

Navigation et paramètres

Framework front-end

Table des matières

1. Introduction.....	2
2. router.navigate() et ActivatedRoute	2
3. signal()	3

1. Introduction

Même si Angular nous propose de concevoir des SPA, cela ne veut pas dire que l'on ne **navigue** pas à l'intérieur. En effet, même sans temps de chargements, il va forcément arriver un moment dans notre développement où nous aurons besoin de passer des **paramètres** à un autre composant, à déclencher des **changements d'état**, etc...

Pour cela nous allons voir plusieurs façons de procéder, d'une part en utilisant le Router, d'autre part en utilisant **signal()**.

2. router.navigate() et ActivatedRoute

Il existe d'autres moyens de déclencher une navigation que de préparer des balises <a> avec un attribut **routerLink**. Un peu comme en PHP / HTML classique, ou l'on aurait des balises <a> avec un attribut **href**, Angular nous propose d'utiliser **routerLink**. Mais en PHP on peut aussi utiliser

```
header("Location: http://localhost/maNouvellePage.php");
```

Et il existe donc son équivalent avec le Router Angular.

En utilisant un objet de la classe **Router** (qui sera à importer puis à injecter dans notre constructeur, comme d'habitude), on peut utiliser la méthode **Router.navigate()**, comme ceci :

```
constructor(private router: Router) {
  this.router.navigate(['/maPage']);
}
```

Et ce n'est pas tout, on peut évidemment, et c'est tout l'intérêt de cette méthode, lui passer... des **paramètres**. Oui oui, comme on le ferait en PHP avec GET par exemple, Angular nous permet de transmettre des **paramètres** à l'URL que l'on va **Activer**. Comme ceci :

```
this.router.navigate(['/maPage'], {
  queryParams: {
    param1: 50,
    param2: "Bonjour"
  }
});
```

Et dans le composant qui va être appelé sur la route « /maPage », nous n'aurons plus qu'à récupérer ces données en utilisant un objet de la classe **ActivatedRoute**. En effet, Angular va être capable de reconnaître la route qui a été **activée**, et donc de la relire avec tous ses paramètres :

Dans le composant appelé sur **maPage**, on importe la classe **ActivatedRoute** pour l'injecter dans le constructeur, puis on appelle les méthodes : **snapshot** et **queryParamMap**. Snapshot va permettre de retourner une sorte d'état de la route au moment où elle a été appelée, et queryParamMap saura lire dans les données qui ont été transmises dans le tableau **queryParams** :

```

constructor(private activatedRoute: ActivatedRoute) {
  this.param1 = Number(this.activatedRoute.snapshot.queryParamMap.get(param1));
  this.param2 = Number(this.activatedRoute.snapshot.queryParamMap.get(param2));
}

```

Ici, des propriétés qui ont été définies dans un composant donné sont transmises directement via la Route dans un composant receptacle, qui pourra les lire et en faire ce qu'il veut unilatéralement.

Maintenant, comment on fait pour que cette liaison ne soit plus unilatérale ?

3. signal()

La méthode **signal()** (à importer, bien sûr...) est une façon de déclarer des propriétés. Avant, on les déclarait simplement en les typant et en leur assignant éventuellement une valeur, comme ceci :

```

amount: number = 0;
category: number = 0;

```

Ici, on va les déclarer plutôt comme ceci :

```

amount = signal<number>();
category = signal<number>();

```

Quelle différence ?

- Dans le cas N°1, les propriétés sont lues seulement par le composant et doivent être manuellement transvasées dans les composants enfant. C'est pour cela par exemple qu'on utilise **@Input** et les **ChangeDetectorRef** : leur état n'est pas **réactif**, elles ne déclenchent rien lorsqu'elles changent, et doivent être gérées manuellement

- Dans le cas N°2, c'est tout l'inverse. On considère que l'état de la propriété devient **réactif**, et donc lisible dans n'importe quelle situation, peu importe son état. Un changement de valeur d'une propriété déclarée avec signal se **propagera** partout où cette propriété est appelée

Alors pourquoi on s'est embêtés avec **@Input** et **ChangeDetectorRef** ? Parce qu'avant de vous filer les clés du camion il faut apprendre où sont les freins. Tout simplement. Mais dans la réalité, **signal()** remplace **ChangeDetectorRef** et **Input** dans 80% des cas.

Alors voyons comment s'en servir, vous êtes prêts !

Premièrement, comme vous avez pu le constater, **signal** est une méthode. On donne donc à la propriété la **valeur retour** de cette méthode, qui sera typée, et contiendra la valeur à assigner entre parenthèses.

Pour lire cette valeur dans le template, je vais également devoir utiliser une méthode. Qui portera le nom de ma propriété. Là où avant je pouvais appeler mon tableau **questions** comme ceci :

```
@for (q of questions; track q.question; let idx = $index) {
```

Je vais maintenant devoir l'appeler comme ceci :

```
@for (q of questions(); track q.question; let idx = $index) {
```

Enfin, afin que les changements de valeurs puissent bien se répercuter partout, je ne pourrais plus faire de simples assignations de valeurs comme avant avec =

```
this.questions = ['test'];
```

Je vais devoir utiliser la méthode **set()** :

```
this.questions.set(['test']);
```

Ou alors **update()** si ma propriété signalée a déjà une valeur :

```
this.questions.update(['test3'])
```