

1. Présentation de la ressource

La ressource R6.A.05 porte sur le développement avancé et présente un *framework* de programmation pour les applications Web réactives de type *Single Page Application* (SPA).

Il s'agit d'un outil permettant de créer des pages Web dynamiques, modulaires et interactives, où les mises à jour se font en temps réel sans rechargement complet de la page.

Plusieurs frameworks permettent de mettre en oeuvre cette technologie (React, Angular, VueJS). Ce cours présente **Vue.js**, qui a été choisi pour sa facilité d'apprentissage et ses performances.

Le volume horaire de ce cours est le suivant :

- 8h de TD
- 8h de TP

La ressource sera évaluée lors d'un TD noté de 2h.

2. Présentation générale de Vue.js

2.1. Qu'est-ce que Vue.js ?

Vue.js est un *framework JavaScript* créé par Evan You en 2014 et désormais maintenu activement par une vaste communauté open source. Il permet de construire facilement des interfaces utilisateur dynamiques et réactives. Contrairement à des frameworks plus lourds, Vue a été conçu pour être simple à prendre en main, tout en restant suffisamment puissant pour développer des applications web complexes.

2.2. Principes clés

- **Réactivité** : le contenu affiché s'adapte automatiquement aux changements de données, sans rechargement de la page.
- **Composants** : chaque partie de l'interface (bouton, formulaire, carte, etc.) est isolée dans un *composant* réutilisable.
- **Approche déclarative** : on décrit l'état de la page en fonction des données plutôt que de manipuler directement le DOM.
- **Intégration progressive** : Vue peut être utilisé sur une simple page HTML ou au cœur d'une application complète (SPA).

2.3. Architecture simplifiée

Vue.js s'appuie sur le **pattern MVVM** (Model–View–ViewModel), qui permet de séparer clairement les données, la présentation et la logique de l'application.

- **Le Modèle (Model)** : contient les *données réactives* de l'application.
- **La Vue (View)** : correspond à la partie visible de l'interface, décrite dans les *templates* HTML enrichis avec la syntaxe Vue : `{{...}}` et attributs `v-*`.
- **Le ViewModel** : fait le lien entre les deux, en assurant la *liaison automatique* entre les données et la vue (data binding) ainsi que la réactivité de l'interface.

Dans Vue.js, ces trois éléments sont regroupés au sein de **composants**, généralement définis dans des fichiers `.vue`, qui intègrent la présentation en HTML (partie `<template>`), la logique en JavaScript ou TypeScript (partie `<script>`) et le CSS (partie `<style>`) d'une même unité fonctionnelle (nous aborderons ces aspects en détails lors du TD2).

Dans ce cours, nous commencerons par utiliser Vue.js à minima pour découvrir ses concepts, plus tard, nous créerons des applications plus conséquentes basées entièrement sur Vue.js dans le cadre Node.js.

3. Variables réactives

Passons maintenant à la pratique, en commençant à utiliser Vue.js dans un petit exemple.

Vous allez programmer une application très simple qui contient un curseur variant entre 0 et 100, et qui affiche sa valeur courante. Dans un premier exercice, vous appliquerez vos connaissances antérieures (développement Web) pour faire fonctionner ce logiciel, et dans un second temps, vous remplacerez certains aspects par Vue.

- ☛ Créez un dossier `R6A05-DeveloppementAvancé` (ou équivalent) dans votre environnement. Placez-vous dans ce dossier.
- ☛ Créez un sous-dossier `TD1` dans ce dossier. Placez-vous dans ce sous-dossier.
- ☛ Récupérez le code source lié au TD sur Moodle et copiez le dans ce dossier.
- ☛ Nous vous conseillons de travailler avec **Visual Studio Code** et une extension comme **Live Server** pour servir les pages HTML que nous allons créer dans ce TD.

3.1. Curseur version HTML+JS

- ☛ Partons du fichier `curseur_html.html`.

Le but de l'exercice est de faire afficher en temps réel la valeur du curseur dans le label de droite.

Les fonctions à écrire sont dans une balise `<script type="module">` qui change la visibilité des variables et fonctions. Il ne suffit pas de définir une fonction pour qu'elle soit globale, comme avec `<script type="text/javascript">` (déprécié). Il faut aussi attacher cette fonction à la fenêtre :
`window.fct = fct.`

Aide :

- Utiliser les attributs d'événements `oninput` et `onclick` sur les balises `<input>` et `<button>`.
 - `const elt = document.getElementById('id')` permet d'accéder aux éléments du DOM à partir de leur identifiant.
 - La valeur d'un `<input>` se récupère à l'aide de `elt.value`.
 - Il est possible de modifier le contenu d'un `<label>` avec `elt.textContent`.
- ☛ Vous avez 15 minutes pour terminer cette question.

3.2. Curseur version VueJS

- ☛ Le fichier `curseur_vue.html` est la version terminée du logiciel fait avec Vue.

Comparons les deux sources, `curseur_html.html` et `curseur_vue.html`. Dans le premier, la partie script contient essentiellement deux fonctions. Dans le second, il semble y avoir plus de choses. Voici quelques explications sur le contenu de la balise `<script>` :

- Il faut d'abord importer les fonctions dont on a besoin. Ces fonctions, `createApp` et `ref` sont expliquées plus bas. Elles font partie de Vue.js qu'on télécharge à l'ouverture de la page. Ce fichier ne fait que 160Ko et il contient tout Vue.js.
- Il y a ensuite la définition de la constante `valeur`. C'est une variable réactive, définie en `const` parce que c'est un proxy dont on peut modifier la valeur (l'attribut `value`) mais pas le proxy lui-même. Pour simplifier, cette variable va être utilisée dans l'interface à plusieurs endroits. Chaque fois qu'elle sera modifiée, tous les endroits où elle est présente seront mis à jour.
- On trouve ensuite un objet nommé `configuration`. Il est très bizarre au premier abord. Normalement, en JS, on définit un objet par `const obj = { prop1: val1, prop2: val2, ... }`. Ici, c'est plutôt quelque chose comme `const obj = { mafonction() { ... return val } }`, sauf que la fonction s'appelle `setup`.

Pour comprendre, il faut rappeler une possibilité pour définir les propriétés d'un objet en JS, version ES2015, voir [Initialisateur d'objet](#) [↗](#). Quand ces propriétés ont le même nom que leurs valeurs. Au lieu d'écrire `const obj = { val1: val1, val2: val2, ... }`, on peut simplifier en `const obj = { val1, val2, ... }`.

Ainsi, ce qu'on trouve dans le code est équivalent à `const configuration = { setup: function setup() { return liste } }`. C'est donc un objet qui possède une propriété appelée `setup` qui est une fonction sans paramètre, et le rôle de cette fonction est de retourner la liste des variables réactives définies dans l'application, sous forme d'un objet, lui aussi simplifié. De fait, le résultat retourné par `setup` est équivalent à `{valeur: valeur}`.

- En quatrième, on voit `createApp(configuration).mount("body")`. La fonction `createApp` crée un objet géré par Vue qui contrôle l'application. Cet objet est associé par `mount` à un élément HTML de la page. Ici, c'est la page entière, mais on peut mettre n'importe quel sélecteur CSS, par exemple `"#app"` pour associer l'application à un `<div id="app">`.

Voici les points à regarder dans les balises HTML :

- D'abord, la balise `<body>` est prise en charge par Vue. C'est sur elle qu'est « montée » l'application. Certains de ses éléments emploient des variables réactives et seront donc dynamiques. Les variables réactives qui seraient en dehors de l'élément monté ne seraient pas prises en charge.
- Regardez ce qui se trouve dans le `<input type="range">`. Il n'y a plus aucun écouteur, uniquement un attribut `v-model` dont la valeur est la variable réactive. `v-model` signifie que la variable est associée de manière bidirectionnelle à la valeur du `<input>`. Ça veut dire que cet élément peut modifier la valeur et aussi, que si on change la valeur par ailleurs, ça la modifie dans le `<input>` sans qu'on ait besoin de rien programmer.
- Regardez ce qui se trouve dans le `<label>`. Il y a une syntaxe avec les "doubles moustaches" pour insérer la valeur d'une variable réactive.
- Regardez l'écouteur du bouton, `v-on:click="valeur=0"`. La syntaxe `v-on:click` est celle d'un écouteur géré par Vue (on peut aussi écrire de manière abrégée `@click`). On peut mettre soit le nom d'une fonction ou d'une méthode sur un objet (*method handler*), soit une instruction JS (*inline handler*). Ici, on affecte `valeur`, ce qui peut paraître étrange parce qu'on l'avait déclarée en tant que `const`. On peut se faire la même remarque avec le `<input>` et son `v-model`. Comment peuvent-ils modifier la valeur ? Dans un template (des éléments HTML), les variables réactives sont vues en tant que telles et non pas en tant que proxies. En clair, ces variables sont vraiment variables et utilisables directement. Donc on peut directement affecter `valeur`.
- Les éléments du DOM n'ont plus d'identifiant (attribut `id`). Ces identifiants sont une source de bugs dans un grand logiciel modulaire écrit à la mode HTML+JS comme le premier exercice. Ça n'arrive pas quand on travaille avec Vue, parce qu'il n'y a pas besoin d'identifiant. Certes, il y a toujours besoin d'identifiants pour lier un `<label>` à son `<input>`, mais ce n'est pas pour attacher un écouteur ou placer une valeur dans un élément.

Constatez qu'il n'y a plus grand chose à programmer. Les lignes qui mettent en place Vue, importation, configuration et lancement, sont toujours les mêmes d'une application à l'autre. Seules les variables réactives sont différentes.

Retenir qu'on définit une variable réactive simple par `const nom = ref(valeur initiale)`.

La variable définie par `ref` est un proxy non modifiable, déclaré en `const`, mais qui possède un champ appelé `value` qui contient la vraie valeur de la variable. Ainsi, la variable elle-même reste la même. Les éléments avec moustache et les attributs `v-*` peuvent garder des pointeurs dessus. Quand on réaffecte la variable, c'est uniquement son champ `value` qui change.

3.3. Définition d'une variable réactive avec `reactive()`

Il existe une autre manière de définir une variable réactive pour des objets et les tableaux. On ne peut pas l'utiliser sur une valeur simple comme un nombre ou une chaîne.

- ☛ Dupliquez `curseur_vue.html` sous le nom `curseur_vue_reactive.html`.
- ☛ Remplacez-y l'importation de `ref` par `reactive`.
- ☛ Remplacez `const valeur = ref(0)` par `const model = reactive({valeur: 0})`. Cette variable représente l'état de l'application.
- ☛ Remplacez `valeur` par `model` dans l'objet configuration.
- ☛ Dans les balises HTML, remplacez partout `valeur` par `model.valeur`.

La fonction `reactive(object)` ajoute de la réactivité à l'objet. Chacune de ses propriétés devient réactive. C'est approprié par exemple pour définir le modèle des données en une seule fois. Ici, on crée un objet contenant une seule propriété, `valeur`. Ensuite, l'emploi est un peu moins confortable qu'avec `ref`. Il faut mettre `model.valeur` partout au lieu de seulement `valeur`.

Dernière remarque, l'exemple précédent qui utilise `reactive` pour gérer une seule propriété est, dans ce cas, totalement disproportionné. On utilise `reactive` sur des tableaux ou des objets ayant plusieurs propriétés. On verra cela dans de prochains TD.

4. Propriétés calculées

Nous allons maintenant étudier d'autres mécanismes de Vue pour gérer des situations plus complexes. Nous allons découvrir la notion de variable réactive calculée, c'est-à-dire que sa valeur est définie par une fonction d'autres variables.

4.1. Convertisseur version HTML+JS

- ☛ Partons du fichier `convertisseur_html.html`.

Ce fichier est une esquisse de ce qu'il faut réaliser. Il y a un `<input>` pour saisir une valeur en inch. L'idée est que saisir un nombre dans ce champ fait afficher sa valeur convertie dans le `<label>`. Pour cela, vous devez rajouter un écouteur. On vous propose de réagir sur l'événement `onchange`, donc quand on valide une valeur avec la touche entrée.

- ☛ Terminez la programmation et testez la page. N'y consacrez pas plus de 15 minutes.

4.2. Convertisseur version VueJS

Passons à la version Vue.

☛ Partons du fichier `convertisseur_vue.html`.

Comme dans le précédent projet, il y a une variable réactive définie par `ref` et une propriété calculée définie par `computed`. Une propriété calculée est une valeur dérivée automatiquement mise à jour lorsqu'une ou plusieurs données réactives dont elle dépend changent.

On les définit par `const prop = computed(() => valeur)` si c'est simple ou `const prop = computed(() => { tout un corps de fonction; return valeur })` si c'est plus compliqué. La fonction `computed` doit être importée au début du script.

Il faut aussi lister les propriétés calculées dans la configuration de l'application.

☛ Testez cette application.

Vous allez constater que quand on entre une valeur en inch, elle est bien convertie en cm, et même à chaque chiffre tapé.

5. Contenu dynamique

En s'intéresse maintenant à rendre dynamique le contenu HTML.

☛ Rouvrir le fichier `curseur_vue.html`.

5.1. Doubles moustaches

Le premier exemple a montré comment insérer des variables réactives en tant que textes dans des balises, avec des doubles moustaches. Voici un exemple :

```
<div>valeur={{ valeur }}</div>
```

Ce qui est entre les moustaches sera remplacé par sa valeur. Cette valeur peut être calculée par n'importe quelle expression JavaScript, par exemple :

```
<div>valeur={{ valeur>=50 ? "c'est beaucoup" : "c'est peu" }}</div>
```

☛ Ajoutez cet exemple après le `<div class="row">` qui affiche le curseur puis tester le fonctionnement de la page.

5.2. Contenu HTML

Les doubles moustaches ne permettent pas de placer du code HTML généré à la volée qui sera interprété par le navigateur. Voici un exemple qui ne fonctionne pas :

```
<div>valeur={{ valeur>70 ? '<strong>'+valeur+'</strong>' : '<em>'+valeur+'</em>' }}</div>
```

☛ Ajoutez cet exemple et vérifiez que les balises HTML ne sont pas interprétées.

Pour afficher du HTML quand on lui fait confiance, il convient de le placer dans un attribut spécial `v-html`.

```
<div v-html="'valeur='+ (valeur > 70 ? '<strong>' + valeur + '</strong>' : '<em>' + valeur + '</em>')">
</div>
```

Faites attention aux contenus provenant d'utilisateurs, car `v-html` peut être vulnérable aux attaques Cross-Site Scripting (XSS) qui consistent à injecter du code malveillant (souvent du JavaScript) dans une page web.

☛ Ajoutez cet exemple puis tester le fonctionnement de la page.

5.3. Attributs dynamiques

On peut aussi employer des attributs dont la valeur est réactive. Il suffit de placer un `v-bind:` devant le nom de l'attribut. Voici un exemple à ajouter après le précédent :

```
<div v-bind:hidden="valeur < 80">Arrêtez-vous là !!!</div>
```

L'attribut `hidden` est un attribut standard HTML qui permet de masquer un élément dans la page sans le supprimer du code : il reste présent dans le DOM, mais n'est pas affiché à l'écran. Ici, sa valeur est booléenne, c'est une condition sur la valeur réactive.


Dans le cas d'un attribut booléen, si la valeur est *vraie par équivalence* alors l'attribut est présent, sinon il est enlevé. Sont considérés comme vrais par équivalence, « truthy » en anglais : la constante `true`, un entier strictement positif, une chaîne non vide, etc. Ici, quand `valeur < 80` alors le DOM contient `<div hidden>...</div>` sinon l'attribut n'est pas présent `<div>...</div>`.

Pour les attributs HTML de type chaîne ou nombre, l'attribut est toujours présent mais sa valeur peut être la chaîne vide.

Ces attributs sont affectés avec une expression JS. Il faut faire attention avec les délimiteurs de chaîne, et utiliser uniquement des apostrophes si l'attribut est encadré par des guillemets (ou vice-versa). Par exemple, `<a v-bind:href="'#' + cible">` définit l'attribut `href` d'un `<a>` comme étant la concaténation d'un `#` avec la variable `cible`. On pourrait aussi écrire `<a v-bind:href='"#" + cible">`. Par contre, `<a v-bind:href="'" + cible">` n'a aucune chance de marcher.

NB : la syntaxe `v-bind:attribute` peut être raccourcie en `:attribute`.

5.4. Classes et styles dynamiques

Pour définir un style ou une classe dynamique, qui change selon les variables réactives, on a un mécanisme spécifique expliqué sur [cette page](#) . Pour une classe, vous employez l'attribut spécial `v-bind:class` et pour un style, c'est `v-bind:style`.

Concernant les classes, il y a plusieurs possibilités pour définir une valeur qui les ajoute :

- directement une chaîne de caractère réactive ou une expression réactive qui retourne le nom de la classe.

- un objet `{'classe1': booléen1, 'classe2': booléen2, etc}`. Les classes dont les booléens sont vrais seront activées dans l'élément. Les `'...'` autour du nom de la classe ne sont pas nécessaires si ce nom ne contient que des lettres et chiffres (pas de tiret).
- un tableau `['classe1', 'classe2', etc]` des noms de classes à activer.

Voici un exemple :

```
<div v-bind:class="valeur == 0 ? 'text-warning' : 'text-info'">
  Conseil : vous pouvez bouger le curseur.
</div>
<div :class="{ 'text-danger': valeur > 60, border: valeur <= 60}">
  Plus de la moitié
</div>
```

Allez vérifier avec l'inspecteur ce qui se passe pour ces éléments.

Concernant les styles, on doit les fournir sous forme d'un objet dans la valeur de l'attribut `:style`. Voici un exemple :

```
<div :style="{ 'font-size': valeur/2+'px', letterSpacing: valeur/3+'px' }">
  {{valeur}}
</div>
```

Remarquez que l'on peut écrire le nom du style en « camelCase » sans quotes (ex : `fontSize`) ou en « kebab-case » avec des quotes (ex : `'font-size'`).

5.5. Rendu conditionnel

On a parfois besoin d'éléments qui n'apparaissent qu'à certaines conditions. Au lieu de rajouter un attribut `hidden` dynamique, on peut utiliser un attribut spécial, `v-if="condition"` qui commande l'insertion de l'élément dans le DOM. Voici un exemple :

```
<div v-if="valeur > 90">
  Le curseur est allé trop loin.
</div>
```

Inspectez le DOM. Vous allez voir que l'élément est remplacé par un commentaire tant que la condition n'est pas vraie. Essayez de mettre `:hidden="valeur <= 90"` à la place du `v-if`. L'élément est constamment dans le DOM mais invisible à l'écran.

Si vous avez besoin de faire une conditionnelle complète « si alors sinon », mettez un attribut `v-else` dans l'élément qui constitue le sinon. Voici un exemple :

```
<div v-if="valeur > 90">
  Le curseur est allé trop loin.
</div>
<div v-else>Tout va bien.</div>
```

L'attribut `v-else` doit être mis sur l'élément du même niveau qui suit immédiatement celui qui porte le `v-if`.

Il y a aussi l'attribut `v-else-if` pour enchaîner plusieurs alternatives. C'est documenté sur [cette page](#) [↗](#).

6. Rendu d'une collection

On va maintenant étudier un aspect important de Vue, la génération d'éléments selon une collection.

☛ Partons du fichier `liste_vue.html`

Cette fois, le code n'est pas terminé. Le but de l'exercice est d'afficher une liste d'items de pouvoir ajouter un nouvel item. Nous ne vous proposons pas de le faire en HTML+JS, mais vous pourriez estimer combien de temps ça vous prendrait, comparé au temps que vous allez passer avec Vue.

Plusieurs fonctionnalités manquent. Nous allons les réaliser une par une.

☛ Visualiser d'abord le rendu de la page `liste_vue.html`.

6.1. Liste des achats

D'abord, il y a une variable réactive, `achats` qui contient une liste d'objets `{nom, fait}`. Il y a déjà quelques objets.

☛ Ajoutez deux ou trois autres éléments dans la liste et constatez qu'il y a autant de lignes en plus sur la page.

Vue propose un attribut spécial pour faire une sorte de boucle dans une collection : `v-for`.

☛ Consultez la documentation [List Rendering](#) et lisez les premiers paragraphes, jusqu'à la ligne de séparation. Puis étudiez le code source.

☛ Actuellement, on ne voit pas le nom des achats. Ajoutez ce qu'il faut pour afficher seulement les noms dans la liste.

Bootstrap propose une structure intéressante pour notre liste, voir [cet exemple](#). Voici les éléments à générer au lieu des `` actuels :

```
<div v-for="achat in achats" class="input-group mb-3">
  <div class="input-group-text">
    <input type="checkbox" class="form-check-input mt-0" aria-label="fait ou non"/>
  </div>
  <input class="form-control" type="text" aria-label="nom de l'achat"/>
</div>
```

☛ Il faudrait que la propriété `fait` d'un achat change lorsque l'on clique sur la case à cocher et que le nom de l'article apparaisse dans le second `<input>`. Utiliser l'attribut `v-model` présenté au début du TP dans les deux `<input>` pour mettre en oeuvre ce fonctionnement.

Maintenant, la liste s'affiche correctement avec le nom et la case cochée en fonction des valeurs de la variable réactive `achats`. A l'inverse, une modification dans la page Web viendra automatiquement changer les propriétés `fait` et `nom` de l'élément concerné dans la variable `achats`. C'est le *view-model* en action.

Si vous ne voulez pas que l'on puisse modifier les éléments de cette manière, alors il faut ajouter des attributs `readonly` ou `disabled` et éventuellement remplacer les `v-model` par `:value`.

6.2. Écouteurs des boutons

- ☛ Définissez l'écouteur du bouton "Nettoyer" sur la fonction `onNettoyerAchats`.

Maintenant le bouton Nettoyer supprime tous les achats cochés.

On va étudier comment se passe l'ajout d'un nouvel achat. Il y a une variable réactive `nomArticle` qui est le `v-model` d'un `<input>` de formulaire. Quand on tape une chaîne, elle est placée dans cette variable. L'idée est que, quand on clique le bouton ajouter, cela enregistre le nouvel achat.

- ☛ Définissez l'écouteur du bouton ajouter sur la fonction `onAjoutAchat`.

La fonction `onAjoutAchat` est déjà programmée pour ne pas perdre de temps avec de la syntaxe JS. En fait, il n'y a quasiment pas de travail. On programme uniquement la logique métier de l'application et Vue se charge de gérer l'interface. On n'a aucune fonction à appeler pour rafraîchir les données et synchroniser la vue avec le modèle.

Vous devez remarquer que la variable `nomArticle` est toujours utilisée par `nomArticle.value` dans la partie script. Par contre, il ne faut pas mettre `achats.value` car cette variable est déclarée avec `reactive`. Comme nous l'avons vu précédemment, `reactive` n'est utilisable que pour des objets et des tableaux.

6.3. Message si liste vide

On voudrait qu'il y ait un message « la liste est vide » dans une `<div>` quand c'est le cas. En utilisant le rendu conditionnel, mettez en oeuvre cette fonctionnalité.

6.4. Activation des boutons

On voudrait que le bouton ajouter ne soit cliquable que si le nom en cours d'édition n'est pas vide. Et de même, on voudrait que le bouton Nettoyer ne soit pas cliquable quand la liste est vide. Ça concerne l'attribut booléen `disabled`, qui doit être présent ou absent.

- ☛ Ajoutez la mise en place automatique de l'attribut `disabled` sur le bouton Nettoyer quand `achats` est vide.
- ☛ Ajoutez la mise en place automatique de l'attribut `disabled` sur le bouton ajouter quand `nomArticle` est vide.

On préférerait maintenant ne pas voir du tout le bouton "Nettoyer" quand la liste est vide.

- ☛ Trouvez une solution pour afficher le bouton Nettoyer en bas de la liste uniquement quand elle n'est pas vide.

7. Bilan du TD

Vue vous permet de simplifier l'association de données avec la couche de présentation. Vous n'avez plus de fonction à écrire qui va chercher les éléments dans le DOM et qui change leur contenu, et les écouteurs d'événements sont réduits au minimum.

Vous savez maintenant programmer une application simple avec Vue.js en mettant en oeuvre :

- des variables réactives

- des propriétés calculées
- du contenu dynamique
- un affichage conditionnel
- le rendu d'une collection