet aussi de définir comment ce champ est exposé (OneWay, TwoWay).

La directive **NgModel** suit l'état d'un champ, elle vous indique si l'utilisateur a touché le champ ou si la valeur du champ a changé ou si la valeur du champ est devenue invalide.

Angular définit des classes CSS spéciales sur le champ pour refléter l'état, comme indiqué dans le tableau suivant.

État	Classe si vrai	Classe si faux
Le contrôle a été visité.	ng-touched	ng-untouched
La valeur du contrôle a changé.	ng-dirty	ng-pristine
La valeur du contrôle est valide.	ng-valid	ng-invalid

**Projet :**

### 1. Fonctionnalité de Connexion

#app.module.ts et authentication.module.ts

```
imports: [
 FormsModule,
],

```

#authentication.component.ts

```
isSuccessful = false;

isSignUpFailed = false;

errorMessage = '';

user: Users={

 login:'',
 password:''

}

onFormSubmit(){

 this.loginServ.getUserLoginAndPassword(this.user).subscribe(
 data => {

 this.isSuccessful = true;
 this.isSignUpFailed = false;

 },
 err => {

 this.errorMessage = err.error.message;
 this.isSignUpFailed = true;

 }
)
}
```

#authentication.component.html,

#la balise form

```
<form name="form" #loginForm="ngForm"
 (ngSubmit)="loginForm.form.valid && onFormSubmit() "
 novalidate>
```

#les balises inputs

```
<input class="form-control" id="inputEmail"
 required
 type="email"
 name="login"
 [(ngModel)]="user.login"
 #login="ngModel">
/>
```

NB :

**[ngClasse]** : permet d'appliquer un style suivant une condition.

- `[class.my_class] = "condition":`Lorsque la condition est vraie `my_class` est appliquée.
- `[ngClass]="'my_class':condition":` Lorsque la condition est vraie `my_class` est appliquée
- `[ngClass]="{`  
`'my_class1': condition,`  
`'my_class2' : condition1'`  
`}`
- `[ngClass]="step == 'step1' ? 'my_class1' : 'my_class2'"`

**Exemple :**

```
[ngClass]="{
 'is-invalid': password.invalid && (password.dirty || password.touched),
 'is-valid': password.valid && (password.dirty || password.touched)
}"
```

## #Erreur Login

```
<div class="text-danger mt-1"

*ngIf="login.errors && (login.dirty || login.touched)">

Le login obligatoire

Le Login doit contenir au moins

</div>
```

## #Désactivation du Bouton Submit

```
<button type="submit" class="btn btn-primary"
[disabled]="!loginForm.form.valid" >Login

</button>
```

## #Ajout d'un Paramètre de Route sur lien Terminer la commande

```
<div class="d-flex justify-content-between">

 <a class="text-decoration-none"

 [routerLink]=["/login"]

 [queryParams]={{page: 'panier'}}

 >

 Terminer la Commande<i
class="fas fa-long-arrow-alt-right ms-2"></i>

</div>
```

## #Ajout d'un Paramètre sur le lien se connecter du Menu

```
<li class="nav-item">
 <a class="nav-link active" aria-current="page"
 [routerLink]=["/login"]
 [queryParams]={{page: 'login'}}>Se Connecter

```

## #Récupération du Paramètre de Route dans component authentification

```
page:string=""

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

```
ngOnInit(): void {

 this.route.queryParams.subscribe(params => {

 this.page = params['page'];

 });

}
```

## 2. Déconnexion

### #Authentification.service.ts

```
public
isAuthenticated:BehaviorSubject<boolean>=newBehaviorSubject<boolean>(false);
```

### #Authentification.component.ts

```
this.loginServ.isAuthenticated.next(true);
```

### #menu.component.ts

## 3. Crédation de la commande

### a. Crédation du Service Commande

```
ng g service commande
```

### b. Réalisation des Méthodes du Service

## #Ajouter une commande

```
addCommande(panier:Panier) :Observable<Commandes>{

 const commande:Commandes={

 date:new Date().toLocaleString().replace(",","", "").replace(/... /," ") ,

 mntTotal:panier.total ,

 isPayed:false ,

 isLivred:false ,

 client : panier.client ,

 products :panier.products

 }

 return this.http.post<Commandes>(`${URL_API}/commandes` ,commande);
}
```

## 4. Création de la commande

### #Enregistrement de la Commande après Connexion

#### #Athentication.component.ts

```
onFormSubmit(){

 this.loginServ.getUserLoginAndPassword(this.user).subscribe(
 data => {
 this.isSignUpFailed = false;

 if(data){
 this.router.navigate(['home']);
 }
 }
);
}
```

```

 if(this.page=="panier"){
 //Enregistrement de la Commande
 this.panierServ.getCard() .subscribe(
 panier=>{
 //Ajouter le Client
 panier.client=data[0]

 this.cmdeServ.addCommande(panier) .subscribe(
);
 }
)
 //Vider le Panier de Commande
 this.panierServ.clearCard();

 }

 this.router.navigateByUrl("/cmdes-front/client/"+data[0]["id"]);

 }{
 this.isSuccessful = true;
 this.errorMessage="Login ou Mot de Passe Incorrect"
 }

},
err => {
 this.errorMessage = err.error.message;
 this.isSignUpFailed = true;
}
)
}

}

```

## 5. Lister les commandes d'un client

#Dans le component

```

ngOnInit(): void {
 const idClient=this.route.snapshot.params['id']
 this.cmdeServ.getCommandesByClient(idClient) .subscribe(
 (commandes)=>this.commandes=commandes
)
}

```

```
}
```

## #Dans la vue

```
<div *ngFor="let commande of commandes" class="card mb-3">
 <div class="card-body">
 <div class="d-flex justify-content-between">
 <div class="d-flex flex-row align-items-center">

 <div class="ms-3">
 <h5>{{commande.date}}</h5>
 <p class="small mb-0">

 <button class="btn btn-outline-success"
*ngIf="commande.isPayed">
 Payer
 </button>
 <button class=" btn btn-outline-success "
style="margin-left: 5px;" *ngIf="commande.isLivred">
 Livrer
 </button>

 <button class="btn btn-outline-danger"
*ngIf="!commande.isPayed">
 A Payer
 </button>
 <button class=" btn btn-outline-danger "
style="margin-left: 5px;" *ngIf="!commande.isLivred">
 A Livrer
 </button>

 </p>
 </div>
 </div>
 <div class="d-flex flex-row align-items-center">
 <div style="width: 100px;">
 <h6 class="fw-normal mb-0">{{commande.products.length}}
 Produits</h6>
 </div>
 <div style="width: 150px;">
 <h6 class="mb-0 text-warning ">{{commande.mntTotal}}
 CFA</h6>
 </div>
 </div>
 </div>
 </div>
</div>
```

```
 </div>
 <i class="fas fa-trash-alt"></i>
 </div>
</div>
</div>
</div>
```

## 6. Lister les détails d'une commande d'un client

### • Reactive Forms

Les formulaires réactifs fournissent une approche basée sur un modèle pour gérer les champs du formulaire. Cette approche se base principalement sur les classes **FormGroup** et **FormControl**.

#### a. FormGroup

La Classe **FormGroup** permet de créer le modèle de formulaires réactifs sera mappés au champ du formulaire html. Ce mappage des champs facilite l'implémentation, la récupération de valeurs ou encore appliquer des validateurs .

Le **FormGroup** est formé de **FormControl** qui sont mappés à chaque champ du formulaire.

#### b. FormControl

La classe **FormControl** est mappée à chaque champ du formulaire html et permet de contrôler et d'accéder à l'état de ce champ.

## c. Utilisation des Réactifs Forms

### i. Création d'un Objet Form Group

Il existe deux méthodes pour créer un Objet Form Group, soit :

- Par instantiation

Exemple :

```
model= new FormGroup({

 nom: new FormControl(),

 prenom: new FormControl(),

 login: new FormControl(),

 password:new FormControl()

}) ;
```

- En utilisant le service **FormBuilder**

```
constructor(private fb:FormBuilder) {

this.model=this.fb.group({

 nom: new FormControl(),

 prenom: new FormControl(),

 login: new FormControl(),

 password: new FormControl()
}) ;
```

```
 password:new FormControl()

})

}
```

**NB :** Avant d'utiliser les réactifs form, il faudra ajouter le Module **ReactiveFormsModule**

### ii. Les Méthodes et Attributs de FormGroup

1.

#### iii. Les Méthodes de FormControl

FormControl fournit différentes propriétés et méthodes permettant de piloter le "control" :

- **value** : permet d'accéder à la valeur actuellement contenu dans d'un champ.
- **valueChanges** : est un observable permettant d'observer les changements de valeur du champ.
- **reset, setValue et patchValue** : permettent de modifier l'état et la valeur du champ.

### i. Mappage avec la vue

#### 1. Dans la Balise form

```
<form method="post"

[formGroup]="model"

(ngSubmit)="onFormSubmit()"
```

>

## 2. Dans les champs

```
<input class="form-control" id="inputEmail"
 type="email" formControlName="login" />
```

OU

```
<input class="form-control" id="inputEmail"
 type="email" [formControl]="this.model.controls.login" />
```

### ii. Validation

**Les constructeurs des "controls" (*FormControl et FormGroup*) acceptent en second paramètre une liste de fonctions de validation appelées "validators".**

**Les "validators" natifs d'Angular sont regroupés sous forme de méthodes statiques dans la classe **Validators****

```
this.model=this.fb.group({
 nom: [null,[Validators.required,Validators.minLength(6)]],
 prenom: [null,[Validators.required,Validators.minLength(6)]],
 login: [null,[Validators.required,Validators.email]],
 password:[null,[Validators.required,
 Validators.minLength(6),Validators.maxLength(10)]]
})
```

### iii. La Gestion des états

Les "controls" disposent d'une série de propriétés et de méthodes permettant d'en vérifier l'état :

- **valid** : Valeur booléenne indiquant si le "control" est valide. Dans le cas d'un FormGroup ou FormArray, le "control" est valide si les "controls" qui le composent sont tous valides.
- **errors** : "plain object" combinant les erreurs de tous les validateurs. Vaut null si le "control" est valide.
- **touched** : Valeur booléenne positionnée à true dès le déclenchement de l'événement blur (*i.e. l'utilisateur change de "focus"*).
- **pristine**: Valeur booléenne indiquant si le "control" a été modifié.

### iv. Validation du Formulaire Inscription

- **Désactiver le Bouton de soumission**

```
<div class="d-flex align-items-center justify-content-between mt-4 mb-0">

 <button [disabled]="model.invalid"
 type="submit" class="btn btn-primary btn-lg" >S'inscrire</button>
</div>
```

- Gérer les Messages de Validation dans la vue

```
<div class="text-danger mt-1" *ngIf="model.get('nom').invalid &&
(model.get('nom').dirty || model.get('nom').touched)">

Le Nom obligatoire

Le Nom doit contenir au moins

</div>
```

- Activer le Styles

```
<input class="form-control" id="inputEmail" formControlName="nom"

[ngClass]="{

 'is-invalid': model.get('nom').invalid

 && (model.get('nom').dirty || model.get('nom').touched),

 'is-valid': model.get('nom').valid

 && (model.get('nom').dirty || model.get('nom').touched)

}">
```

## IV. Les Pipes

### A. Définition

Les Pipes sont des filtres utilisables directement depuis la vue afin de transformer les valeurs lors du "binding".

Exemple 1: `<div>{{ user.firstName | lowercase }}</div>`

Exemple 2: `<div>{{ user.firstName | slice:0:10 }}</div>`

**NB:** Les pipes peuvent être enchaînées

```
<div>{{ user.firstName | slice:0:10 | lowercase }}</div>
```

## B. Pipe Personnalisé

Pour générer un pipe personnalisé, on utilisera :

```
ng g pipe nomPipe
```

**Exemple 1: Pipe sans Argument**

Nous souhaitons faire ceci :

```
{{'Birane Baila Wane' | author}}
```

Résultat: Ecrit par Sam

```
import {Pipe, PipeTransform} from '@angular/core';
@Pipe({
 name: 'author'
})
export class AuthorPipe implements PipeTransform {
 transform(name: string): string {
 return `Écrit par ${name}`;
 }
}
```

**Exemple 2: Pipe Avec Argument**

Nous souhaitons faire ceci :

```
{{'Birane Baila Wane' | author:'Informaticien'}}
```

Résultat: est un informaticien

```
import {Pipe, PipeTransform} from '@angular/core';
@Pipe({
 name: 'author'
})
export class AuthorPipe implements PipeTransform {
 transform(name: string, fonction:string): string {
 return `${name} - ${fonction}`;
 }
}
```

```
 }
}
```

## Projet:Définir un pipe qui filtre les commandes par date

### 1. moment.js

**Moment.js** est un script utilisé pour analyser, valider, manipuler et afficher des dates et des heures en JavaScript.

#### a. Installation

```
npm install moment --save
```

#### b. Importation dans les composants

```
import * as moment from 'moment';
```

### 2. Générer le pipe

```
ng g pipe dateFilter
```

### 3. Définition du Pipe

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

export class DateFilterPipe implements PipeTransform {
```

```
transform(commandes: Commandes[], start?: Date, end?: Date): Commandes[] {
 }
}
```

#### 4. Filtrer les Commandes du Jour

#date-filter.pipe.ts

```
if(start && !end) return commandes.filter(cmde =>{

 if(cmde.date){

 let myDate=cmde.date.split("-");

 let myDateString=myDate[2] + "-" + myDate[1] + "-" + myDate[0]

 let diff=moment(start).diff(moment(myDateString), "days");

 return diff==0

 }

 return cmde;

});
```

#cmdes-front-list.component.ts

```
filterDate: Date = new Date();
```

#cmdes-front-list.component.html

```
<div *ngFor="let commande of commandes | dateFilter: filterDate"
class="card mb-3">
```

#### 5. Filtrer les Commandes à une date

#cmdes-front-list.component.ts

```
onFilterCmdeByDate(date:any) {

 this.filterDate=date

}
```

### #cmdes-front-list.component.html

```
<p class="mb-0">

 Date: &nbsp &nbsp <input #mydate type="date"
(change)="onFilterCmdeByDate(mydate.value)">

</p>
```

## 6. Filtrer les Commandes entre 2 Dates

### #date-filter.pipe.ts

```
if(start && end)

 return commandes.filter(

 cmde =>{

 if(cmde.date){

 let myDate=cmde.date.split("-");

 let myDateString=myDate[2] + "-" + myDate[1] + "-" + myDate[0]

 return
moment(myDateString).isBetween(moment(start).add(-1,"days"),moment(end).add(1,"days"))

 }else{

 return null;

 }

 }

 }
```

```
)
```

### #cmdes-front-list.component.ts

```
filterDateStart= new Date();

filterDateEnd= new Date();
```

### #cmdes-front-list.component.html

```
<p class="mb-0">

 Du :

 <input type="date"
[ngModel]="filterDateStart" >

 </p>

<p class="mb-2">

 Au:

 <input type="date"
[ngModel]="filterDateEnd" >

 </p>
```

**Projet:**Changer l'état d'une commande

### #cmdes-front-list.component.ts

```

onChangeStatutCmde(statut: any, commande: any) {

 if(statut=="payement") {

 if(commande) {

 commande.isPayed=true;

 }

 }

 if(statut=="livrer") {

 if(commande)

 commande.isLivred=true;

 }

 if(this.commandeById)

 this.cmdeServ.updateCommande(commande as Commandes).subscribe(

 cmde=>this.commandeById=cmde

);

 }

}

```

### #cmdes-front-list.component.html

```

<button class="btn btn-outline-danger" *ngIf="!commande.isPayed "
(click)="onChangeStatutCmde('payement',commande)">

 A Payer

</button>

<button class=" btn btn-outline-danger "
style="margin-left: 5px;" *ngIf="!commande.isLivred "
(click)="onChangeStatutCmde('livrer',commande)">

 A Livrer

```

```
</button>
```

## #commande.service

```
updateCommande (commande: Commandes) : Observable<Commandes>{
 console.log(commande);
 return this.http.put<Commandes>(`${URL_API}/commandes/${commande.id}`, commande);
}
```

# V. Guard

## A. Définition

Dans les applications côté serveur, l'application vérifie toujours les autorisations d'accès à une ressource sur le serveur et renvoie le code erreur 403 si l'utilisateur n'avait pas accès à cette ressource.

Pour avoir le même traitement côté client en angular, on fera appelle au Guard.

**NB:** Les "Guards" ne doivent en aucun cas être considérés comme un mécanisme de sécurité.

## B. Type de Gard

Il existe quatre types de gardes différents :

**CanActivate** : Vérifie si un utilisateur peut visiter un itinéraire.

**CanActivateChild** : Vérifie si un utilisateur peut visiter un itinéraire enfants.

**CanDeactivate:** Vérifie si un utilisateur peut quitter une route.

**Resolve:** Effectue la récupération des données d'itinéraire avant l'activation de l'itinéraire.

**CanLoad:** Vérifie si un utilisateur peut router vers un module chargé paresseusement.

**NB:** Pour un itinéraire donné, nous pouvons implémenter zéro ou n'importe quel nombre de gardes .

Les deux derniers nécessitent des modules de chargement **lazy**.

## 1. CanActivate

Les gardes sont implémentés en tant que services.

Les gardes retourne soit **true** si l'utilisateur peut accéder à un itinéraire, soit **false** s'il ne le peut pas.

Ils peuvent également renvoyer un **Observable** de booléen au cas où le garde ne pourrait pas répondre immédiatement à la question, par exemple, il pourrait avoir besoin d'appeler une API . Angular fera attendre l'utilisateur jusqu'au retour de la garde **true** OU **false**.

**Projet:** Créer un guard d'authentification qui vérifie l'accès aux pages.

### 1. Générer le Gard

## ng g guard authentification

### 2. authentication.guard.ts

```
export class AuthenticationGuard implements CanActivate {

 constructor(private authServ:LoginService,private router:Router) {}

 canActivate(
 route: ActivatedRouteSnapshot,
 state: RouterStateSnapshot): Observable<boolean | UrlTree> | Promise<boolean | UrlTree> | boolean | UrlTree {
 let isAuth=false;

 this.authServ.isAuthenticated.asObservable().subscribe(
 data=>{
 isAuth=data

 if(!data){
 this.router.navigateByUrl("/login")
 }
 }
);
 return isAuth;
 }
}
```

**NB :** Pour aider à déterminer si un garde doit accepter ou refuser l'accès, la fonction de garde peut recevoir certains arguments :

- `component: Component` est le composant lui-même.
- `route: ActivatedRouteSnapshot` est le futur itinéraire qui sera activé si le garde passe. Nous pouvons utiliser sa `params` propriété pour extraire les paramètres de route.
- `state: RouterStateSnapshot`, représente la route qui est active si le garde passe .
- Nous pouvons trouver l' URL vers laquelle nous essayons de naviguer depuis la `url` propriété.

### 3. app.routing.module.ts

```
{ path: 'cmdes-front', loadChildren: () =>
import('./front/cmdes-front/cmdes-front.module').then(m =>
m.CmdesFrontModule), canActivate: [AuthentificationGuard] },
```

## 2. CanActivateChild

Ce Guard pose des autorisations sur les routes enfants.

### 1. Générer le Gard

```
ng g guard access-cmde
```