Intéragir avec le DOM

Ibrahim ALAME

14/02/2023

1 Introduction aux directives

1.1 Les directives

Les directives sont des attributs spéciaux avec le préfixe v-. L'objectif d'une directive est d'appliquer de manière réactive des effets au `DOM` lorsque la valeur de son expression change. Les directives natives sont les suivantes :

- v-text (qui équivaut à)
- v-bind
- v-html
- v-once
- v-show
- v-if
- v-else
- v-else-if
- v-for
- v-on
- v-model
- v-pre
- v-cloak
- v-slot
- v-memo

Il est également possible, comme nous l'étudierons, de créer ses propres directives Vue. Commençons par la directive la plus simple v-text.

1.2 La directive v-text et l'interpolation de texte

La directive v-text permet de mettre à jour le contenu de la propriété textContent de l'élément `HTML`. La directive prend en argument la propriété déclarée sur le composant dont la valeur doit être utilisée. On appelle cela l'interpolation de texte. Par exemple, s'il existe une propriété message sur le composant dans la partie script, alors nous pouvons utiliser la directive de cette manière côté template :

```
<span v-text="message"></span>
```

La propriété textContent de l'élément `span` sera alors remplacé par la valeur de la propriété `message`. Comme lier une propriété est extrêmement courant, il existe une forme raccourcie appelée la notation moustache (de part sa forme), utilisant deux paires d'accolades :

```
<span>{{message}}</span>
```

Est du sucre syntaxique pour :

```
<span v-text="message"></span>
```

Note : en programmation, on appelle sucre syntaxique une syntaxe plus concise ou plus élégante qui est remplacée automatiquement par le code complet et donne le même résultat qu'une expression plus longue.

1.3 Utilisation d'expression JavaScript

Avec Vue.js il est possible d'utiliser toute expression `JavaScript` valide comme argument d'une directive. Voici un exemple avec un ternaire :

```
{{ condition ? 'Oui' : 'Non' }}
```

Autre exemple :

```
{{ message.reverse().toUpperCase() }}
```

Encore un autre :

```
<h1>Bonjour {{ `${prenom.toLowerCase()} ${nom.toUpperCase()}` }}</h1>
```

Nous verrons énormément d'exemples au cours de la formation.

2 Directive v-html

La directive v-html permet de mettre à jour l'attribut innerHTML d'un élément. Le contenu sera inséré comme du HTML simple.

Attention ! Le rendu dynamique de HTML sur votre site Web peut être très dangereux car il peut facilement conduire à des attaques `XSS`. Utilisez uniquement v-html sur un contenu sûr et jamais sur du contenu fourni par l'utilisateur. Voici un exemple :

```
<p>Contenu HTML: <span v-html="varHtml"></span></p>
```

3 Liaison de propriété avec v-bind

La directive v-bind permet d'associer dynamiquement un attribut à une propriété.

3.1 Syntaxe de base

Elle prend en argument l'attribut auquel la propriété est liée. Ainsi par exemple :

```
<a v-bind:href="url"></a>
```

L'attribut href de l'ancre est liée ici à la valeur de la variable url grâce à l'utilisation de la directive v-bind.

3.2 Syntaxe raccourcie

Comme cette directive est extrêmement utilisée, il existe une forme raccourcie :

```
<a :href="url"></a>
```

C'est cette forme que nous utiliserons dans la formation et qui est recommandée.

3.3 Utilisation sur des attributs booléens

Les attributs booléens sont les attributs pour lesquels la simple présence de l'attribut représente la valeur true et leur absence la valeur false. Il y en a 26 différents.

Des exemples sont : disabled, checked, multiple, readonly, selected, hidden etc. Lorsque la directive v-bind est utilisée sur un attribut booléen, le comportement est le suivant : si la variable liée vaut true alors l'attribut sera ajouté par Vue.js automatiquement, si la variable vaut false alors l'attribut sera retiré. Par exemple :

```
<button :disabled="isButtonDisabled">Button</button>
```

Si isButtonDisabled vaut true alors l'attribut disabled sera ajouté par Vue.js sur le DOM, s'il vaut false il sera enlevé.

3.4 Utilisation sur plusieurs attributs

La directive v-bind peut être utilisée pour lier un objet contenant des paires nom-valeur d'attribut. Exemple :

```
<div v-bind="{ id: propriété, 'autre-attribut': autrePropriété }"></div>
```

Ainsi, si nous avons côté script :

```
const objet = {
  id: 'container',
  class: 'container'
}
```

Nous pouvons lier cet objet avec la directive v-bind :

```
<div v-bind="objet"></div>
```

Attention ! Dans ce cas la notation raccourcie n'est pas possible.

3.5 Utilisation avec class et style

Lorsqu'elle est utilisée avec les attributs class ou style, la directive v-bind peut prendre un tableau ou un objet en argument de cette façon :

```
<div :style="{ fontSize: size + 'px' }"></div>
```

Ou :

```
<div :class="[classA, classB]"></div>
```

Nous verrons des exemples dans la formation.

3.6 Utilisation avec un attribut dynamique

Vous pouvez utiliser une liaison dynamique également sur l'attribut avec cette syntaxe :

```
<button :[clé]="valeur"></button>
```

Ici l'attribut peut être modifié dynamiquement en modifiant la valeur de la variable clé.

4 Utilisation de fonction dans les liaisons

Comme nous l'avons vu, il est possible d'utiliser des expressions `JavaScript` dans les templates. Nous pouvons donc invoquer des fonctions. Il est donc courant de voir :

```
<span :title="toTitleCase(titre)">
  {{ formatDate(date) }}
</span>
```

Ici, nous utilisons deux invocations de fonction. L'une avec v-bind et l'autre avec v-text. Nous verrons de nombreux exemples au cours de la formation car c'est très courant.

5 Notions de base TypeScript

5.1 Les types statiques

TypeScript permet de tout taper statiquement : vos variables, vos fonctions (arguments, valeur de retour) et vos classes. Le langage ne vous force pas à taper, ce qui vous laisse totalement libre de le faire ou non et ainsi d'adopter progressivement le typage statique en fonction de votre apprentissage.

5.2 L'inférence de types

TypeScript va détecter automatiquement le type de votre variable en regardant la valeur qui lui est assignée. Nous reviendrons en détails sur cette fonctionnalité très pratique, mais elle permet sans aucune ligne supplémentaire d'avoir ce résultat :

```
const obj = { width: 1, height: 2 };
const aire = obj.width * obj.heigth;
// Property 'heigth' does not exist on type '{ width: number; height: number; }'. Did you mean // '
```

Ici TypeScript a détecté qu'aucune propriété `heigth` n'existait sur objet vous invite à vérifier que vous n'avez pas fait une erreur. Magique, non ?

5.3 Les types

Comme son nom l'indique et comme nous l'avons vu TypeScript est avant tout un langage pour taper. Il est possible de taper absolument tout comme nous le verrons : que ce soit les variables mais aussi les retours de fonction ou des situations encore plus complexes.

Typer au début peut sembler fastidieux : il faut ajouter un peu de code à chaque fois. Mais les bénéfices se remarquent très vite et dans cet ordre souvent :

1. Vous aurez moins de bugs lors de l'exécution car beaucoup auront été signalés lors de la compilation : typo dans les noms, erreurs dans les conditions, dans les arguments passés, dans les retours de fonction etc.
2. VS Code vous fournira avec un survol de la souris plein d'informations sur vos fonctions : quels arguments sont attendus etc, et vous gagnerez un temps fou grâce à l'autocomplétion (car TypeScript aura toutes les propriétés sur vos objets etc).
3. Lorsque vous serez sur un projet à plusieurs, vous pourrez lire et comprendre beaucoup plus vite le code des autres développeurs car vous verrez facilement ce que prend et ce que retourne chaque fonction, quels sont les types acceptés par chaque variable etc.

Une fois les premiers jours, où cela semblera être une surcouche embêtante, passée, vous ne pourrez plus faire sans !

5.4 Les types primitifs

Nous allons commencer par les types les plus basiques : les primitifs !

Pour préciser un type en TypeScript nous utilisons la plupart du temps (nous verrons en détails les autres cas) cette syntaxe :

```
variable: type;
```

Il est possible de simplement taper une variable sans rien assigner ou de taper et assigner directement :

```
variable: type;
variable: type = x;
```

5.4.1 Les chaînes de caractères :string

Les chaînes de caractères en TypeScript comprennent exactement comme en `JavaScript` les guillemets simples ou doubles et les accents graves (`backticks`) :

```
let prenom: string = "Jean";
prenom = 'Paul';
prenom = `Patrick`;
```

5.4.2 Les nombres :number

Tous les nombres valides en `JavaScript` le sont en TypeScript :

```
let decimal: number = 42;
let flottant: number = 42.22;
let hexadecimal: number = 0x2A;
let binaire: number = 0b101010;
let octal: number = 0o52;
```

5.4.3 Les booléens :boolean

Le type `boolean` n'accepte que `true` ou `false`:

```
let actif: boolean;
actif = true;
actif = false;
```

5.5 Le type any

Il est possible de ne pas connaître le type de variables. Ces valeurs peuvent provenir d'un contenu dynamique, par exemple de l'utilisateur ou d'une bibliothèque tierce.

Il est également possible que vous souhaitiez taper plus tard votre fonction ou variable ou que vous soyez en train de convertir un fichier `JavaScript` en TypeScript et que vous mettiez les types peu à peu.

Dans ces cas, nous souhaitons désactiver la vérification de type. Pour ce faire, nous les étiquetons avec le type any.

```
let pasSur: any = 4;
pasSur = 'en fait une string';
```

Lorsque vous commencez en TypeScript vous aurez tendance à beaucoup trop utiliser any. A chaque fois que vous ne saurez pas trop le type vous mettrez any. Au début ce n'est pas très grave mais il faut vous forcer à faire l'effort de tout bien typer pour garder les bénéfices de l'usage de TypeScript. Nous vous recommandons de ne quasiment jamais utiliser any sauf dans les cas vus ci-dessus.

5.6 Le type object

Le type objet permet simplement de représenter le type non primitif : tout ce qui n'est pas `number, string, boolean, bigint, symbol, null` ou `undefined`. Cela peut donc être un tableau, un objet littéral, etc.

```
let monObj: object;
monObj = {};
```

Notez que object et sont deux manières de typer un objet, nous pouvons écrire :

```
let monObj: {};
monObj = {};
```

5.7 Le type array

Il y a deux syntaxes en `TypeScript` pour taper les tableaux . En `TypeScript` un tableau est un tableau `JavaScript` contenant un nombre indéfini d'éléments . Première possibilité :

```
type[];
```

Où type est le type des éléments contenus dans un tableau. Il est possible d'accepter par exemple que les nombres :

```
let liste: number[] = [1, 2, 3];
```

Tous les types :

```
let liste: any[] = [1, true, {}];
```

5.8 Qu'est-ce qu'une interface ?

Une interface est un contrat qui définit la forme qui doit prendre un objet `JavaScript` (objets littéraires, classes et fonctions) . Il est important de noter qu'une interface n'est pas compilée : aucun code `JavaScript` ne sera créé à partir de l'interface. Elle sert uniquement pour l' IDE, c'est-à-dire pour VS Code, (auto-complétions et erreurs de types) et lors de la compilation.

5.8.1 Notre première interface

Prenons un premier exemple avec un objet :

```
interface User {
    prenom: string;
}
```

Ici, notre objet User doit obligatoirement avoir une propriété prenom contenant une chaîne de caractères. Ensuite nous pouvons utiliser l' interface pour taper par exemple un paramètre d'une fonction :

```
function printUser(user: User) {
    console.log(user.prenom);
}
```

Ici, la fonction printUser doit recevoir en argument un objet respectant l' interface User . Par exemple :

```
let paul = {prenom: 'Paul'};
printUser(paul);
```

Attention ! Lorsque vous tapez le paramètre de cette manière `user: User` , cela revient à dire que le paramètre doit au moins avoir le ou les propriétés de l'interface User et donc avoir une propriété `prenom`.

En revanche, lorsque vous tapez un objet avec une interface, il faut que l'objet respecte exactement l'interface . C'est-à-dire qu'il doit avoir exactement les propriétés définies, ni plus ni moins. Voici un exemple pour bien comprendre :

```
interface User {
    prenom: string;
}

function printUser(user: User) {
    console.log(user.prenom);
}

const paul = {prenom: 'Paul', nom: 'Dupont'};

printUser(paul); // Cela fonctionne car paul a au moins une propriété nom

const jean: User = {prenom: 'Jean', nom: 'Duprey'}; // Erreur
```

Dans le cas de l'objet Jean, `TypeScript` nous signalons l'erreur suivante : **Type ' prenom: string; nom: string; ' is not assignable to type 'User'. Object literal may only specify known properties, and 'nom' does not exist in type 'User'**.

Autrement dit, il se plaint car Jean a une propriété `nom` qui ne respecte pas le contrat imposé par l' interface User.

5.8.2 Les propriétés optionnelles avec?

Il est possible de définir des propriétés optionnelles comme nous l'avons vu pour les arguments de fonction en utilisant ?. Ces propriétés sont également utilisables dans les interfaces.

```
interface User {
  prenom: string;
  nom?: string;
}
```

Ici, nous établissons comme contrat que les objets qui respecteront l' interface User auront obligatoirement une propriété `prenom` contenant une chaîne de caractères, et une propriété optionnelle `nom`, qui si elle existe, devra également contenir une chaîne de caractères. En outre, les propriétés optionnelles permettent l'autocomplétion et l'autocorrection.

5.9 Les unions de type

Les unions de type permettent de combiner deux ou plusieurs types et s'utilisent avec — . Les unions s'utilisent dans toutes les situations.

5.9.1 Unions de type et variables

Prenons un premier exemple :

```
let uneVar: string | number;}
```

Ici, nous effectuons une union entre les types de chaîne de caractères et de nombres. La variable pourra ainsi recevoir l'un des deux types.

5.9.2 Unions de type et fonctions

Vous pouvez également taper les paramètres des fonctions en utilisant des unions :

```
function ajouter(a: string | number, b: string | number) {
  return Number(a) + Number(b);
}
```

Ici par exemple, notre fonction peut ajouter des nombres et des chaînes de caractères. Nous voulons donc utiliser des unions pour les paramètres de la fonction. Vous pouvez également taper les retours de fonctions en utilisant des unions :

```
function ajouter(isAdmin: boolean): string | void {
  if (isAdmin) {
    return 'secret';
```

```
  } else {
    return;
  }
}
```

6 Les événements natifs

6.1 La gestion des événements

L'objectif de la gestion des événements est de pouvoir réagir à un évènement. Les événements HTML sont des événements qui se produisent suite aux interactions de l'utilisateur avec son clavier, sa souris ou alors à la modification d'éléments du DOM:

- *modification de l'objet window* : par exemple la page est fini d'être chargé
- *modification d'un formulaire* : par exemple lorsqu'un champion est invalide
- *interactions clavier ou souris* : par exemple une pression de toucher
- *événements relatifs à un média (vidéo, audio)* : par exemple lorsqu'une vidéo finie
- *événements relatifs au glisser-déposer* : par exemple lorsqu'un élément a été glissé

Lorsqu'il Vue.js est appliqué sur une page, HTML il peut réagir à ces événements.

6.1.1 L'approche de Vue.js

Dans Vue.js les écouteurs sont déclarés dans le HTML. Pourquoi cette approche ?

1. permet de voir directement dans le HTML où se situent les écouteurs d'événement et à quel élément ils sont attachés.
2. permet de ne pas avoir à mettre des sélecteurs et des écouteurs latéraux JavaScript offrant ainsi la possibilité de n'avoir que la logique dans le JavaScript et de pouvoir mieux tester.
3. Lorsqu'un composant est détruit, tous les écouteurs sont également retirés automatiquement.

6.1.2 v-on: écouter des événements du DOM

La directive v-on permet d'écouter des événements et d'attacher un gestionnaire d'événement à un évènement . Le type d'événement est déterminé par l'argument qui est passé à v-on, ainsi par exemple : `v-on:keyup` permet d'écouter l'événement du relâchement d'une touche du clavier.

6.1.3 Le raccourci@

v-on peut être remplacé par un raccourci : @.

```
<button @click="direBonjour">Dire bonjour</button>
```

Ici un clic sur le bouton va déclencher la fonction `direBonjour()` du composant. Autrement dit, l'événement click va exécuter le gestionnaire d'événement `direBonjour()`.

6.1.4 L'objet event reçu

Le gestionnaire d'événement reçoit automatiquement l'événement du DOM en argument . Par exemple :

```
<template>
  <button @click="direBonjour">Dire bonjour</button>
</template>

<script setup lang="ts">
function direBonjour(event: MouseEvent)
{
    console.log(event);
}
</script>
```

Notez que nous typons l'événement comme un MouseEvent pour pouvoir accéder à toutes les propriétés et méthodes par autocomplétion.

6.1.5 Passer un argument à un gestionnaire d'événement

Notez que vous pouvez passer un argument à un gestionnaire d'événement, dans ce cas l'événement sera disponible en deuxième argument . Il faut passer $event en deuxième argument au gestionnaire d'événement pour pouvoir y accéder dans la fonction :

```
<template>
  <button @click="direBonjour('Salut', $event)">Dire bonjour</button>
</template>

<script setup lang="ts">
function direBonjour(arg: string, event: MouseEvent)
{
    console.log(arg);
    console.log(event);
}
</script>
```

6.2 L'assertion TypeScript avec as

Dans certains cas, vous aurez besoin de signaler TypeScript que vous êtes sûr d'un type et qu'il n'a pas à s'en préoccuper. Autrement dit, vous lui désignez un type et vous l'obligez à le respecter, c'est ce qu'on appelle l'assertion. L'assertion doit être manipulée avec prudence car vous perdez le bénéfice du contrôle de type avant l'exécution. Pour l'assertion il faut utiliser :

```
as type
```

Par exemple :

```
let maVar: any = "Une chaîne";

let longueur: number = (maVar as string).length;
```

Ici nous avons forçons TypeScript à reconnaître `maVar` comme étant une chaîne de caractères pour pouvoir utiliser l'autocomplétion et la propriété `length`. Le plus souvent nous souhaitons utiliser une interface HTML comme dans la vidéo, afin d'obtenir les bonnes propriétés et les bonnes méthodes. En effet, TypeScriptne peut pas connaître l'élément du DOM HTML et il faut donc le lui préciser. Vous verrez donc asla plupart du temps dans ce cas.

7 Les modificateurs d'événements

7.1 Les modificateurs d'événements (event modifiers)

Les modificateurs d'événements permettent de modifier des événements avant qu'ils soient gérés dans les gestionnaires d'événements. Ils sont très pratiques pour deux raisons : premièrement, ils permettent aux gestionnaires de ne s'occuper que de la logique et deuxièmement, de raccourcir et simplifier la syntaxe. Vue.js proposer huit modificateurs d'évènements :

- .stop: `verserevent.stopPropagation()`
- .prevent `event.preventDefault()`
- .self pour ne déclencher l'événement que s'il est envoyé de cet élément
- .once pour ne déclencher l'événement qu'une seule fois
- .passive pour ajouter `le mode passif`
- .capture pour ajouter l'écouteur d'événement en mode `capture`

Il s'utilise de cette manière :

```
<button type="submit" @click.prevent="calculer">Calculer</button>
```

7.2 Touches de contrôle (key modifiers)

Vue.js facilite grandement la gestion des événements claviers et souris, il fournit ainsi de nombreuses méthodes afin de réagir à ces événements. Par exemple pour réagir à la touche entrée :

```
<input @keyup.13="submit"> // en utilisant le code de la touche
<input @keyup.enter="submit"> // en utilisant l'alias proposé par Vue
```

La liste des alias proposés par Vue pour le clavier est la suivante :

- .enter
- .tab
- .delete
- .esc
- .space
- .up
- .down
- .left
- .right
- .ctrl
- .alt
- .shift
- .meta((équivaut à cmd ou touche windows))

Pour la souris :

- .left
- .right
- .middle

Exemple exécutable Vous pouvez également utiliser directement ce code exécutable. N'hésitez pas à l'ouvrir dans un nouvel onglet :