

VUEJS & CDN

Ressources :

<https://github.com/michelonlineformapro/Projet-10-B-Vue-CLI>

<https://github.com/michelonlineformapro/Projet-10-VueJs-Json-server>

Les + :

<https://github.com/michelonlineformapro/Symfony-API-plateforme-VueJS>

<https://github.com/michelonlineformapro/Prototype-Fullstack-Php-Back-VueCLI-Front>

Objectif :

Comprendre le fonctionnement du Framework VueJs au travers des tests avec vuejs CDN.

Guide en FR :

<https://fr.vuejs.org/v2/guide/>

1 – Origine :

Vue a été créée par [Evan You](#) après avoir travaillé pour Google en utilisant [AngularJS](#) dans un certain nombre de projets. Il a ensuite résumé son processus de réflexion : Je me suis dit : "Et si je pouvais juste extraire la partie que j'aime vraiment dans Angular et construire quelque chose de vraiment léger". Le premier commit de code source du projet était daté de juillet 2013, et Vue a été publié pour la première fois en février suivant, en 2014.

2 – Atouts

Liaison de donnée bidirectionnelles : (data-binding)

Le moindre changement d'une donnée dans le modèle est directement propagé à la vue grâce à un système d'association, les modèles de données sont de simple objet JavaScript et liés à HTML

Des composants réutilisables :

Vuejs permet de décomposer vos interfaces en composants et sous composants réutilisables à l'aide de la syntaxe import et d'export de module

Rapide à charger :

VueJs minifié et compressé fait 33.3ko, c'est le Framework JavaScript le plus léger, il est donc rapide à charger par les navigateurs

Progressif :

Adapté à tous types de projet, utilisable avec son CDN ou CLI (qui se base sur l'empaqueteur web pack, et tous autres outils tels que Vue Router ou Vuex) pour les projets les plus gros.

Flexible :

Séparation de la couche logique métier en JavaScript de la couche représentation HTML/CSS ou organiser le code au sein de la même page pour chaque composant

Des outils d'aide au développement :

Votre IDE et le plugin Vue (auto complétion, générateur de code, analyse, etc...) et des extension navigateur.

Une Grande communauté : (Forum, tuto, doc officiel, etc...)

Courbe d'apprentissage rapide :

Commencer a utilisé VuesJs en créant des instance Vue en JavaScript et en utilisant les directives de base v-on, v-if, v-for, etc...

Développer des application SSR (Server Side Rendering)

Grace Nuxts.js vous développée des SPA (Single page Application) pour un rendu calculé coté serveur pour optimiser le SEO et économisé de la bande passante au premier chargement

Possibilité d'utilisé JSX et Type Script

Une bonne documentation en français : <https://vuejs.org/>

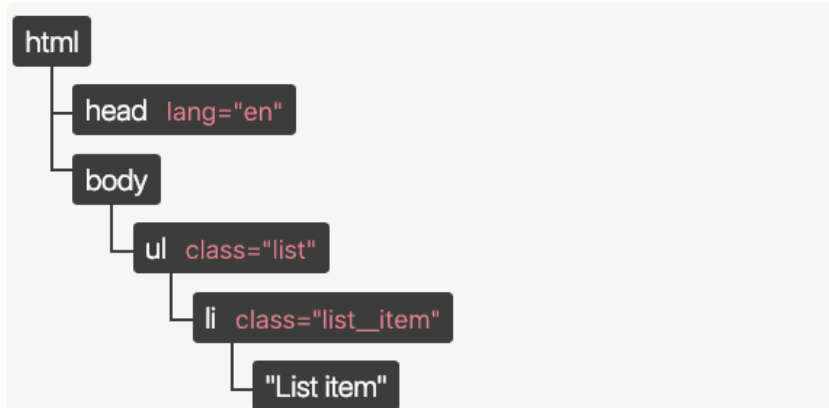
3 – Un Virtual DOM

Pour être réactif VueJs copie votre DOM original en une représentation en simple Objet JavaScript

DOM DE BASE

```
<!doctype html>
<html lang="en">
  <head></head>
  <body>
    <ul class="list">
      <li class="list__item">List item</li>
    </ul>
  </body>
</html>
```

Ce document peut être représenté par l'arbre DOM suivant :



En algorithmique le DOM est composé de plusieurs nœuds chaque nœud est un objet DOM API qui hérite de l'interface Node.

Ici 5 nœuds dit Élément (tag ou balise HTML) (html, head, body, ul, li) qui possède un ou des nœuds de type attribut qui eux même possède des nœuds de type texte

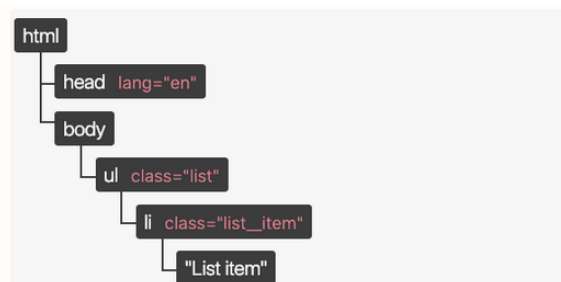
Quand la première spécification du DOM a été publiée en 1998, notre conception et notre gestion des pages web étaient différentes. On s'appuyait beaucoup moins sur les API du DOM pour créer et mettre à jour le contenu de nos pages et celui-ci ne changeait pas au même rythme qu'aujourd'hui.

L'utilisation de méthodes simples comme `document.getElementsByClassName()` est parfaite à petite échelle, mais si nous mettons à jour de nombreux éléments à chaque seconde, les requêtes et mises à jour constantes deviennent coûteuses.

De plus, en raison de la construction même des API, il est en général plus simple de réaliser des opérations coûteuses sur des ensembles plus vastes que d'aller chercher et mettre à jour les éléments spécifiques. Pour revenir à notre liste d'éléments, il peut être plus facile de remplacer toute la liste non ordonnée que d'en modifier quelques éléments.

LA SOLUTION : DOM VIRTUEL

Un DOM virtuel peut être vu comme une copie du DOM original. Cette copie peut être manipulée et mise à jour fréquemment, sans utiliser les API DOM. Une fois toutes les mises à jour effectuées dans le DOM virtuel, nous pouvons voir quels changements doivent être apportés au DOM original et les réaliser d'une façon ciblée et optimisée.



On peut aussi représenter cet arbre sous forme d'objet JavaScript :

```
const vdom = {
  tagName: "html",
  children: [
    { tagName: "head" },
    {
      tagName: "body",
      children: [
        {
          tagName: "ul",
          attributes: { "class": "list" },
          children: [
            {
              tagName: "li",
              attributes: { "class": "list_item" },
              textContent: "List item"
            } // end li
          ]
        } // end ul
      ]
    } // end body
  ]
} // end html
```

On peut voir cet objet comme notre DOM virtuel. Tout comme le DOM original, il s'agit d'une représentation objet de notre document HTML. Mais comme c'est un simple objet JavaScript, nous pouvons le manipuler librement sans toucher au DOM jusqu'à ce que nous en ayons besoin.

Plutôt que d'utiliser un objet pour représenter l'objet entier, il est plus courant de travailler par petites sections du DOM virtuel. Par exemple, nous pourrions travailler sur un composant `list` qui correspondrait à notre élément liste non ordonnée.

Comment ça marche ?

La première chose à faire est de réaliser une copie du DOM virtuel, contenant les changements que nous souhaitons opérer. Puisque nous n'avons pas besoin d'utiliser l'API DOM, nous pouvons simplement créer un nouvel objet.

```
const copy = {
  tagName: "ul",
  attributes: { "class": "list" },
  children: [
    {
      tagName: "li",
      attributes: { "class": "list__item" },
      textContent: "List item one"
    },
    {
      tagName: "li",
      attributes: { "class": "list__item" },
      textContent: "List item two"
    }
  ]
};
```

Cette **copie** est utilisée pour créer ce qu'on appelle une "diff" entre le DOM virtuel original — dans ce cas la liste — et la nouvelle version. Une diff pourrait ressembler à ceci :

```
const diffs = [
  {
    newNode: { /* nouvelle version de list item one */ },
    oldNode: { /* version originale de list item one */ },
    index: /* index de l'élément in la liste des noeuds enfants
du parent */
  },
  {
    newNode: { /* list item two */ },
    index: { /* */ }
  }
]
```

Cette diff fournit des instructions sur ce qui doit être mis à jour dans le DOM. Une fois toutes les diffs réunies, nous pouvons déclencher les modifications du DOM par batches, en ne réalisant que les mises à jour nécessaires.

Par exemple nous pouvons faire une boucle sur les diffs et soit ajouter un nouvel enfant, soit mettre à jour un enfant déjà existant, selon ce que spécifie la diff.

CONCLUSION :

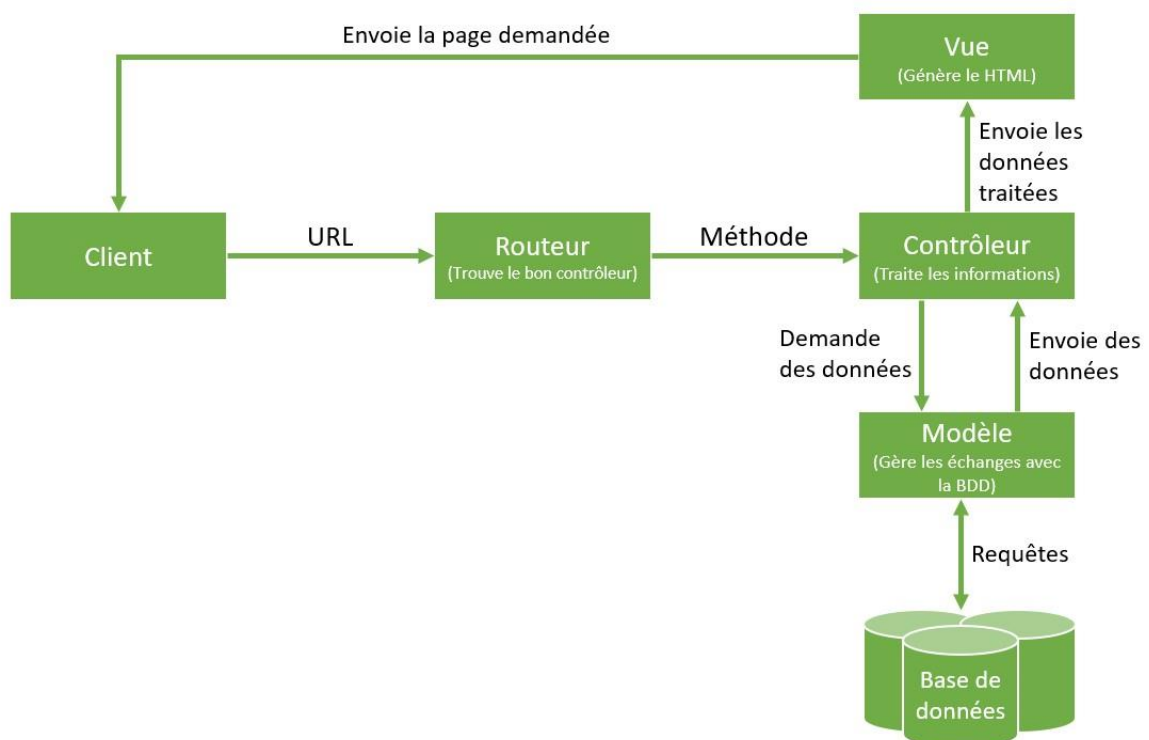
Avec VueJs pas besoin de définir et manipuler des objet JavaScript, les éléments modifier dan l'interface sont reliées à des données déclarée dans une instance de vuejs.

Lorsque l'on modifie des données dans l'instance (let test = new Vue()) VueJs met à jour l'objet copie et l'objet diff et applique automatiquement les changements dans le DOM original de vos page web.

MVC vs MVVM :

Dans les application client-serveur, c'est-à-dire communication via des requête http dans un navigateur de l'utilisateur et un serveur, les trois couche MVC se situe coté serveur ex :

- PHP + MySQL + Apache ou Framework (Symfony, Laravel, Zend, etc...)
- Java EE (Apache Mavean + SQL + Spring)
- Python Django



MVVM (Model – View – View Model)

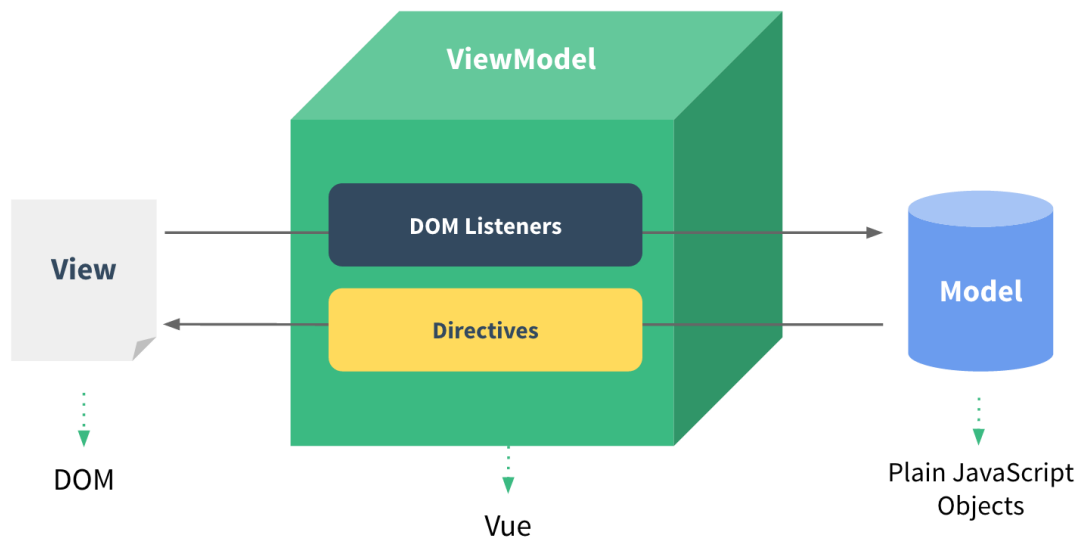
Pour une navigation sur des page web dynamique et riche, les SPA sont des application Monopage avec le patron de conception (Design Pattern) MVVM.

La navigation intervient sans rechargement de la page dont le rendu est recalculé par le serveur.

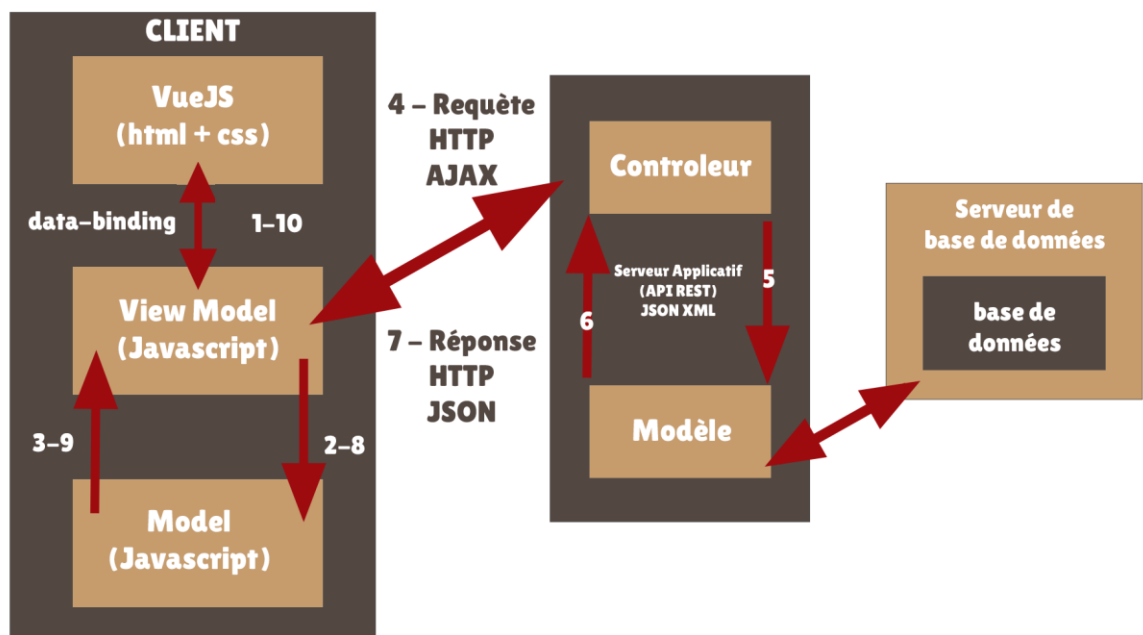
M comme Modèle = domaine application et état sous forme d'objet JavaScript

V comme Vue = couche qui génère HTML et CSS

VM comme Viewmodel = couche qui contient la logique métier dont les données sont directement liées à la vue via le data-binding le tous est génère par le DOM VIRTUEL et consomme également des API REST pour effectuer des opération CRUD via le protocole http.



Exemple de traitement d'un formulaire :



1 – Utilisateur remplit le formulaire et réalise une transmission automatique au ViewModel grâce au data-binding

2 – Ce dernier traite, contrôle, formalise et alimente les objets du model

3 – A la validation la vue lève un évènement JavaScript qui est propagé au ViewModèle, il récupère les données et les envoie via une requête http à une API REST (4), il persiste les données et les envoie via une requête http à une API REST (5 et 6)

7 – API renvoie ensuite une réponse http au navigateur (Vue) puis ViewModel récupère la réponse et met à jour l'état des objets avec l'information enregistrées (8 et 9)

10 – Les données sont mises à jour et indiquent à l'interface utilisateur que le traitement s'est bien passé.

3 – Vue vs React vs Angular

<https://www.npm trends.com/angular-vs-react-vs-vue>

4 – Démarrer un projet avec CDN :

<https://fr.vuejs.org/v2/guide/>

5 – Le cycle de vie d'une instance :

Le cycle de vie possède des hooks (crochets) ce sont des événements levés aux différentes étapes auquel il est possible d'attacher des traitements métier

-beforeCreated() = appelé de manière synchrone à l'initialisation de l'instance de vue (new Vue()) mais pas les observateurs de l'option data ni computed, watch, methods

-created() = appelé une fois que les observateurs du modèle ont été initialisés les options computed, watch, methods sont accessibles mais pas el :

-beforeMounted() appelé juste avant de monter le Template qui a été compilé en fonction du rendu (render()) sur la balise englobante de l'option el :

-mounted() = appelé lorsque la fonction du rendu render() de l'instance a été appelée pour la première fois, le DOM virtuel est créé et monté sur un élément DOM (div id englobante)

-beforeUpdate() = appelée quand les données du modèle changent mais qu'elles n'ont pas encore été rafraîchies sur la page

-updated() = appelé une fois que l'option data a changé et que le DOM virtuel de l'instance ou du composant a été mis à jour

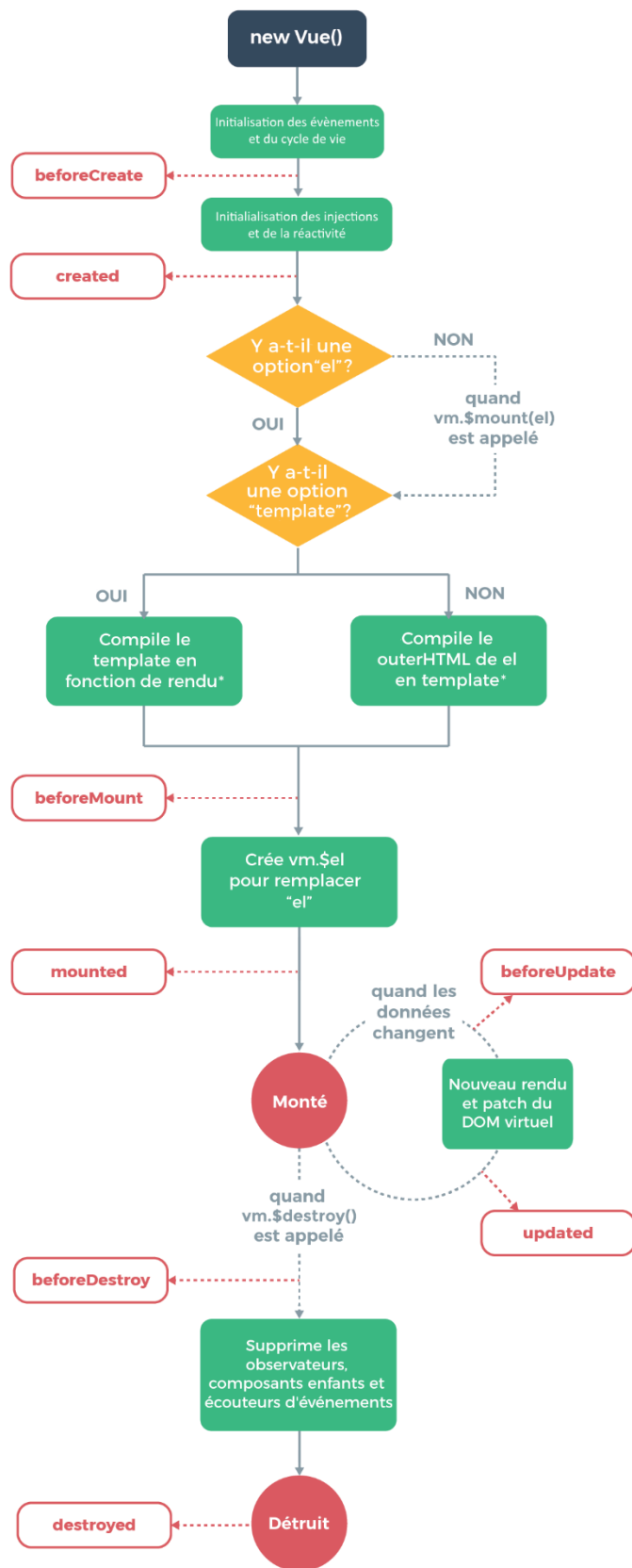
-activated() = appelée quand un composant est gardé en mémoire avec l'attribut keep-alive

-deactivated() = inverse de activated()

-beforeDestroyed() = appelée avant que l'instance new Vue soit détruite

-destroyed() = appelé après la destruction de l'instance new Vue, la mémoire qu'elle occupait est libérée

-errorCaptured() = appelée lorsqu'une exception provenant d'un composant enfant est levée



* la compilation est faite à l'avance si vous utilisez une étape de build, par ex. des composants monofichiers

Toutes les méthodes en rouge peuvent être utilisé dans l'instance de vue

Exemple :

```
4 //Titre de la page h3
5 //instance de Vue
6 let app = new Vue({
7   //prend en valeur un element HTML de type Element (balise ou tag) ou chaine de caractère selecteur CSS
8   el: "#titre",
9   //prend en valeur un objet qui représente le modèle = liste ensemble des données réactive de l'instance de vue
10  //la valeur de ses données a un instant t représente l'état de l'application
11  data: {
12    titre: "1) Un titre réactif"
13  },
14  computed: {
15    //prend en valeur un objet dont les méthodes appelées propriétés calculées retourne un résultats
16    //a partir des données du modèle lorsqu'elle sont modifiées
17  },
18  watch:{
19    //prend en valeur un objet dont les méthodes, appelées observateur sont exécutées lorsque les données du modèles sont modifiée
20  },
21  methods:{
22    //prend en valeur un objet dont les méthodes, pouvant être appelée suite a une action de utilisateur ou a partir d'une autre méthode
23  },
24  mounted: {
25    function(){
26      this.titre = 'Je suis un titre changé !'
27    }
28  },
29  destroyed() {
30    this.titre = '';
31  }
32});
```

Liste des options de l'instance de Vue :

```
//prend en valeur un élément HTML de type Élément (balise ou tag) ou chaine de caractère selecteur CSS
el: "#titre",
```

```
//prend en valeur un objet qui représente le modèle = liste ensemble des données réactive de l'instance de vue
//la valeur de ses données a un instant t représente l'état de l'application
//Ce sont des élément accesseurs et mutateur = getter et setter
data: {
  titre: "1) Un titre réactif"
},
```

```
//prend pour valeur une chaine de caractère HTML qui représenta la vue
template:{
  paragraphe:
    `<p>
      <ul>
        <li>Une liste</li>
      </ul>
    </p>`
},
```

```
computed: {
  //prend en valeur un objet dont les méthodes appelées propriétés
```

```
calculées retourne un résultats
  //à partir des données du modèle lorsqu'elle sont modifiées
//Cet élément est par défaut un getter
//Il est différent de méthodes ce dernier peut mettre en cache un élément
non modifier, il sert donc à économiser des ressources
},
```

```
watch:{
  //prend en valeur un objet dont les méthodes, appelées observateur sont
exécutées lorsque les données du modèles sont modifiée
//Cet élément sert essentiellement au debug
},
```

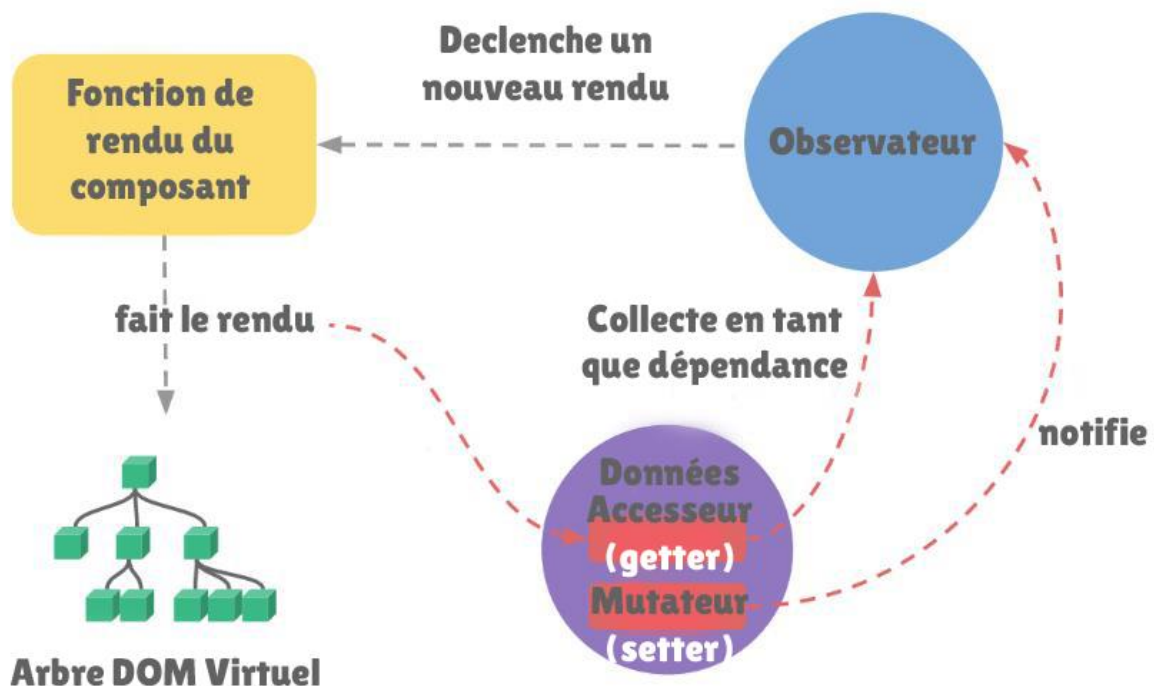
```
methods:{
  //prend en valeur un objet dont les méthodes, pouvant être appelée
suite à une action de utilisateur ou à partir d'une autre méthode
//Cet élément sert donc a appelé des fonctions
},
```

Le système de liaison de donnée :

Lors de l'instance de vue les premiers éléments appelés sont beforeCreated et created :

Ces 2 éléments boucle sur l'objet Data, created convertit les propriétés de Data en accesseur et mutateur (getter et setter) cela permet comme en POO d'encapsuler des propriétés et d'effectuer des traitements.

VueJs utilise donc une fonction statique defineProperty() du type natif Object de JavaScript.



Les directives pour manipuler le DOM :

Des attribut HTML Spéciaux :

- v-text : met à jour les propriété textContext de l'élément pour mettre à jour son contenu avec interpolation {{ message }}
- v-html : met à jour innerHTML de l'élément
- v-show : affiche ou cache un élément en utilisant les propriété CSS display (true/false)
- v-once : ne prend pas de valeur et permet d'afficher un élément ou un composant une seule fois
- v-if : fait un rendu si la condition retourne true
- v-else-if : fonctionne avec v-if pour un rendu conditionnel
- v-else : retourne une valeur quand la condition est fausse
- v-for : permet de boucler et d'afficher plusieurs fois un élément
- v-on : permet d'écouter un évènement natif du DOM (clics souris, touche clavier, survol, etc...)
- v-bind : permet de lier relativement un attribut a une expression JavaScript
- v-model : permet de lier relativement de manière bidirectionnelle au élément des formulaires, comme un champ texte, check box, radio, select etc...
- v-slot : permet de fournir du contenu a un slot donné pour mutualisé des fonctionnalités et / ou injecter du contenu
- v-pre : permet de ne pas compiler un élément du DOM ainsi que ses enfants
- v-cloack : reste sur l'élément jusqu'à ce qu'il soit compilé

Projet 10 VUEJS CLI – CRUD – Json – Server

OBJECTIF :

Installer VueJS de manière globale (via CLI (Command Line Interface), comprendre le système de monofichier.vue, installer vue Router, créer et consommé des API REST pour réaliser des opérations de CRUD simple

1. Installer Vue CLI de manière globale via les gestionnaire paquet npm et/ou yarn
 - a. npm install -g @vue/cli
 - b. # OU (selon si vous avez l'habitude d'utiliser npm ou yarn)
 - c. yarn global add @vue/cli
2. Vérifié l'installation : vue --version
3. Installer un projet : vue create votre-application
4. Choisissez vue version 2 + Babel (ES6 -> pour les anciens navigateurs) et ESLint (ESLint est un outil d'analyse de code statique pour identifier les modèles problématiques trouvés dans le code JavaScript).
5. Un préprocesseur CSS (Sass/SCSS)
6. Il est possible de réaliser ces opérations sur un navigateur web via une interface locale avec la ligne de commande vue ui.

LA STRUCTURE :

1. Node_modules : contient les dépendances
2. Public : contient index.html qui est votre Template de base (et favicon.ico)

3. Package.json : métadonnées + scripts exécuté + dépendances requise
4. .gitignore : les fichiers exclus de votre dépôt (repository)
5. Src : va contenir
 - a. **Assets** - il s'agit du répertoire dans lequel vous placerez les images et les autres ressources obligatoires auxquelles vous devrez peut-être faire référence dans votre application (c'est-à-dire les vidéos, les documents PDF, etc.).
 - b. **Components** - ce répertoire contiendra les composants de notre application. Si vous ne savez pas encore ce que sont les composants, ne vous inquiétez pas. Je traiterai ce sujet dans le prochain chapitre !
 - c. **main.js** - c'est ici que sont définies les options de configuration plus high-level de Vue. Ce fichier peut sembler légèrement différent de ce à quoi nous sommes habitués, mais l'instanciation d'une nouvelle application Vue devrait ressembler à ce que nous avons fait précédemment.
6. Installation de Vue Router : vue add router
 - a. Dans votre fichier de configuration (main.js) Vue CLI à ajouter la propriété router en l'attachant à l'instance de vue et un nouveau dossier est apparu (router/index.js)

```
//Appel du framework vuejs
import Vue from 'vue'
//Appel du point d'entrée App.vue
import App from './App.vue'
import router from './router'

//Import de bootstrap
import { BootstrapVue, IconsPlugin } from "bootstrap-vue";
import axios from 'axios'
import VueAxios from 'vue-axios'

import 'bootstrap/dist/css/bootstrap.css';
import 'bootstrap-vue/dist/bootstrap-vue.css';

Vue.config.productionTip = false

Vue.use(BootstrapVue)
Vue.use(IconsPlugin)
Vue.use(VueAxios, axios)

new Vue({
  router,
  render: h => h(App)
}).$mount('#app')
```

7. Dans votre fichier router/index.js :

- a. path (chemin) - l'URL correspondant au composant ;
- b. name (nom) - le nom de cette route pour l'étiquetage et le débogage ;
- c. component (composant) - le composant qui doit s'afficher lorsque le path est trouvé.

8. Le fonctionnement du router

- a. `<router-view></router-view>` - il définit la zone de la page dans laquelle apparaîtra le composant que nous définissons dans chaque route. Il est particulièrement important si vous imbriquez des vues les unes dans les autres ;
- b. `<router-link></router-link>` - similaire à la balise `anchor` en HTML, ce composant permet également la navigation de l'utilisateur dans l'application. Cependant, il est privilégié par rapport à l'utilisation d'une balise d'ancrage standard, car il dispose de fonctionnalités intégrées, comme le fait qu'il évite de recharger systématiquement la page.

```
import Vue from 'vue'
import VueRouter from 'vue-router'
//Import de la vue

import Accueil from "@/views/Accueil";
import Produits from "@/views/Produits";

import AjouterProduit from "@/views/AjouterProduit";
import DetailsProduit from "@/views/DetailsProduit";

Vue.use(VueRouter)

const routes = [
  {
    //Si url = / appel de la vue Accueil qui appel AccueilComponent
    path: '/',
    name: 'Accueil',
    component: Accueil
  },
  {
    //Si url = /produits appel de la vue Produit qui appel ProduitComponent
    path: '/produits',
    name: 'Produits',
    component: Produits
  },
  {
    //Si url = /ajouter-produit appel de la vue AjouterProduit qui appel AjouterProduitComponent
    path: '/ajouter-produit',
    name: 'AjouterProduit',
    component: AjouterProduit
  },
  {
    //Si url = /produits/:id appel de la vue DetailsProduit qui appel DetailsProduitComponent
    path: '/produits/:id',
    name: 'DetailsProduit',
    component: DetailsProduit
  }
]

//Insatnce du middleware Router (vue add router dans un terminal)
const router = new VueRouter({
  mode: 'history',
  base: process.env.BASE_URL,
  routes
})

export default router
```

9. Les composants monofichier.vue

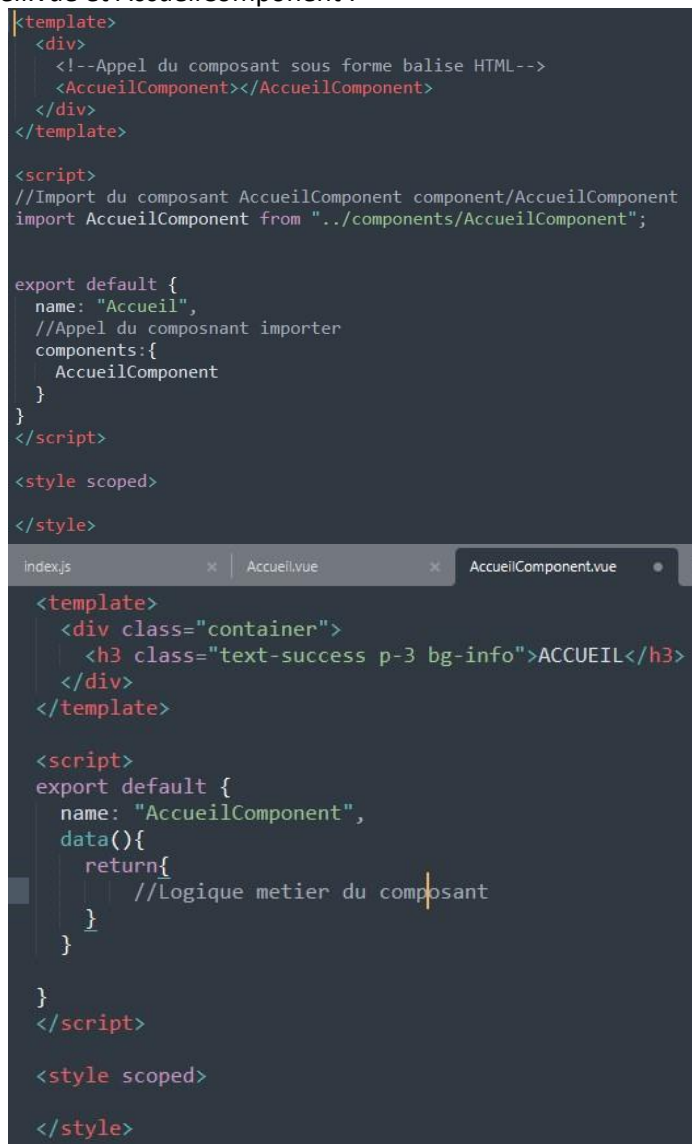
a. La structure des fichiers composant.vue

Les composants monofichiers vous permettent de créer des éléments HTML personnalisés qui encapsulent leur comportement d'une manière facilement maintenable. Ils constituent l'une des meilleures caractéristiques de Vue et peuvent être identifiés par leur extension .vue. Ils sont composés de trois blocs primaires :

- b. Script - où vit votre JavaScript ;
- c. Template - où vit votre HTML ;
- d. Style - où vit votre CSS.

POUR UNE BONNE ORGANISATION DES COMPOSANTS

- e. Pour chaque vue (dans votre dossier views) appeler le composant concerné de votre dossier component
- f. Exemple : le fichier views/Accueil.vue => appel composants/AccueilComponent.vue
- g. Accueil.vue et AccueilComponent :



```
<template>
  <div>
    <!-- Appel du composant sous forme balise HTML -->
    <AccueilComponent></AccueilComponent>
  </div>
</template>

<script>
// Import du composant AccueilComponent component/AccueilComponent
import AccueilComponent from "../components/AccueilComponent";

export default {
  name: "Accueil",
  // Appel du composant importer
  components: {
    AccueilComponent
  }
}
</script>

<style scoped>
</style>
```

index.js x Accueil.vue x AccueilComponent.vue • m

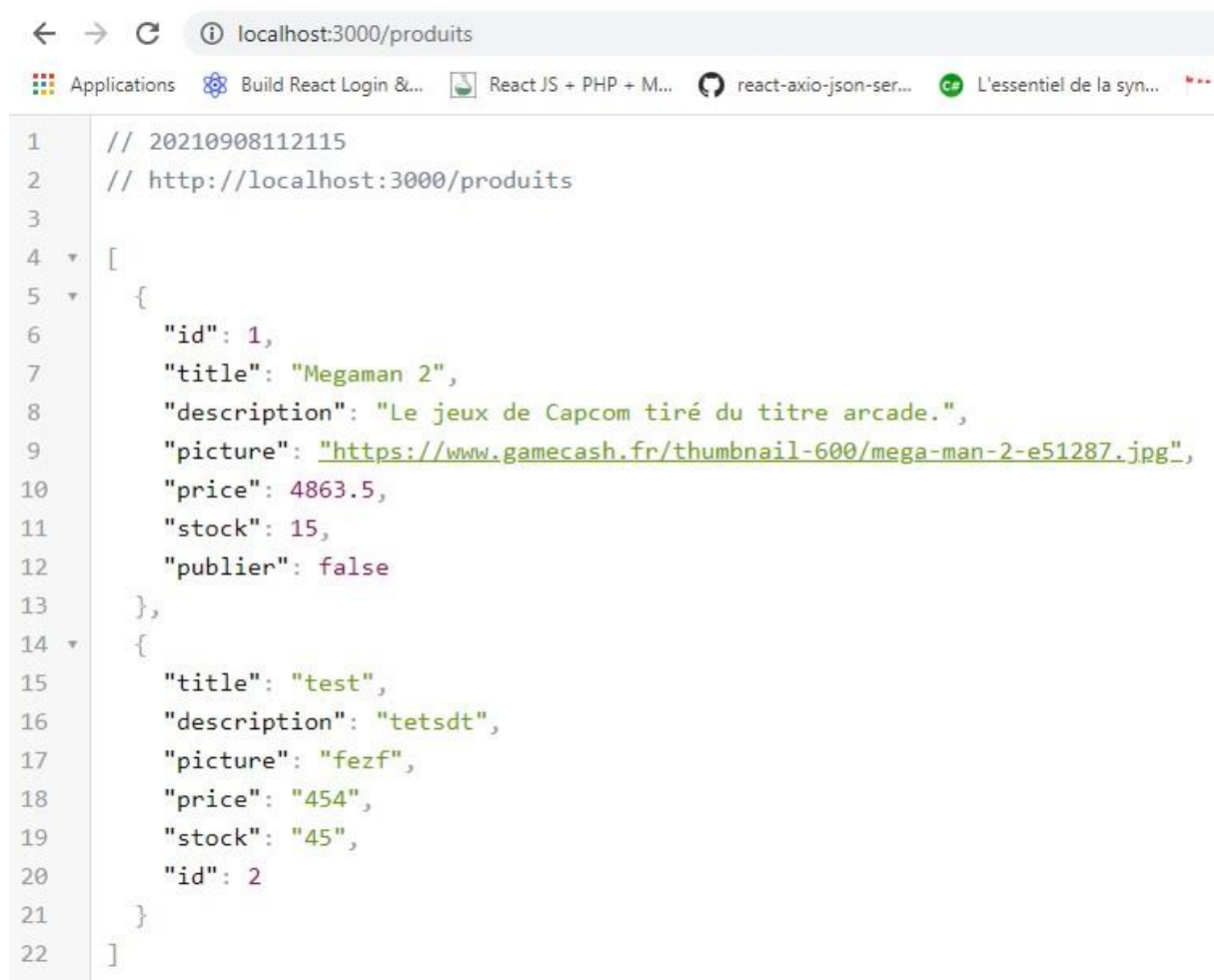
```
<template>
  <div class="container">
    <h3 class="text-success p-3 bg-info">ACCUEIL</h3>
  </div>
</template>

<script>
export default {
  name: "AccueilComponent",
  data() {
    return {
      // Logique métier du composant
    }
  }
}
</script>

<style scoped>
</style>
```

LA GESTION DU CRUD AVEC VUE CLI

10. Créer votre API REST à l'aide de json-server : `npm install json-server`
 - a. Lancer votre serveur (ce dernier va générer un fichier `db.json` à la racine du projet)
 - b. Commande : `json-server --watch db.json` (pour changer de port ajouter le flag : `--port` Numéro du port)
 - c. Nous avons donc : Vue CLI sur le port 8080 (`npm run serve`) et vos données au format json sur le port 3000
 - d. Modifier vos données dans le fichier `db.json` (ici produits)



```
1 // 20210908112115
2 // http://localhost:3000/produits
3
4 [
5   {
6     "id": 1,
7     "title": "Megaman 2",
8     "description": "Le jeux de Capcom tiré du titre arcade.",
9     "picture": "https://www.gamecash.fr/thumbnaill-600/mega-man-2-e51287.jpg",
10    "price": 4863.5,
11    "stock": 15,
12    "publier": false
13  },
14  {
15    "title": "test",
16    "description": "tetsdt",
17    "picture": "fezf",
18    "price": "454",
19    "stock": "45",
20    "id": 2
21  }
22 ]
```

11. Etape 1 : Afficher les produits via une requête http et des promesses
 - a. Installer Axios : `npm i axios`
 - b. Créer un composant `ProduitsComponent` et une vue `Produits` (`Produits.vue` appel `ProduitsComponent` comme vu précédemment)
12. Créer un fichier de configuration http : (à la racine du projet `htt-config.js`)
 - a. Importer Axios
 - b. Exporter un objet `axios.create` + base URL (`json-server`) + options du header requête http


```
index.js  x  Accueil.vue  x  AccueilComponent.vue  ●  Prod

//Import de axios (npm install axios dans un terminal)
import axios from "axios";
//Module a exporter
export default axios.create({
  //Url de jsonserver
  baseURL: 'http://localhost:3000',
  //Options de requête HTTP
  headers:{
    "Access-Control-Allow-Origin" : "*",
    "Content-type": "application/json"
  }
})
```

13. Créer un dossier services et un fichier ProduitsDatasServices.js
 - a. Ce fichier est une classe et donc la passerelle entre votre back json-server et votre front Vue CLI
 - b. Il va effectuer les 6 requêtes http suivante : afficher les produits, afficher un produit, ajouter un produit, mettre à jour un produit, supprimer un produit et rechercher un produit par nom
 - c. Import du fichier http-config.js
 - d. Créer les 6 méthodes qui retournent toutes une route définie dans votre dossier router/index.js
 - e. On utilise les différentes méthodes API REST (GET, POST, PUT, DELETE) en fonction des tâches à exécutées
 - f. Exemple :

```
//Import de fchier de configuration axios et des resuêtes HTTP (GET, POST, PUT/PATCH et DELETE)
import http from '../http-config';

//Creation d'une classe réutilisable
class ProduitsDatasServices{

    //recuperer tous les produits
    recupererTousProduits(){
        return http.get('/produits')
    }

    //Recuperer les datails d'un produits

    recupererProduitParId(id){
        return http.get(`/produits/${id}`)
    }

    //Creer un produit
    ajouterProduit(data){
        return http.post('/produits', data)
    }

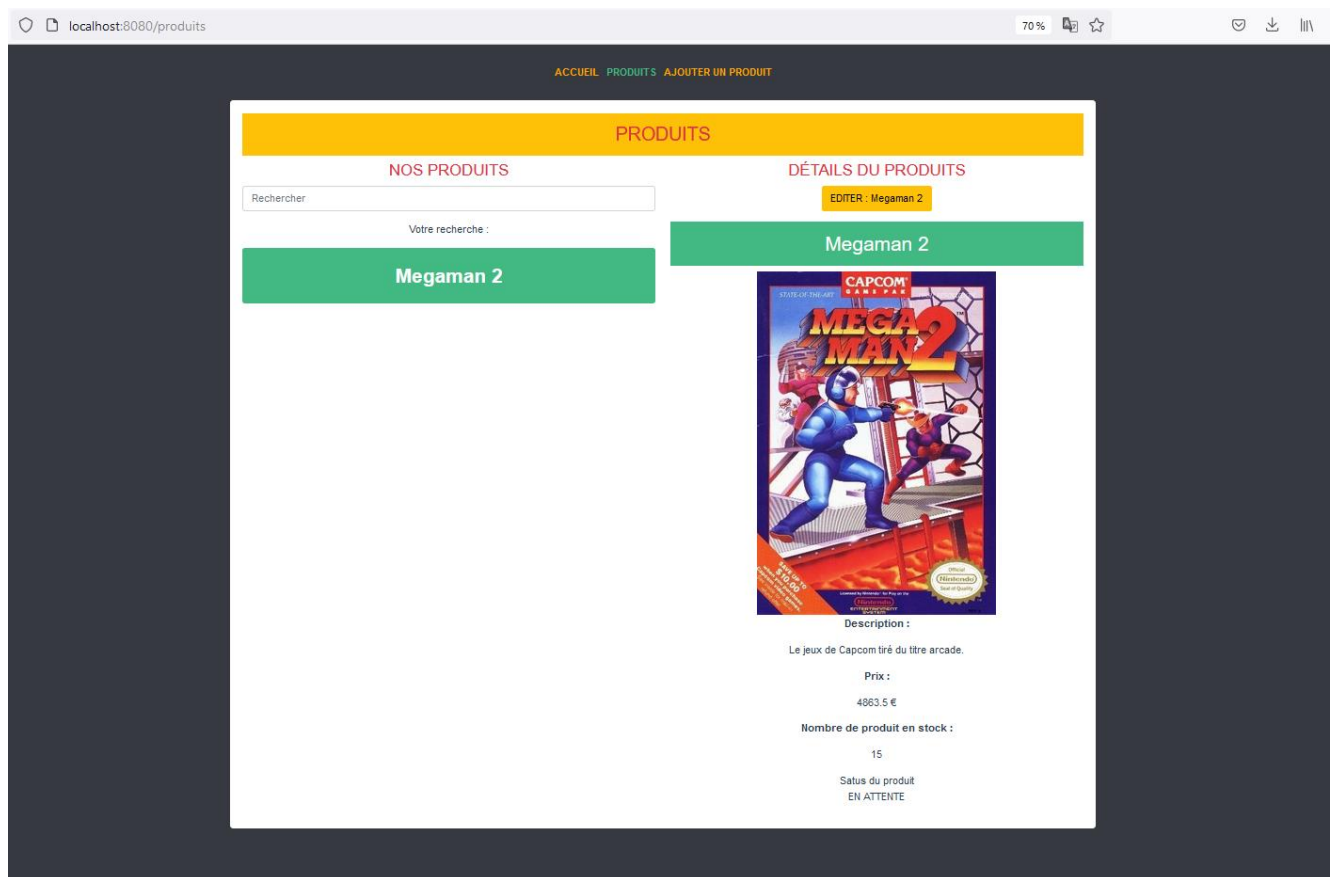
    //mettre a jour un produit
    mettreAJourProduit(id, data){
        return http.put(`/produits/${id}`, data)
    }

    //Supprimer un produit
    supprimerProduit(id){
        return http.delete(`/produits/${id}`)
    }

    //Recherche par titre
    trouverParNom(nomProduit){
        return http.get(`/produits?title=${nomProduit}`)
    }
}

export default new ProduitsDatasServices()
```

RENDU FINAL



14. Dans le fichier ProduitsComponent :
- Dans la partie <script> :
 - Importer votre fichier de services

```
//Import du fichier services/ProduitsDatasServices.js -> classe + methodes CRUD axios requête HTTP
import ProduitsDatasServices from "../../services/ProduitsDatasServices";
```

- Dans la partie réserver au Data-binding (accesseur = getter et mutateur = setter=
- Créer un tableau de produit vide (il sera rempli par le json grâce à votre requête http GET)
- Dans la partie methods :
- Créer une fonction afficherProduits
- Pour la requête http GET : appeler votre fichier de services et la méthode concernée
- Créer une promesse (résolve, reject) qui va remplir votre tableau initialisé dans le data-binding, sinon afficher une erreur et un debug

```

export default {
  name: "ProduitsComponent",
  //Data-binding = getter et setter (accesseur et mutateur)
  data() {
    return {
      //Tableau vide des produits
      produits: [],
      //Produit cliqué = aucun par défaut
      produitCourant: null,
      //Index du tableau[0,1,2,etc...] qui commence a index 0
      produitIndex: -1,
      //Recherche par titre
      rechercher: null
    }
  },
  methods: {
    //Fonction appeler quand le composant est monter (mounted() -> cycle de vie)
    afficherproduit() {
      //Appel de la methode (getAll) de la classe de service Requête HTTP methode GET
      ProduitsDatasServices.recupererTousProduits()
      //Promesse = le tableau de produit est rempli par la requête axios get
      .then(response => {
        //le tableau vide est remplis des données lors de la réponse
        this.produits = response.data
        //debug f12 => console
        console.log(response.data)
      })
      //Sinon on declenche une erreur
      .catch(error => {
        console.log("Pas de produit trouvé :" + error)
      });
    }
  },
}

```

15. Dans la partie <template> :

- Créer une liste et dans votre créer une boucle de parcours via v-for
- Ajouter égale un v-on :click (ou @click) qui appellera la fonction capable d'afficher les détails de votre produit
- Afficher les données grâce l'interpolation {{ produit.title }}

```

<ul class="list-group">
  <!--Listes nom des jeux nes-->
  <!--La classe active bootstrap switch de <li> lors du click-->
  <!--On boucle sur le tableau de produits remplis grace a la requette HTTP (axios) de notre services avec v-for -->
  <!--Au click sur un produit on appel la fonction setProduitConcerner qui recupe le produit dans le tableau grace a son index-->
  <li
    class="list-group-item text-success"
    v-bind:class="{active: index === produitIndex}"
    v-for="(produit, index) in produits" v-bind:key="index"
    v-on:click="setProduitConcerner(produit, index)"
  >
    {{ produit.title }}
  </li>
</ul>

```

16. Pour utiliser votre fonction afficherProduits(), cette dernière est appelée après beforeMount() du cycle de vie soit : mounted()

```

//Dans le cycle de vie Vuejs on appel la fonction afficher quand le composant est monté
//1 -> insatnce de vue main.js
//2 beforeCreate + created (greffe sur le name du composant) + beforeMount et enfin mounted()
//3 lors du changement détecté dans le composant beforeUpdate + updated
mounted() {
  this.afficherproduit()
}
}

```

17. Afficher les détails d'un produit :

- Dans votre data-binding (data()) :
- Ajouté une propriété produitCourant : null
- Ajouter une propriété produitIndex : -1 (Index du tableau [0,1,2,etc...] qui commence à index 0)

```
//Data-binding = getter et setter (accesseur et mutateur)
data() {
  return {
    //Tableau vide des produits
    produits: [],
    //Produit cliqué = aucun par défaut
    produitCourant: null,
    //Index du tableau[0,1,2,etc...] qui commence à index 0
    produitIndex: -1,
    //Recherche par titre
    rechercher: null
  }
},
```

- RAPPEL : Votre liste de produits retourne à l'aide de v-on :click un appel à la fonction setProduitConcerner et 2 paramètres (produit (de la boucle v-for et index)
- Dans la partie methods : Créer la fonction setProduitConcerner(produit, index)
- Assigné les propriétés de votre data-binding au paramètre de la fonction
- Ex : this.produitCourant = produit (idem pour l'index)

```
//Au click sur le produit concerné
setProduitConcerner(produit, index) {
  //Produit courant passe de null au produit cliqué
  this.produitCourant = produit;
  //Index passe de -1 a la position du produit cliqué dans son tableau
  this.produitIndex = index
},
```

18. On découper notre page en 2 blocs horizontaux ;

- A gauche on affiche la liste de jeux par nom et au click on affiche les détails du jeu entier à droite
- On utilise un booléen (produitCourant ? true ou false) au clic sur le titre d'un produit, la fonction setProduitConcerner donne une valeur à produitCourant et ce dernier retourne TRUE
- On utilise la condition v-if pour afficher/cacher un bloc HTML

```

<!--Block de droite pour afficher le contenu du produit cliqué-->
<div class="col-md-6 col-sm-12">
  <!--Si un produit est cliqué on affiche les details-->
  <div v-if="produitCourant">
    <h4 class="text-danger">DÉTAILS DU PRODUITS</h4>

    <!--Bouton editier-->
    <a class="btn btn-warning" v-bind:href="`"/produits/${produitCourant.id}`">EDITER : {{produitCourant.title}}</a>
    <!--FIN-->

    <div class="mt-3">
      <h5 style="background-color: #42b983; font-size: 30px; padding: 20px" class="text-white p-3">{{
        produitCourant.title }}</h5>
    </div>
    <div>
      
    </div>
    <div>
      <p><b>Description :</b></p>
      <p>{{ produitCourant.description }}</p>
    </div>
    <div>
      <p><b>Prix :</b></p>
      <p>{{ produitCourant.price }} €</p>
    </div>
    <div>
      <p><b>Nombre de produit en stock :</b></p>
      <p>{{ produitCourant.stock }}</p>
    </div>

    <div>
      <!--Le produit produit est il publier-->
      <label>Satus du produit</label>
      <p>{{produitCourant.publier ? "PUBLIER" : "EN ATTENTE"}}</p>
      <!--Fin de la condition v-if="produitCourant"-->
    </div>
  </div>
  <!--Si aucun produit est cliqué on affiche ce block-->
  <div v-else>
    <p class="text-danger">MERCI DE SELECTIONNER UN PRODUIT</p>
  </div>
</div>

```

AJOUTER UN PRODUIT

19. Créer une vue AjouterProduits.vue qui appel le composant AjouterProduitComponent.vue (à créer également)
20. Dans AjouterProduitComponent.vue :
 - a. Dans la partie <script> : data-binding
 - b. Créer un objet Produits qui reprend les propriétés de votre fichier db.json

```

export default {
  //Nom du fichier courant
  name: "AjouterProduitComponent",
  //Data binding (accesseur et mutateur getter et setter)
  data(){
    return{
      //Nom de la collections
      produits:{
        "id": "",
        "title": "",
        "description": "",
        "picture": "",
        "price": "",
        "stock": "",
        publier: false
      },
      //Boolean est valide
      valide: false
    }
  },
},

```

- c. Ajouter 2 booléen (publier et valide : false par défaut)
- d. Importer (sous la balise <script>) votre fichier ProduitsDatasServices.json
- e. Dans la partie methods : Créer une fonction de sauvegarde des données du formulaire
- f. Créer un objet donnée et assigné chaque clé a la valeur de votre objet Produits créer précédemment
- g. Créer votre requête http : Appeler vos services et la méthode concernée (ajouterProduit(et data prend la valeur de l'objet données)
- h. Créer une promesse qui retourne la création du nouvel objet et un debug
- i. Sinon retourner une erreur
- j. Si la promesse est tenue (resolve) le booléen : valide passe à true
- k. Ajouter une fonction qui vide les formulaire (réinitialise les valeurs de l'objet Produits)


```

methods:{
  //Fonction sauvegarder le produit ( a appeler au click sur le bouton du formulaire)
  sauvegarderProduit(){
    //On assigne les valeurs initialisé ci dessus au valeur name du formulaire (= $_POST[] en php)
    let donnees = {
      title: this.produits.title,
      description: this.produits.description,
      picture: this.produits.picture,
      price: this.produits.price,
      stock: this.produits.stock
    };
    //Appel de la methode de la casse du fichier services
    ProduitsDatasServices.ajouterProduit(donnees)
    //Creation d'une promesse Js
    .then(response => {
      //On creer l'elements
      this.produits.id = response.data.id
      //Debug
      console.log("Votre produit a été créer : " + response.data)
      //le booléen valide devient true et affiche le message et bouton grace a v-else
      this.valide = true
    })
    //Sinon on declenche une erreur
    .catch(error => {
      console.log(error)
    })
  },

  //Fonction qui vide le formulaire
  viderFormulaire(){
    this.valide = false;
    this.produits = {}
  }
}

```

21. Dans la partie <template> :

- Créer le formulaire avec une condition v-if= !valide (si valide vaut false)
- Ajouter un attribut name à chaque input et un v-model qui sera assigné à chaque éléments de votre objet Produits
- Un bouton qui appelle la fonction sauvegarderProduit
- Un bouton qui appelle la fonction vider les champs quand le produit est ajouté dans un v-else et un message de succès.


```

<template>
  <!--SI la condition valide est false-->
  <div v-if="!valide" class="container">
    <h3 class="text-danger bg-success p-3">AJOUTER UN PRODUIT</h3>
    <!--Formulaire ajout de produit title + description + picture + price + stock-->
    <div class="mb-3">
      <input
        type="text"
        class="form-control"
        id="title"
        required
        v-model="produits.title"
        name="title"
        placeholder="Nom du jeux vidéo"
      />
    </div>

    <div class="mb-3">
      <textarea
        class="form-control"
        id="description"
        rows="5"
        required
        v-model="produits.description"
        name="description"
        placeholder="Description du jeux vidéo"
      ></textarea>
    </div>

    <div class="mb-3">
      <input
        type="text"
        class="form-control"
        id="picture"
        required
        v-model="produits.picture"
        name="picture"
        placeholder="Url du jeux vidéo"
      />
    </div>

    <div class="mb-3">
      <input
        type="number"
        step="0.01"
        class="form-control"
        id="price"
        required
        v-model="produits.price"
        name="price"
        placeholder="Prix du jeux vidéo en €"
      />
    </div>

    <div class="mb-3">
      <input
        type="number"
        class="form-control"
        id="stock"
        required
        v-model="produits.stock"
        name="stock"
        placeholder="Nombre de jeux en stock"
      />
    </div>
    <!--Bouton de validation ajout-->
    <button class="btn btn-outline-warning" @click="sauverProduits">Valider</button>
  </div>

```

ET LE v-else :

```

  <!--Si valide devient true on vide le formulaire-->
  <div v-else>
    <h4 class="text-info text-center">Le produit à bien été ajouter</h4>
    <button class="btn btn-outline-danger" @click="viderFormulaire">Retour</button>
  </div>

</template>

```

Mettre à jour et supprimer un jeu :

ACCUEIL PRODUITS AJOUTER UN PRODUIT

DÉTAILS DU JEUX VIDEO

Megaman 2

Le jeux de Capcom tiré du titre arcade.

https://www.gamecash.fr/thumbnail-600/mega-man-2-e51287.jpg

4863,5

15

Status :
EN ATTENTE

PUBLIER

SUPPRIMER

METTRE A JOUR

Retour

22. Lors du click sur le nom d'un jeu on ouvre un volet sur la droite avec les détails de ce dernier, au click sur le bouton EDITER on appelle une URL et via notre router un nouveau composant

Components/ProduitsComponent.vue

```
<!--Bouton editer-->  
<a class="btn btn-warning" v-bind:href="/produits/${produitCourant.id}">EDITER : {{produitCourant.title}}</a>  
<!--FIN-->
```

Router/index.js

```
{  
  //Si url = /produits/:id appel de la vue DetailsProduit qui appelle DetailsProduitComponent  
  path: '/produits/:id',  
  name: 'DetailsProduit',  
  component: DetailsProduit  
}
```

23. Dans la partie <script> de DetailsProduitsComponent :
- Import de fichier de services : ProduitsDatasServices
 - Dans le data-binding :
 - Initialiser les objets produitCourant : null
 - Dans la partie methods :
 - Créer une fonction produitParID(id) avec id en paramètre
 - A l'aide de vos services appeler la méthode recupererProduitParId(id) (requête http)
 - Créer une promesse : dans resolve assigner produit Courant au donnée json + debug
 - Sinon (reject) déclencher une erreur et un debug

```

<script>
//get by id + update + supprimer un produits
import ProduitsDatasServices from "../../services/ProduitsDatasServices";

export default {
  name: "DetailsProduitComponent",
  data(){
    return{
      //Recuperer le produit courant
      produitCourant: null,
      //Message quand la mise a jour est ok
      message: '',
      publier: status
    };
  },

  methods:[
    produitParId(id){
      //requête HTTP get By ID +classe + methode axios
      ProduitsDatasServices.recupererProduitParId(id)
      //Promesse => le produit corant recup des données grace a id
      .then(response => {
        this.produitCourant = response.data;
        console.log(response.data)
      })
      //Sinon on declenche un erreur
      .catch(error => {
        console.log("Pas de produit pour cet ID !" + error)
      })
    },
  ],
}

```

24. Appeler cette méthode après beforeMounted() (Cycle de vie) soit : mounted()

```

//Dans le cycle de vie Vuejs on déclenche la fonction produitParId quand le composant est monté
//Instance de vue + beforeCreate + created (data-binding getter et setter- puis beforeMounted et monted() intercepte les modifications dur la page (beforeUpdated et updated)
mounted() {
  this.message = '';
  this.produitParId(this.$route.params.id) //localhost:3000/produits/:id
}

```

25. Créer une condition avec v-id=produitCourant, si cette dernière retourne true, afficher votre formulaire de mise à jour (identique au formulaire d'ajout)
- Pour chaque input ajouter un v-model= produitCourant.attribut name
 - A l'aide d'une autre condition v-if, vous avez le choix de publier ou depublier le produit

```

<template>
  <div v-if="produitCourant" class="container">
    <h4 class="text-info text-center">DÉTAILS DU JEUX VIDEO</h4>
    <!-- formulaire de mise a jour -->
    <div class="mb-3">
      <input
        class="form-control"
        type="text"
        id="title"
        placeholder="Nom du jeux"
        v-model="produitCourant.title"
      />
    </div>

    <div class="mb-3">
      <textarea
        class="form-control"
        id="description"
        rows="5"
        placeholder="Description du jeux"
        v-model="produitCourant.description"
      ></textarea>
    </div>

    <div class="mb-3">
      <input
        type="text"
        class="form-control"
        id="picture"
        required
        v-model="produitCourant.picture"
        name="picture"
        placeholder="Url du jeux vidéo"
      />
    </div>

    <div class="mb-3">
      <input
        type="number"
        step="0.01"
        class="form-control"
        id="price"
        required
        v-model="produitCourant.price"
        name="title"
        placeholder="Prix du jeux vidéo en €"
      />
    </div>

    <div class="mb-3">
      <input
        type="number"
        class="form-control"
        id="stock"
        required
        v-model="produitCourant.stock"
        name="stock"
        placeholder="Nombre de jeux en stock"
      />
    </div>
  </div>

```

Les boutons de mise à jour :

```

<div class="mb-3">
  <label>Status :</label>
  <p>{{produitCourant.publier ? "PUBLIER" : "EN ATTENTE"}}</p>
</div>
<!--Booleen publier / depublier-->
<div class="mb-3">
  <button
    type="button"
    class="btn btn-outline-danger"
    v-if="produitCourant.publier"
    @click="miseJourEtPublication(false)"
  >
    DEPUBLIER
  </button>
  <button v-else
    type="button"
    class="btn btn-outline-success"
    @click="miseJourEtPublication(true)"
  >
    PUBLIER
  </button>
</div>

```

26. Le fonction mettreJourEtPublier(status) :

- Créer un objet données pré-rempli grâce au v-model des inputs
- Créer votre requête http à l'aide de vos services et de la méthode concernée (passer l'id et l'objet données)
- Créer une promesse qui retourne le statut de vos changements et un debug
- Sinon une erreur et un debug

27. La fonction de mise à jour du produit ()

- Cette dernière est appelée lors du clic sur le bouton du formulaire (@click= "majProduit()")
- Exécuter une requête http à l'aide de vos services et de la méthode concernée (en paramètre passer id et le produit Courant)
- Dans une promesse afficher le debug et un message de succès sinon une erreur et un debug

```

//Cette fonction est appelée au @click sur le bouton mettre a jour du formulaire
majProduit(){
  //Appel du fider de services crud requête HTTP
  ProduitsDatasServices.mettreAJourProduit(this.produitCourant.id, this.
    produitCourant)
  .then(response => {
    console.log(response.data)
    //Si ca marche on affiche le message de réussite
    this.message = "Le jeux video a bien été mis a jour !"
    //ici on peu faire une redirection avec un setTimeout
  })
  .catch(error => {
    console.log("Erreur lors de la mise a jour " + error)
  })
},

```

```

<!--bouton mettre a jour-->
<div class="mb-3">
  <button
    class="btn btn-outline-warning"
    @click="majProduit"
  >
    METTRE A JOUR
  </button>
</div>

```

28. La fonction supprimer un produit :

- Exécuter une requête http à l'aide de vos services et appeler la méthode concernée avec le produit Courant et son id en paramètre
- Créer une promesse qui effectue une redirection à l'aide de \$router.push(URL)
- Sinon une erreur et un debug
- Cette fonction est appelée au clic sur le bouton supprimer (@click="supprimerProduit")

```

//Cette fonction est appelée au click sur le bouton supprimer
supprimerProduit(){
  //requête http a partir de la classe ProduitsDatasService.js -> methode supprimer axios
  //On recupere id a l'aide du produit courant soit /produit/:id dans le router ex: /produit/id de la collection json
  ProduitsDatasServices.supprimerProduit(this.produitCourant.id)
  //Creation d'une promesse pour la reponse
  .then(response => {
    //Debug f12
    console.log(response.data)
    //Redirection vers la page des produit a l'aide de Vue router grace a $router et push()
    this.$router.push('../produits')
  })
  .catch(error => {
    //Sinon on declenche une erreur
    console.log("impossible de supprimer ce jeux " + error)
  })
}
}

```

Le bouton supprimer

```

<!--bouton supprimer au click on lance la fonction supprimerProduit ci-dessous-->
<div class="mb-3">
  <button
    class="btn btn-outline-info"
    v-on:click="supprimerProduit"
  >
    SUPPRIMER
  </button>
</div>

```

VOTRE CRUD VUEJS CLI ET JSON SERVER EST DESORMAIS TERMINÉ

