COURS SUR VUEJS

Histoire du langage

Vue (prononcé comme le terme anglais « view ») est un framework JavaScript open-source pour construire des interfaces utilisateur. Conçu et pensé pour pouvoir être adopté de manière incrémentale, Vue est très simple à intégrer à d'autres bibliothèques ou projets existants tout en offrant la possibilité de développer efficacement tout type d'interface utilisateur.

Le framework a été créé, à l'origine, par **Evan You**, ancien ingénieur de Google ayant notamment travaillé avec Angular. Après avoir expérimenté une mécanique de réactivité qu'il jugeait plus intéressante que celle d'Angular, il décide de publier ses premiers résultats en 2013. Cinq ans plus tard, Vue.js devient le 3e projet sur Github en nombre de stars et Evan enchaîne les conférences à travers le monde.

Vue dispose aujourd'hui équipe internationale d'une trentaine de personnes, constituée de contributeurs bénévoles qui se sont formés avec les années. La décentralisation de l'équipe est à la fois une contrainte et une force qui lui a permis de diffuser le framework beaucoup plus rapidement à plusieurs endroits à la fois. Contrairement à React et Angular qui sont portés par les entreprises Facebook et Google, Vue.js est totalement indépendant et entièrement piloté par la communauté.

Installation de Vue.js

Installation locale

L'installation de Vue.js pour un usage local nécessite la présence de **node js**. Pensez donc d'abord à l'installer (en version 8.9 minimum) car son gestionnaire de packages **npm** est essentiel pour créer un véritable projet Vue.

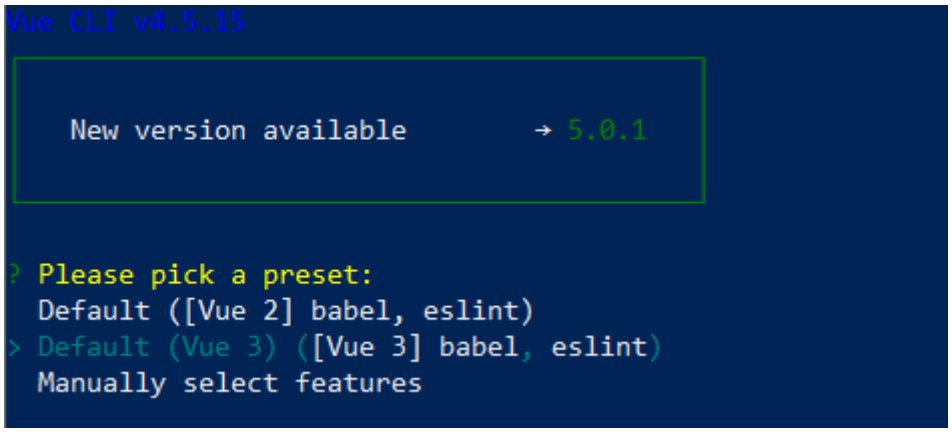
Une fois installé, vous pouvez utiliser la commande ci-dessous dans un terminal pour installer **Vue CLI**, l'interface en ligne de commande de Vue.js, qui servira à fournir le standard nécessaire pour le développement des applications Vue :

- npm install -g @vue/cli>

Une fois cette étape bouclée, elle ne sera plus nécessaire pour créer de nouveaux projets Vue. Dorénavant, pour créer un projet Vue, il suffit de taper cette ligne de commande dans le terminal dans le répertoire de stockage du projet :

- vue create name-project

Une interface d'installation sera ensuite visible. Il faudra donc choisir les pré-configurations disponibles avec Vue CLI. Utilisez les touches fléchées pour vous déplacer puis cliquez sur *entrer* pour choisir la pré-configuration par défaut Vue 3, qui est la dernière version de Vue.js.



Le terminal va ensuite commencer à installer tous les paquets nécessaires pour la CLI. Une fois l'installation terminée, vous pouvez voir votre projet en ouvrant, dans un éditeur de code, le répertoire portant le nom du porjet. Pour lancer le projet, exécutez la commande suivante :

- npm run serve

Le serveur de Vue CLI se mettra en route et compilera les données pour ensuite générer une page unique où l'application Vue sera utilisable. Pour voir cette page sur le navigateur (chrome, de préférence), entrez l'url indiquée à la fin du traitement du serveur de Vue CLI.

Usage en ligne

Outre ces étapes, vous pouvez également utiliser Vue.js en ligne et l'intégrer directement dans votre projet à travers un lien CDN comme ci-dessous, en plaçant la balise soit dans l'entête de votre page (header), soit juste avant la balise de fin du corps de la page (body).

- <script src="https://unpkg.com/vue@3" ></script>

Avec cette méthode, il faudra configurer Vue.js en le liant à une balise identifiée par un *id* puis créer l'application Vue dans une balise script suivant le modèle suivant :

```
<div id="app">{{ message }}</div>

<script>
  Vue.createApp({
    data() {
      return {
        message: 'Hello Vue!'
      }
    }
  }).mount('#app')
</script>
```

Pour développer en Vue.js, vous aurez besoin d'un bon éditeur de code. Nous conseillons donc **Visual Studio Code** qui est un éditeur de code léger et très populaire en plus d'être adapté au framework en proposant l'extension **Vetur** pour coder en Vue.js. Pour tester le code, le navigateur Chrome fournit l'outil **vue-devtools** qui permet le débogage du code.

Bases du framework Vue.js

Une fois votre projet Vue créé, vous pouvez profiter des diverses fonctionnalités qu'offre le framework et cela, très facilement.

Dans Vue.js, les données gérées par l'application se trouvent dans une fonction spéciale qui retourne à l'instance de Vue toutes les données disponibles : la fonction **data()**. Elle permet la déclaration et l'initialisation des données mais ne gère pas leur modification. Les données sont déclarées comme des propriétés de l'objet que retourne la fonction. Ces données peuvent être des *chaînes de caractères*, des *données numériques* (int et float), des *booléens*, des *tableaux* ou encore des *objets*. Chaque donnée de la fonction data() peut être récupérée entre les balises HTML en tapant le nom de la variable et en l'entourant de deux moustaches : c'est *l'interpolation*.

```
<span>Message: {{ msg }}</span>
```

Ces données peuvent aussi être récupérées dynamiquement dans un attribut de balises HTML en utilisant la directive **v-bind** de Vue qui permet de transformer le nom d'un attribut en variable afin de la lier à une donnée précise de l'instance Vue. Cela rend la valeur de l'attribut dynamique.

```
<div v-bind:id="dynamicId"></div>
```

Cette syntaxe peut aussi être abrégée en retirant « v-bind » :

```
<div :id="dynamicId"></div>
```

Pour mieux manipuler les données, Vue.js propose l'usage de directives manipulant les conditions : **v-if**, **v-else-if** et **v-else**.

```
<div v-if="type === 'A'">
  A
</div>
<div v-else-if="type === 'B'">
  B
</div>
<div v-else-if="type === 'C'">
  C
</div>
<div v-else>
  Not A/B/C
</div>
```

La première div sera créée et affichée dans le navigateur si le contenu de la variable type est A, sinon ce sera la seconde condition qui sera testée et ainsi de suite. Une balise contenant une directive *v-if* peut ne pas être précédée d'autres directives comme *v-else-if* ou *v-else*. Mais ces dernières nécessitent obligatoirement d'être précédées d'une balise *v-if*.

Pour effectuer des boucles, Vue nous propose la directive **v-for** qui peut être utilisée soit par incrémentation, soit suivant le contenu d'un objet ou d'un tableau.

Dans le cas d'une *incrémentation* suivant une valeur définie, on a :

```
<span v-for="n in 10">{{ n }}</span>
```

Ici, la balise span sera rendue 10 fois avec n initialisé à 1 et non 0 comme pour la plupart des langages de programmation.

Dans le cas d'un *objet*, on a :

```
<div v-for="item of items"></div>
```

Dans le cas d'un *tableau*, on a :

```
data() {
  return {
    items: [{ message: 'Foo' }, { message: 'Bar' }]
  }
}
```

```
<li v-for="item in items">
  {{ item.message }}
</li>
```

Dans un formulaire, Vue permet facilement de lier une variable à un champ de saisie afin de récupérer sa valeur. La directive utilisée dans ce cas est **v-model** et elle peut lier une variable définie dans *data()* à un champ input, un textareas ou encore un select. La présence de cette directive ignore l'attribue *value* des balises du champ auquel il est associé.

```
<input v-model="text">
```

Grâce à la fonction *data()*, on peut stocker des données mais pas les modifier. Vue nous propose là, une propriété spéciale de l'instance qui est un objet : **methods**.

Dans cet objet, on peut créer des fonctions qui, elles, seront capables de modifier les données de la fonction *data()*. Ces fonctions peuvent être utilisées depuis le template comme à l'intérieur d'autres fonctions.

```
data() {
  return {
    count: 0
  }
},
methods: {
  increment() {
    this.count++
  }
},
```

```
<button @click="increment">{{ count }}</button>
```

Composants Vue

Le système de Vue se base principalement sur l'usage de **composants**. Les composants nous permettent de diviser l'interface utilisateur en éléments indépendants et réutilisables, et de penser à chaque élément isolément. Ceci est très similaire à la façon dont nous imbriquons des éléments HTML natifs, mais Vue implémente son propre modèle de composant qui nous permet d'encapsuler un contenu et une logique personnalisés dans chaque composant.

Un composant Vue possède ses propres données et donc un *template* pour le HTML et une fonction *data()* pour gérer ses propres données.

```
data() {
  return {
    count: 0
  }
},
template: `
  <button @click="count++">
    You clicked me {{ count }} times.
  </button>`
```

Une fois créé, le composant doit être déclaré dans une propriété *components* au sein de l'application Vue qui l'utilise. Seulement après, il peut être utilisé dans le template de la Vue comme une balise HTML normale. Ainsi, tout le contenu du composant est réutilisable avec seulement une ligne de code qui appelle qui affiche tout le contenu du template de ce composant. Pratique, pas vrai ?

```vue
<script>
import ButtonCounter from './ButtonCounter.vue'

export default {
  components: {
    ButtonCounter
  }
}
</script>

<template>
  <h1>Here is a child component!</h1>
  <ButtonCounter />
</template>
```

```vue
<h1>Here are many child components!</h1>
<ButtonCounter />
<ButtonCounter />
<ButtonCounter />
```

Liaisons de de données entre composants

Il est bien de pouvoir réutiliser tout un contenu prédéfini, mais il est encore mieux de pouvoir dynamiser ce contenu. Et Vue excelle également dans ce domaine grâce à l'usage des **props**. Il s'agit de données qu'un *composant parent* (la Vue qui utilise un composant) peut transmettre à un *composant enfant* (le composant utilisé par la Vue courante) qui la récupère et l'utilise comme une donnée définie sur sa propre instance.
Le composant parent passe une **donnée statique** au composant enfant :

```vue
<BlogPost title="My journey with Vue" />
<BlogPost title="Blogging with Vue" />
<BlogPost title="Why Vue is so fun" />
```

Le composant enfant récupère la donnée passée dans l'attribut *title* en la définissant dans une propriété *props* de l'instance et l'affiche :

```vue
<!-- BlogPost.vue -->
<script>
export default {
  props: ['title']
}
</script>

<template>
  <h4>{{ title }}</h4>
</template>
```

Le composant parent passe un ensemble de données dans un objet qu'il lie avec la directive *v-bind* à l'attribut cible (**données dynamiques**) :

```vue
export default {
  // ...
  data() {
    return {
      posts: [
        { id: 1, title: 'My journey with Vue' },
        { id: 2, title: 'Blogging with Vue' },
        { id: 3, title: 'Why Vue is so fun' }
      ]
    }
  }
}
```

Le composant enfant récupère ensuite toutes ces données et les affiche dans une boucle de composants :

```vue
<BlogPost
  v-for="post in posts"
  :key="post.id"
  :title="post.title"
 />
```

De la même façon que le parent peut envoyer des données à l'enfant, ce dernier peut également émettre des données à son parent. Il s'agit là d'utiliser dans l'enfant, la fonction **$emit()** qui permet depuis l'enfant, de créer un *évènement personnalisé* qui sera écouté par le parent par la directive **v-on** de Vue qui permet d'écouter des évènements sur la balise sur laquelle elle est utilisée.

```vue
<!-- BlogPost.vue, omitting <script> -->
<template>
  <div class="blog-post">
    <h4>{{ title }}</h4>
    <button @click="$emit('enlarge-text')">Enlarge text</button>
  </div>
</template>
```

```vue
<BlogPost
  ...
  @enlarge-text="postFontSize += 0.1"
 />
```

De cette manière, en cliquant sur le bouton « Enlarge text », l'évènement personnalisé « enlarge-text » sera créé et émis dans le composant parent. Ainsi, grâce à la directive *v-on* (abrégée en @), l'évènement est écouté puis l'instruction consistant à ajouter 0.1 au contenu de la variable postFontSize est exécutée.

N.B : L'évènement personnalisé créé dans le composant enfant, doit être déclaré dans une propriété *emits* au sein du composant enfant. La syntaxe des évènements personnalisés est le kebab-case.

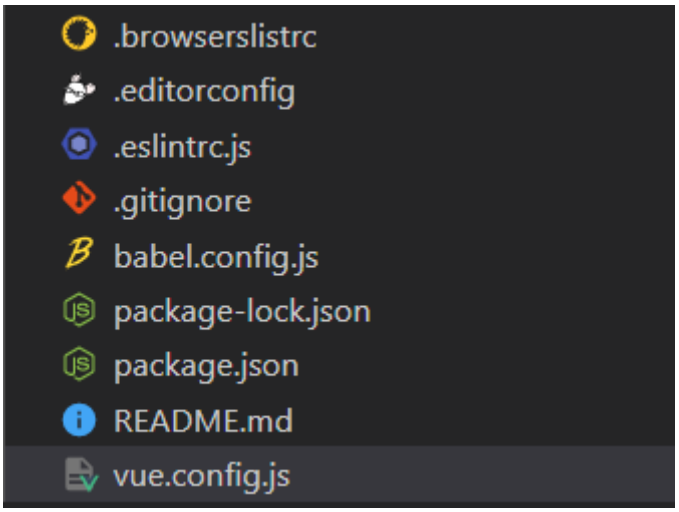
```vue
<!-- BlogPost.vue -->
<script>
export default {
  props: ['title'],
  emits: ['enlarge-text']
}
</script>
```

Communication avec une API

Le cœur de la bibliothèque de Vue.js étant concentré uniquement sur la partie Vue, on utilise une API pour interagir directement avec une base de données. C'est ainsi que Vue nous propose une certaine configuration pour faciliter cette liaison avec l'API.

1. Tout d'abord, on crée un fichier vue.config.js à la racine du projet Vue.

Dans ce fichier, on pourra écrire un code simple qui permet de rediriger les requêtes de la Vue vers un port spécifique où l'API écoutera et récupérera ces requêtes. De cette façon, la Vue peut directement communiquer avec l'API qui elle, intégrera une base de données afin d'enregistrer les données reçues de l'interface utilisateur.



1. Grâce à ce fichier, on définit un objet **devServer** qui contiendra un objet **proxy** dans lequel une propriété porte par exemple le nom **^/api**.

Il s'agit là de traiter toutes les requêtes contenant la sous-chaine "/api", précédée de l'adresse du serveur client.

Cette propriété aura pour valeur un objet avec deux propriétés spécifiques : **target** et **changeOrigin**

- La propriété *target* définit l'adresse du serveur vers laquelle rediriger les requêtes de l'API
- La propriété *changeOrigin* permet quant à elle, d'autoriser la redirection des requêtes d'où sa valeur booléenne vraie

```
module.exports = {
    devServer: {
        proxy: {
            '^/api': {
                target: 'http://localhost:3000',
                changeOrigin: true
            }
        }
    }
}
```

1. On peut maintenant définir des fonctions qui peuvent lancer des appels à l'API et récupérer les réponses.

- Notons qu'une fonction qui fait appel à l'API doit toujours être **asynchrone**
- On fait une requête à l'API avec la fonction **fetch()**
- On récupère la réponse de l'API avec **await** puis on la retourne à la Vue
- Ne surtout pas oublier **d'exporter la fonction** !

```
// Requête vers l'API
async function traitement () {
  // Requête vers l'api et récupération de la réponse
  const response = await fetch('/api/select/data')
  // Réponse de l'api à la vue
  return await response
}

// Exportation de la fonction
module.exports = {
  traitement
}
```

1. Une fois les configurations terminées, la Vue n'a plus qu'à appeler la fonction de requête à l'API.

On peut définir une fonction qui appelle celle de traitement de la requête vers l'API. En appelant ensuite cette fonction de Vue, la fonction asynchrone se déclenche, récupère et stocke les données reçues dans une variable locale *data* de l'instance de Vue.

- D'abord, on **définit une fonction** qui s'occupera d'appeler la fonction de traitement de la requête vers l'API
- Après avoir importé la fonction asynchrone dans la Vue, on la **déclenche**
- Aussitôt, à l'aide de la fonction **.then()**, on attend une réponse qu'on stocke dans la variable res
- On **convertit** ensuite cette réponse de l'API en *JSON* avec la fonction **.json()**
- On attend toujours de récupérer cette nouvelle donnée transformée avec un second **.then()**
- **Dans cette dernière fonction** qui prend en argument la nouvelle donnée, on peut maintenant traiter les données reçues de l'API et les enregistrer dans la Vue pour d'autres usages par exemple

```
methods: {
  // Fonction de récupération des données
  selectData: () => {
    traitement()
      .then(res => res.json())
      .then(data => {
        // Récupération et stockage des données
        this.data = data
      })
  },
```

TP Vue.js

Vous êtes stagiaire dans une entreprise de lecture de livres en ligne. Votre tuteur de stage vous demande de créer en local un prototype du site de lecture des livres en Vue.js en intégrant uniquement la fonctionnalité d'affichage des livres disponibles. Ces livres seront récupérés depuis une base de données via une API que l'entreprise utilise déjà sur le port 3000 du serveur local.

Chaque livre comporte un libelle, un auteur, un nombre de chapitres et un statut qui définit si le livre est en cours d'écriture ou non. Votre tuteur de stage vous informe qu'il a déjà créé une route sur l'API qui écoutera la requête de votre projet sur l'adresse : /api/les-livres et renverra un tableau contenant les livres disponibles avec chaque objet de livre défini avec les attributs : *libelle* et *auteur* de type chaine de caractères, *nbChapitre* de type entier et *enCours* de type booléen.