Pour finir...

Angular, Framework front-end

Table des matières

I. Bonnes pratiques

1. constructor ou ngOnInit ?

Pour répondre à cette question, on va devoir différencier les états du composant crée.

Au départ, Angular instancie le composant en utilisant son constructeur. C'est le **point de départ** de la vie du composant. Rien n'est executé avant le constructeur.

ngOnInit en revanche est executée lorsque le composant est **prêt**, c'est-à-dire qu'il est inclus dans la boucle du **cycle de vie d'Angular**. Le constructeur est executé avant ngOnInit.

Alors comment on choisit ?

Pour faire simple, angular recommande les bonnes pratiques suivantes :

- Le constructeur étant la première chose executée, il est l'endroit idéal pour y inclure tout ce qui va être « injections de dépendances ». Tout ce dont a besoin le composant pour fonctionner, que ça soit un service, un client http ou un Router, on le met dès le départ dans le composant en **l'injectant** dans le constructeur.

- ngOnInit() en revanche va contenir plutôt toute la **logique** du composant. Pas de dépendances à gérer ici, mais plutôt les **actions** à proprement parler. Par exemple lancer un appel à une API, déclencher un chargement, etc…

En ultra résumé : Le constructeur contient les dépendances, ngOnInit contient les fonctionnalités. Le constructeur c'est ce que les composant **est** au départ, ngOnInit est ce que le composant **fait** au départ.

2. signal ou changeDetectorRef?

La méthode signal() permet de déclarer des propriétés **réactives**, ce qui permet de déclencher la détection de ses changements automatiquement. Pour faire très simple, c'est comme si le template **s'abonnait** automatiquement à notre propriété est qu'il écoutait donc ses changements d'état.

Les propriétés déclarées sans signal() en revanche ne permettent pas de déclencher quoi que ce soit au changement d'état. C'est là qu'arrive **ChangeDetectorRef**

Mais quand est-ce qu'on a besoin de provoquer le changement manuellement puisqu'on pourrait le faire automatiquement avec signal() ?

Il existe des cas où **signal()** ne détecte pas les changements. Ce sont des cas peu fréquents, mais qui existent : des Websockets, des librairies Javascript externes, du Javascript natif navigateur… La modification de valeur peut éventuellement s'effectuer en dehors de la **zone** Angular, et donc il ne détectera pas le changement. On le déclenchera alors à la main avec ChangeDetectorRef.

3. Nommage

On en a brièvement parlé, mais les nomenclatures Angular on changé après le passage à Angular 17. Avant, les composants étaient nommés app.**component.ts**, les services api.**service.ts**, les models item.**models.ts**…

Ces nomenclatures ont été abandonnées car Google a décidé que c'était à nous de nommer comme il faut nos outils plutôt que de nous forcer la main. En concordance avec cette décision, Angular nous recommande donc de **nommer** nos classes de manière cohérente.

On va donc préférer nommer nos classes avec un suffixe correspondant à leur nature.

- Mon composant Menu va s'appeler MenuComponent
- Mon modèle « Adresse » va s'appeler AdresseModel
- Mon service API va s'appeler ApiService

Le nom du fichier doit permettre d'en comprendre la fonction **sans l'ouvrir**

Concernant les variables, la même règle doit s'appliquer. À titre personnel je recommande fortement d'utiliser des **verbes** pour les booléens par exemple.

Pour gérer un état, par exemple un temps de chargement, utilisez les verbes **is / has** en anglais. Exemple : Une propriété **isLoaded** qui indique l'état d'un chargement, **hasAccount** pour savoir si un client possède un compte ou non… Vous voyez l'idée.

Pour les méthodes, là aussi je recommande l'usage de **verbes**. Une méthode est nécessairement une **action**, dont rendez cette action **claire** sans avoir à relire votre code à l'intérieur.

Votre méthode :

- Récupère les 3 produits les plus populaires d'une catégorie ? -> **getPopularProducts(quantity, id_category)**

- Calcule et renvoie le montant d'un panier ? -> **getTotalPanier()**

- Envoie les données utilisateurs vers une API pour créer un compte ? -> **createAccount()**

Le build

Le processus de build intégré dans le cllient Angular peut, comme on l'a vu, générer du CSS dans le Javascript, et provoquer des résultats inattendus notamment dans le fichier CSS qui peut se retrouver vide.

Sachez que tout est prévu à cet effet, c'est normal. En effet, il se trouve que ce sont les méthodes optimales de chargement du CSS, et tout ça a à voir avec le LazyLoading.

En effet, les fichiers CSS étant chargés **en entier**, il est tout à fait possible de se retrouver à charger plus de CSS que nécessaire, puisqu'ici Angular ne pourra pas tenir compte du LazyLoading. Par contre, en plaçant ce CSS dans les fichiers JavaScript, Angular garde la main sur le moment où il doit l'**injecter** au navigateur.

Donc pas de souci au niveau du build, ça reste la meilleure façon de procéder.

II. Slider

Un incontournable des sites Web est le traditonnel slider, permettant d'afficher une grosse quantité de contenu tout en engageant l'interaction utilisateur.

On avait déjà vu slick-slider, très adapté à jQuery, mais là on n'a pas jQuery, alors comment on fait ?

Pas de panique, Angular a tout ce qu'il faut, et notamment le package **swiper**.

Premièrement on va l'installer :

```
npm i swiper
```

Ensuite, on va devoir « enregistrer » ce package comme un élément custom a intégrer dans le moteur d'Angular, pour cela on va ouvrir le fichier **main.ts**

main.ts

```
import { register } from 'swiper/element/bundle';

register();
```

Dans le fichier **styles.scss**, on peut également importer les feuilles de styles existantes comme ceci :

styles.scss

```
@use "swiper/css";
@use "swiper/css/navigation";
@use "swiper/css/pagination";
```

Maintenant que ça c'est fait, on va pouvoir se rendre dans le composant dans lequel on veut mettre en place notre slider, et on va importer ceci :

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

Et là :

```
  schemas:  [CUSTOM_ELEMENTS_SCHEMA],
```

Si on veut, on peut aussi définir dans le composant des paramètres responsives, au cas où :

```
  breakpoints = {
    0: {
      slidesPerView: 1
    },
    768: {
      slidesPerView: 2
    },
    1024: {
      slidesPerView: 3
    }
  };
```

Et maintenant on va pouvoir mettre notre slider dans le template :

```
        <swiper-container
            slides-per-view="3"
            pagination="false"
            navigation="true"
            loop="true"
            autoplay-delay="3000"
            class="carousel"
            [breakpoints]="breakpoints"
            >
            <swiper-slide>
                <div>
                    Slide 1
                </div>
            </swiper-slide>
            <swiper-slide>
                <div>
                    Slide 2
                </div>
            </swiper-slide>
            <swiper-slide>
                <div>
                    Slide 3
                </div>
            </swiper-slide>
        </swiper-container>
```

Et voilà ☺

Documentation au cas où : https://swiperjs.com/element

Liste des propriétés CSS à modifier si besoin :

https://swiperjs.com/swiper-api#pagination-css-custom-properties

Exemple de modification des styles :

```
swiper-container {
    --swiper-navigation-color: #{variables.$yellow};

    --swiper-pagination-color: #{variables.$yellow};
    --swiper-pagination-bullet-size: 15px;
    --swiper-pagination-bullet-horizontal-gap: 15px;
    --swiper-pagination-bullet-inactive-color: transparent;
    --swiper-pagination-bullet-inactive-opacity: 1;

    --swiper-pagination-bottom: 0px;
}
```

Liste des attributs que l'on peut mettre dans le HTML :

https://swiperjs.com/swiper-api#parameters