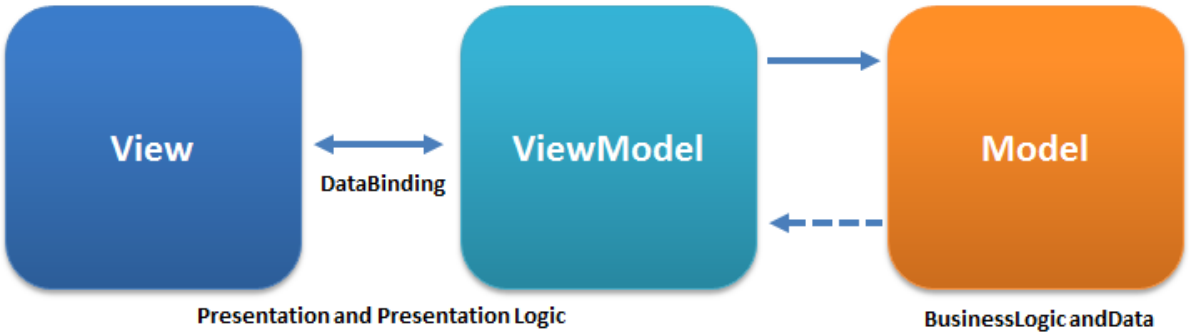
VUE -> MVVM



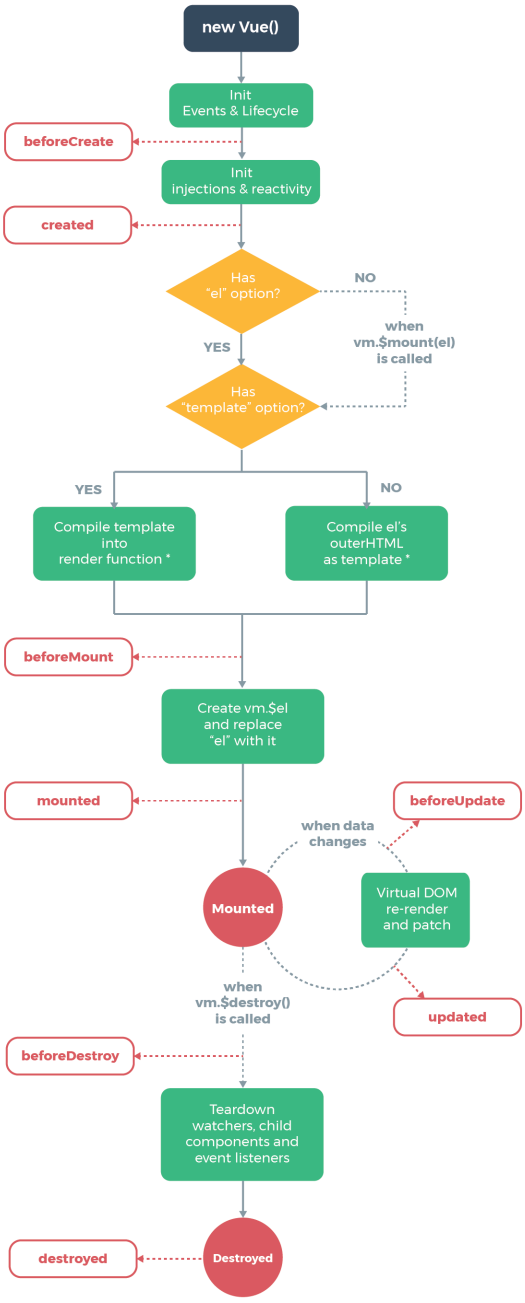
Que pouvons-nous faire avec vue.js :
Améliorer le HTML sans étape de construction
Intégration de composants Web sur n'importe quelle page
Application monopage (SPA)

Fullstack / Rendu côté serveur (SSR)
Jamstack / Génération de sites statiques (SSG)
Ciblage des ordinateurs de bureau, des mobiles

Dans ce cours nous allons aborder 4 phases de Vue.js :

1) le fonctionnement d'une instance de l'objet Vue (new Vue( ) en version2 et createApp en version3)
2) la création d'un composant Vue (Component)
3) vue-cli : la génération d'une application (cli : command line interface)
4) Vuex (communication entre composants par le Store)

1) L’instance Vue



* template compilation is performed ahead-of-time if using a build step, e.g. single-file components

Le cœur de Vue.js est son instance. Chaque application Vue commence par en créer une et c'est un objet qui vous aidera à créer le comportement souhaité.

L'instance de Vue contient différentes options:

données,
getters,
setters,
modèle,
méthodes,
calcul (computed),
observateurs,
cycles de vie.

Comme vous pouvez l'imaginer, l'instance est responsable de différentes choses, par exemple le paramétrage de l'observation des données, la compilation du modèle, le montage de l'instance dans le DOM, la mise à jour du DOM lorsque les données changent.

Syntaxe des templates

Vue compile les templates en des fonctions de rendus de DOM virtuel. Combiné au système de réactivité, Vue est en mesure de déterminer intelligemment le nombre minimal de composants pour lesquels il faut redéclencher le rendu et d'appliquer le nombre minimal de manipulations au DOM quand l'état de l'application change.

Interpolation

La forme la plus élémentaire de la liaison de données est l'interpolation de texte en utilisant la syntaxe "Mustache" (les doubles accolades)

Les doubles moustaches interprètent la donnée en tant que texte brut, pas en tant que HTML. Pour afficher réellement du HTML, vous aurez besoin d'utiliser la directive v-html afin de se protéger des attaques XSS :

```
<p>En utilisant les doubles moustaches : {{ rawHtml }}</p>
<p>En utilisant la directive `v-html` : <span v-html="rawHtml"></span></p>
```

Les moustaches interprètent les **expressions** javascript mais pas les déclarations ni les blocs (if… for…etc)

exemple:

{{ number + 1 **}}** OK !
{{ var = 1 **}}** PAS OK!
{{ ok ? OUI : NON **}} OK ! (** opérateur ternaire)
{{ if (ok) { … } **}}** PAS OK !

Attributs

Les moustaches ne sont pas utilisés dans les attributs des éléments pour cela il y a des directives du type **v-directive**

v-bind:
v-on:
v-html:
v-model:
v-text:
v-if
v-for

exemple

<button v-on:click="fonctionClick()">je clique

Création d’un objet Vue:

Version 2

```
new Vue({  })
ou
var ov = new Vue({  })
```

Version 3

```
const App = {…}

Vue.createApp(App).mount("#el")
```

Attachement d’un objet Vue à un élément du DOM:

```
<div id="app">
</div>

<script>
new Vue({
     el: "#app"
})
</script>
```

Les cycles de vie d’un objet Vue:

Des actions peuvent être exécutées à chaque évènements,t du cycle de vie listé ci-dessous. (ce qu’on appelle des **hooks**)

beforeCreate:
Il est appelé immédiatement après que l'instance ait été initialisée et avant que toutes le les options soient traitées.

create:
Il est appelé juste après la création de l'instance et après que toutes les options[1] aient été mises en place.

beforeMount:
Il est appelé juste avant que le montage du DOM ne commence.

mounted:
Il est appelé lorsque l'instance a été montée et que le el(le DOM) a été remplacé.

beforeUpdate:
Il est appelé lorsque certaines données changent et avant que le DOM n'ait été rendu.

updated:
Elle est appelée lorsque certaines données ont été modifiées et que le DOM a été rendu à nouveau.

activated:
Ce crochet est utilisé pour les composants <keep-alive> , il vous permet de savoir quand un composant à l'intérieur du tag <keep-alive></keep-alive> est activé.

deActivated:
Ce crochet est également utilisé pour les composants <keep-alive>, il vous permet de savoir quand un composant à l'intérieur de la balise <keep-alive></keep-alive> est désactivé.

beforeDestroy:
Il est appelé juste avant que l'instance Vue ne soit détruite. À ce stade, l'instance est encore pleinement fonctionnelle.
Appelée côté client

destroyed:
Il est appelé après que l'instance Vue ait été détruite, cela ne signifie pas qu'il va supprimer tout le code du DOM mais qu'il va supprimer toute la logique du Java Script et que l'instance n'existera plus.

beforeCreate, created, beforeMount, mounted sont appelées automatiquement. Les autres quand il se passe quelque chose.

Propriétés (Properties)

Les propriété d'un objet Vue sont déclarées ainsi

```
new Vue({
el: "#app",
data: {
     clé : valeur,
     clé: valeur
     }
}
```

Les datas peuvent être déclarées à l'extérieur

ou

```
var donnees = {
     clé : valeur,
     clé: valeur
     }

new Vue({
el: "#app",
data: donnees
}
```

version 3

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

<script>
     const App = {
               data() {
                    return {
                    counter: 0
                    }
               }
          };

     Vue.createApp(App).mount('#app');

</script>
```

Méthodes (Methods)

Ce sont des fonctions statiques habituellement utilisées pour réagir aux événements qui se produisent dans le DOM et elles acceptent les arguments.

Elles sont utiles pour relier des fonctionnalités à des événements, ou même simplement pour créer de petites parties de logique à réutiliser. Vous pouvez appeler une méthode à l'intérieur d'une autre méthode

Propriétés calculées (computed properties)

Elles n'acceptent pas les arguments et sont très pratiques pour composer de nouvelles données à partir de sources existantes, elles obtiennent des valeurs dynamiques basées sur d'autres propriétés.

Une fonction peut être calculée sans qu'il y ai une action spécifique du genre cliquer sur un bouton.

Elles sont souvent utilisées pour faire des calculs et, à la fin, pour renvoyer une valeur.

Une caractéristique intéressante des propriétés calculées est qu'elles sont mises en cache, ce qui signifie que la fonction ne sera exécutée qu'une seule fois tant que les valeurs ne changent pas, même si elle est appelée plusieurs fois dans le même modèle.

C'est la même chose en vue sauf qu'il ajoute du caching

- Vue va chercher des **données réactives** dans votre fonction **computed**
- La **première** exécution de la fonction computed **va créer un cache**, qu'il trouve ou non une donnée réactive
- Si au prochain rendu (à l'update), il voit qu'une **donnée réactive utilisée dans le computed a changé**, il ré-exécute le computed pour créer un nouveau résultat et le remet en cache
- S'il ne trouve aucune dépendance (donnée réactive), il renverra toujours le même résultat, celui du cache précédent.

Il y a plusieurs façons dans Vue de fixer des valeurs pour la vue. Cela inclut la liaison directe de la valeur des données à la vue, l'utilisation d'expressions simples ou l'utilisation de filtres pour effectuer des transformations simples sur le contenu. En plus de cela, nous pouvons également utiliser des propriétés calculées pour obtenir des valeurs basées sur des éléments à l'intérieur du modèle de données.

Les propriétés calculées dans Vue sont incroyablement utiles. A tel point que, à moins que vous ne cherchiez à déclencher une fonction après un événement, comme un clic, vous devriez probablement analyser si votre fonction peut se situer à l'intérieur de la propriété calculée avant d'examiner la propriété des méthodes. **La raison est que les calculs effectués à l'intérieur des propriétés calculées sont mis en cache et ne seront mis à jour qu'en cas de besoin.**

Watchers

Fonction asynchrones (les computed sont synchrones)

fonction exécutée si une data change mais de manière asynchrone.

Exemple: je tape du code dans dans un input et à l'évènement "blur" je déclenche une requête à une BDD

(Voir exercice vue14.html)

COMPOSANTS

Le système de composants est un autre concept important de Vue, car il s'agit d'une abstraction qui nous permet de construire des applications à grande échelle composées de petits composants autonomes et souvent réutilisables.

Lorsque vous appelez new Vue(), vous créez ce que l'on appelle une instance Vue. Une instance Vue est en fait un objet JavaScript qui a la capacité d'interagir avec le DOM.

Les composants Vue sont également des instances Vue. Mais la principale différence est que les composants appartiennent à d'autres composants/ instances Vue, tandis que les instances simples sont montées directement dans un élément existant du DOM (par existant, j'entends du HTML qui n'est pas généré par Vue et qui existait avant la création de l'instance).

Donc, fondamentalement, la principale différence vient de l'utilisation : vous créez généralement une instance Vue simple avec new Vue lorsque vous devez attacher cette instance directement au DOM comme élément racine. Les composants Vue, quant à eux, sont toujours attachés à d'autres composants.

SINGLE FILE COMPONENTS : fichier .vue

Un single-file-component sous Vue.js c'est :

- Une balise **<template>** contenant les éléments HTML à rendre pour chaque appel du component, où l'utilisation de Vue.js via les attributs ou via {{ }} est possible (HTML + {{ }} + directive).

- Une balise **<script>** qui export un objet représentant la configuration de ce component (propriété data, methods, etc...), et où il est possible d'utiliser "import" pour importer d'autres components.

- Une balise **<style>** permettant d'ajouter du css si besoin

App.vue est le root component qui appelle et gère les autres components

VERSION 3

Dans le code d'initialisation de l'application Vue 2, nous avons utilisé l'objet Vue importé de la bibliothèque pour créer cette instance d'application et toutes les autres nouvelles.

Avec cette approche, il était impossible d'isoler certaines fonctionnalités à une seule des instances Vue puisque les applications Vue utilisaient toujours le même objet Vue importé de la bibliothèque.

Application vue CLI

vue cli permet de générer une application complète qui à la fin génère un code avec un bundle javascript généré en un seul fichier optimisé et minimisé.

```
npm i -g @vue/cli

vue create myapp

cd myapp

npm run serve
```

…. création des component fichiers avec extension **.vue**

npm run build -> génération index.html avec bundles js et css

npm run serve

exécute l'application vue.js en mode **Développement** sur le port 8080 par défaut. Ce n'est pas notre application finale.

Il faut executer un build pour générer l'application finale qui est exécutée par défaut sur le port 5000 , on est en production.

ROUTER V4 pour version VUE3

Installer le module (dans notre application créé par vue-cli

```
npm i vue-router@next
```

créer le répertoire **router** dans **src**
dans router créer le fichier **index.js**

exemple

```
import { createWebHistory, createRouter } from "vue-router";
import Home from "@/views/Home.vue";
import About from "@/views/About.vue";

const routes = [
  {
    path: "/",
    name: "Home",
    component: Home,
  },
  {
    path: "/about",
    name: "About",
    component: About,
  },
];

const router = createRouter({
  history: createWebHistory(),
  routes,
});

export default router;
```

Dans main.js

```
import { createApp } from 'vue'
import App from './App.vue'
import Router from './router'

createApp(App).use(Router).mount('#app')
```

Créer les vues dans un repertoire views (fichiers .vue)

exemple page Home.vue

```
<template>
  <h1>Home Page</h1>
</template>
```

exemple page About.vue

```
<template>
  <h1>About Page</h1>
</template>
```

Dans le composant "menu" (exemple dans App.vue ou un autre composant).

```vue
<template>

  <div id="nav">
      <router-link to="/">Home</router-link> |
      <router-link to="/about">About</router-link> |
      <router-view/>
    </div>

</template>
```

Vuex

Vuex est un magasin (Store) central pour les applications Vue.js.

Les données stockées dans le magasin Vuex sont accessibles depuis n'importe quel composant de l'application.

Le partage des données de cette manière résout le problème du "prop drilling" (passage des données par des composants qui n'en ont pas besoin pour les amener là où elles sont nécessaires).

Vue.js rend la création d'applications SPA simple et facile. Les applications monopages sont des applications qui fonctionnent à l'intérieur du navigateur et n'ont pas besoin de recharger les pages pendant l'utilisation. Elles réécrivent dynamiquement la page en cours.

L'un des défis majeurs de la construction d'un SPA avec Vue.js est de faire passer les données d'un composant de Vue.js à un autre.

Vuex est la bibliothèque officielle de Vue.js qui résout les problèmes de stockage. Les données qui y sont stockées peuvent être accessibles depuis n'importe quel composant de l'application.

Un stockage peut être mis en place dans Vue.js de cette manière :

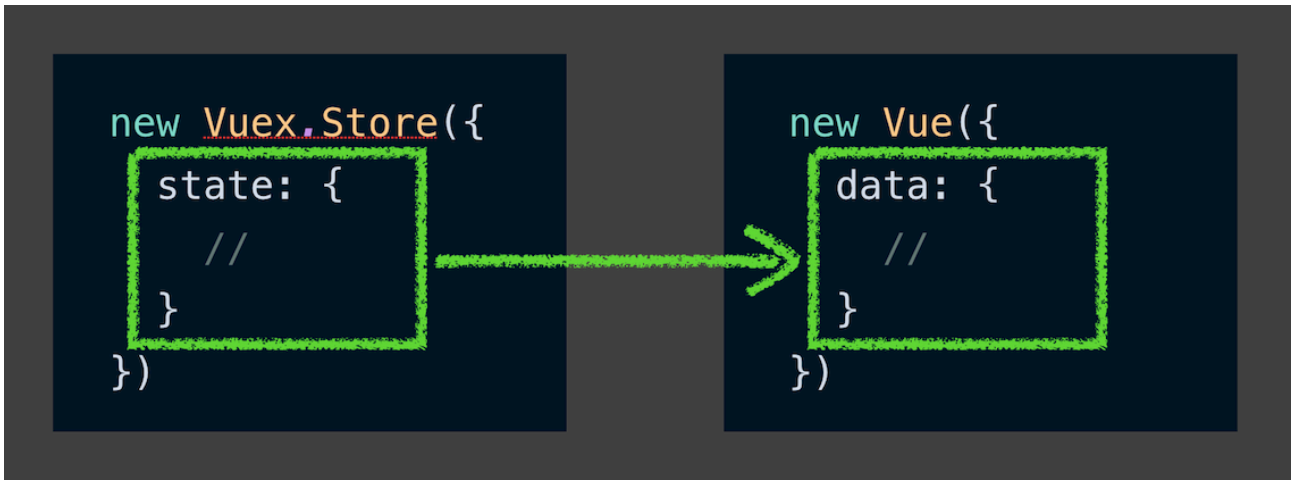
```
npm install vuex@next —save
```

Dans le main.js:

```
import { createApp } from 'vue'
import { createStore } from 'vuex'

const store = createStore({
     state() {
          return {
               nom:           "Dupont",
               prenom:   "Jean",
               age:           35
          }

})

createApp(…).use(store)
```

Ajoute le folder store avec index.js qui a un état disponible pour tous les composants, ça correspond au data: que vous connaissez:

STATE



Exemple

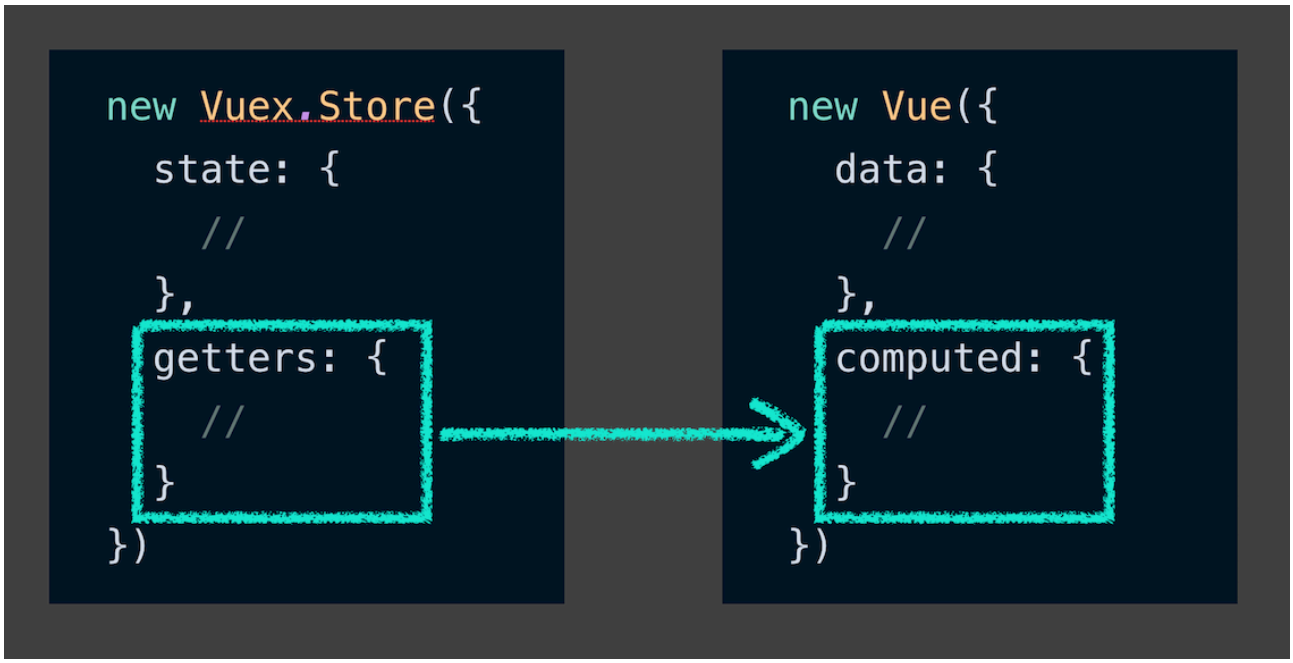
```
new Vue.Store({
      state: {
            nom:        "Dupont",
            prénom:     "Jean",
            age:        35
      }
})
```

Pour accéder aux données dans le template , dans un fichier .vue:

```
<template>
      <div>
            nom :       {{ $store.state.nom }}
            prénom :    {{ $store.state.prenom }}
            age:        {{ $store.state.age }}
      </div>
</template>
```

GETTERS:

correspond au computed



Exemple

```
toujours dans main.js

      state()
            return {
                  nom: "Dupont",
                  prénom: "Jean",
                  age: 35
      },
      getters:
            {
                  affiche: (state) => {
                  return `${state.name} ${state.prenom} age de $
{state.age}`
            }
```

dans un template retrouver la variable calculée:

{{ $store.getters.affiche }}

Mutations

propriétés responsables des **mutations** du **state**

La seule façon de vraiment modifier l'état dans un **store Vuex** est d'acter une mutation. La mutation est comme un évènement. Elle a un **type** et un **gestionnaire** et reçoit en argument le **state** et éventuellement des arguments **payload**

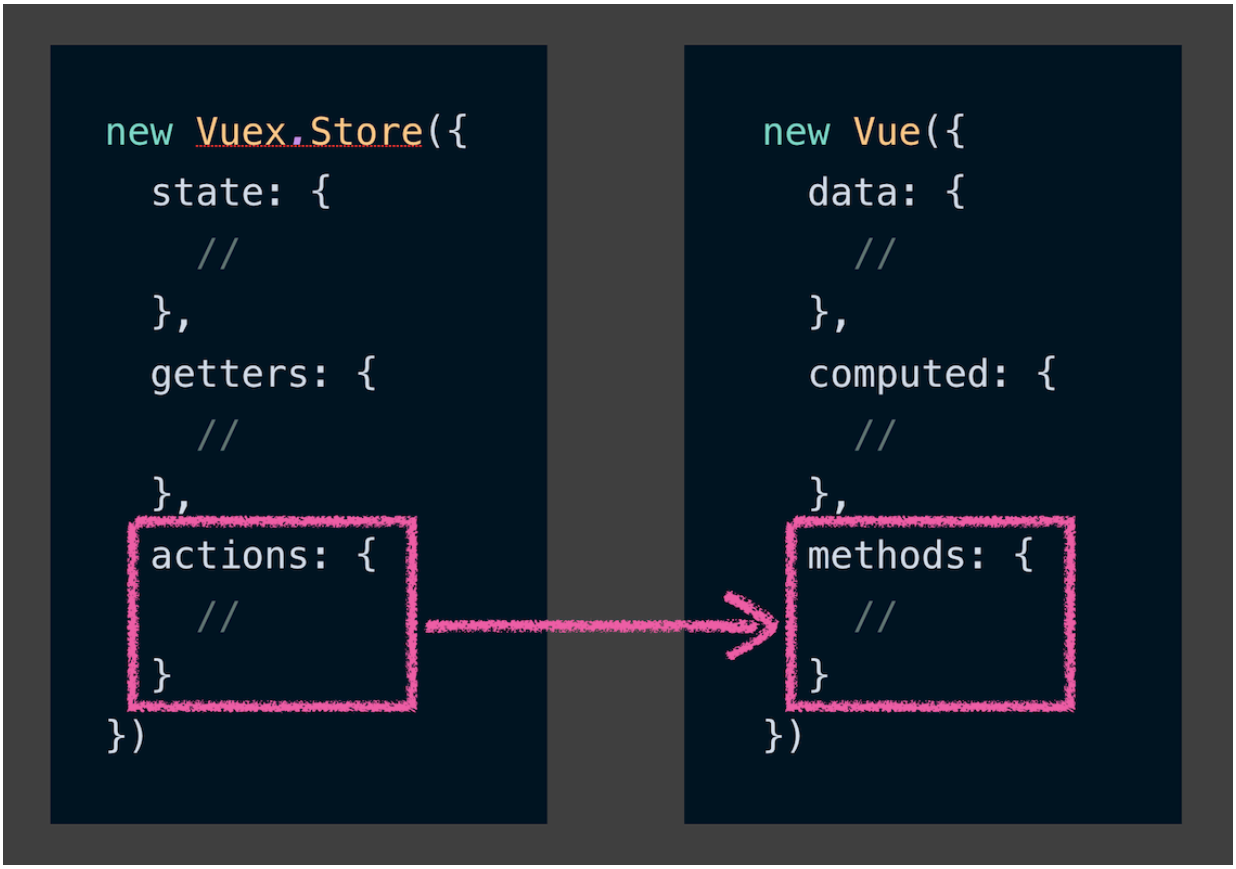
```
      state: {
            nom: "Dupont",
            prenom: "Jean",
            age: 35
      },
      getters: {
            affiche: (state) => {
                  return `${state.name} ${state.prenom} age de ${state.age}`
            }
      },

      mutations: {
            AJOUTE_ANNEES(state) {
            state.age++           }

      }
```

Dans la mutation un 2ème argument **payload** permet d'envoyer des données. Pour l'instant les données ne sont pas modifiées il faut "*acter*" ces action soit par un commit à partir du **$store** dans le composant soit par des a**ctions** déclenchées par un dispatch. La mutation est une sorte d'abonnement à un évènement. Les mutations sont des opérations synchrones.

Dans une méthode des composants utiliser ainsi

$store.commit('AJOUTE_ANNEES',5)

Actions



Elles nous servent à coordonner la logique derrière les mutations. Les actions sont asynchrones et permettent de déclencher plusieurs mutations en une seule action.

- Au lieu de modifier l'état, les actions actent des mutations.
- Les actions peuvent contenir des opérations asynchrones.

```
new Vue.Store({
      state: {
            nom: "Dupont",
            prenom: "Jean",
            age: 35
      },

      mutations: {
            AJOUTE_ANNEES(state, payload) {
            state.age += payload
      },

      actions: {
            ajouteAnnees(context,arg) {
                  context.commit('AJOUTE_ANNEES',arg);
            }
      }
})
```

Autre exemple

```
export default new Vuex.Store({
    state: {
        count: 0
    },
    mutations: {
        INCREASE_COUNT(state, amount = 1) {
            state.count += Number(amount)
        },
        DECREASE_COUNT(state, amount = 1) {
            state.count -= Number(amount)
        }
    },
    actions: {
        updateCount(context, amount) {
            if (amount >= 0) {
                context.commit('INCREASE_COUNT', amount)
            } else {
                context.commit('DECREASE_COUNT', amount)
            }
        }
    }
})
```

une action peux accéder à plusieurs mutations

Router

sur linux ou mac os x

```
vue add vue-router
```

sur windows

```
vue add router
```

ajoute un folder router avec création d'un index.js par defaut:

ici vue.js 3

```
import { createRouter, createWebHistory } from 'vue-router'
import Home from '../views/Home.vue'

const routes = [
  {
    path: '/',
    name: 'Home',
    component: Home
  },
  {
    path: '/about',
    name: 'About',
    // route level code-splitting
    // this generates a separate chunk (about.[hash].js) for this route
    // which is lazy-loaded when the route is visited.
    component: () => import(/* webpackChunkName: "about" */ '../views/
About.vue')
  }
]

const router = createRouter({
  history: createWebHistory(process.env.BASE_URL),
  routes
})

export default router
```

Plug-ins

Les modules CLI officiels sont préfixés d'un @ : @vue/cli-plugin-xxx sinon ils sont simplement préfixés par vue.

installer un plug-in

exemple

vue add @vue/cli-plugin-eslint

système de raccourci

vue add @vue/eslint

voire

vue add apollo

Exemple installation de Bootstrap

plug-in bootstrap pour vue

bootstrap-vue.js.org

vue-cli-plugin-bootstrap-vue

raccourci -> vue add bootstrap-vue

main.js modifié et dossier plugins créé

Bootstrapvue -> components liés à Bootstrap

Transmission d’une valeur d’un composant parent à un composant enfant

Exemple

Dans *App.vue*, on inclut un composant disposant de plusieurs attributs :

```vue
<template>
  <div id="app">
    <hello firstname="Marge" lastname="Simpson" from="New York" img="/static/marge.png"></hello>
  </div>
</template>

<script>
  import Hello from './components/chap3/Hello.vue'

  export default {
    name: 'app',
    components: {
      Hello
    }
  }
</script>
```

/src/App.vue

Dans la définition du composant *Hello.vue*, les attributs disponibles sont listés dans *props* et peuvent être accédés / affichés directement via le mécanisme d’interpolation :

```vue
<template>
  <div>
    <img :src="img">
    <h1>Hello {{firstname}} {{lastname}} from {{from}} !!!!</h1>
  </div>
</template>


<script>
  export default{
    data () {...},
    props: ['firstname', 'lastname', 'from', 'img'], //attributs de l'élément HTML
  }
</script>
```

/src/components/chap3/Hello.vue

Dans ce composant, la propriété *img* est utilisée pour valoriser un attribut d'un élément HTML standard, ici l'attribut *src* de *img* : l'interpolation ne fonctionne pas à ce niveau avec Vue.js 2.0. I l faut nécessairement passer par la directive *v-bind:src* (ou plus synthétiquement *:src*) pour que la valeur de la propriété *img* soit appliquée.

On obtient ainsi le résultat suivant :

Hello Marge Simpson from New York !!!!