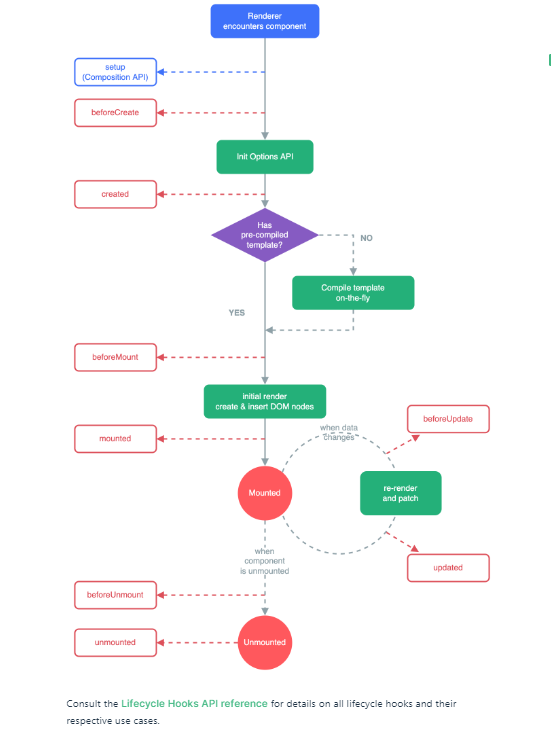
**Rabah Daguelou VueJs**

**Projet Cocktail**

****

**Passer une regex à un paramètre d’une url :**

Recevoir uniquement un id en chiffres

{path:'users/edit/:id(\\d+)', name:'UserEdit', component: Admin.UserEdit, props: true},

**Redirection vers une vue :**

* L’on peut dans ce cas rediriger l’utilisateur vers une autre page si l’url n’est pas correcte !

path: '/:pathMatch(.\*)\*', redirect: '/',

**Récupération des paramètres dans les url**

**1/ La méthode params pour passer des paramètres dynamiques**

export default {

    name: 'UserEdit',

    data() {

        return {

            test: '',

        }

    },

    mounted () {

        console.log (this.$route.params.id)

        this.test = this.$route.params.id

    },

    updated () {

        console.log (this.$route.params.id)

        this.test = this.$route.params.id

    }

**2/ La méthode props :**

* Dans le router, spécifier de passer des propriétés (props) dans l’url (paramètres)
* Activer la fonction :

{path:'users/edit/:id', name:'UserEdit', component: Admin.UserEdit, props: true},

* Déclarer les propriétés dans la vue dans un tableau en string:

 props: ['id'],

* Utiliser la props dans le template qui reconnait la variable

<p> Numéro utilisateur: {{ id }} </p>

**Empêcher l’envoi du formulaire :**

* Intercepter l’action du bouton submit pour la bloquer et stopper le rechargement de la page
* Exécuter la fonction « login » au moment du submit

<form @submit.prevent="login">

**Fermeture d’une seule route :**

**Avec les hooks du router :**

* Chaque route du routeur bénéficie d’un hook (événement) (beforeEnter(to, from, next)
* Fermer une route :

path: '/login', name: Login, component: Login, beforeEnter: (to) => {

      return false;

**Avec un fichier guard :**

* Créer un dossier/fichier \_helpers/auth-gaurd.js qui va vérifier l’existence d’un token
* import router from "@/router"
* export function authGuard(to) {
* let token = localStorage.getItem('token')
* console.log ("Token:", token)
* if(token) {
* return true
* }
* router.push('/')
* }
* Importer le guard dans le router et l’appeler sur la route

 path: '/login', name: Login, component: Login, beforeEnter: authGuard

**Fermeture de plusieurs routes enfants : (path composé)**

**Fermer toutes les routes de Admin :**

**Avec beforeEnter**

path:'/admin', beforeEnter: authGuard,

**Avec beforeEach**

* beforeEach se détecte sur toutes les routes enfants contrairement à « beforEnter »
* Insérer la fonction suivante après la déclaration du router dans router/index.js
* // Fermeture des routes Admin
* router.beforeEach((to, next) => {
* console.log ("To.matched[0].name:", to.matched[0].name)
* if (to.matched[0].name == 'admin') {
* console.log (' Vous voulez aller sur Admin')
* authGaurd()
* }
* next()
* })

**Avec les meta :**

* Les meta servent à coller une sorte d’étiquette à une route pour adapter un comportement
* L’inconvénient c’est qu’elles fonctionnent sur une seule route contrairement à beforeEach
* Insérer une meta d’authentification « requiresAuth » à true

{path:'dashboard', name:'Dashboard', component: Admin.Dashboard, meta: {requiresAuth: true}},

**La clé to de la route Dashboard dans la console :**

**Une image contenant texte

Description générée automatiquement**

**Une image contenant texte

Description générée automatiquement**

**Gérer la connexion :**

* Le formulaire est branché sur l’évènement submit avec un prevent default pour stopper l’envoi et le rechargement de la page et appeler plutôt la méthode login() :

<form @submit.prevent="login">

 methods: {

login() {

}

}

* Les champs sont branchés avec les données ( fonction data ) grâce à v-model

<input type="user\_email" id="email" v-model="user.email">

 data () {

      return {

user: {

email: "",

password: "",

}

}

},

**Avec la méthode Fetch :**

Une image contenant texte

Description générée automatiquement

**Avec Axios :**

* Créer un fichier \_services/caller\_service.js et y paramétrer axios
* import axios from ('axios');
* const Axios = axios.create ({
* baseUrl: 'http://localhost:8888'
* })
* export default Axios;
* Créer un fichier \_services/account\_service.js et y gérer la connexion, le token, …. (voir fichier)
* Créer un fichier \_services/index.js qui va exporter tous ce qui est service

export \* from './account\_service'

* Importer les services dans la vue login.vue (par exemple)

import { accountServices } from '@/\_services'

* Faire un appel API

login() {

            accountServices.login(this.user)

Gérer les requêtes :

* Voir fichier \_services/caller\_service

Gérer les requêtes users :

* Voir fichier \_services/user\_services.js

**L’intercepteur d’axios :**

L’intercepteur d’axios choppe la requête avant qu’elle sorte de l’applicatif, injecte le token dans la requête et relache la requête…

Réaliser cette opération dans le fichier \_services/caller\_services.js :

// Intercepter les requêtes

Axios.interceptors.request.use(request => {

    // Si connecté on ajoute le token à la requête

    if (accountServices.islogged()) {

        request.headers.Authorization = 'Bearer ' + accountServices.getToken()

    }

    // Relâcher la requête

    return request

})

**Insérer un bouton de déconnexion :**

* Créer un bouton dans components/AdminHeader.vue avec @click=’disconnection’

<button @click="disconnection">  Se déconnecter </button>

* Importer accountServices :

import {accountServices} from '@/\_services'

* Appeler la méthode ‘disconnection’
* methods: {
* disconnection() {
* accountServices.logout();
* this.$router.push('/')
* }
* }

**Gérer une fin de session ( Token expiré)**

* Fichiers \_services/user\_services: importer le router

import router from '@/router'

* Intercepter la réponse grâce à axios :
* // Intercepter les réponses API
* Axios.interceptors.response.use(response => {
* return response
* }, error=> {
* if (error.status == 401) {
* accountServices.logout()
* router.push('/login')
* }
* })

**Affichage de la liste utilisateurs**

Fichier admin/users/userIndex.vue :

<tr v-for="user in users" :key="user.id">

                    <td> {{ user.id }}</td>

                    <td> {{ user.first\_name }}</td>

                    <td> {{ user.last\_name }}</td>

                    <td> {{ user.pseudo }}</td>

                    <td> {{ user.updatedAt }}</td>

                </tr>

data () {

        return {

            users : [],

        }

    },

mounted() {

        userServices.getAllUsers

        .then (res => {

            console.log ("users:", res.data);

            this.users = res.data

        })

        .catch (err => console.log(err))

    },

**Formater la date :**

* La méthode **split()** divise une [chaîne de caractères](https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Global_Objects/String) en une liste ordonnée de sous-chaînes, place ces sous-chaînes dans un tableau et retourne le tableau. La division est effectuée en recherchant un motif ; où le motif est fourni comme premier paramètre dans l'appel de la méthode.
* user.updatedAt : 
* user.updatedAt.split('T')[0].split('-').reverse().join('/') :

*Mais, en Vuejs, le template doit être simple est déclaratif (Afficher les éléments et non les y traiter). C’est pourquoi il faut utiliser les « Computed » pour le traitement.*

L’on peut utiliser également des méthodes comme suit:

 methods: {

        dateFormat (date) {

            return date.split('T')[0].split('-').reverse().join('/')

        }

    },

<td> {{ dateFormat(user.updatedAt) }}</td>

Encore une fois mieux vaut utiliser les computed car les méthodes ne sont appelées qu’une seule fois alors que les computed (mises en cache) sont réévaluées à chaque changement de variables

* Les computed ne sot pas des méthodes, l’on ne peut pas y greffer des paramètres comme les méthodes.

**Utiliser Computed :**

* Avec la computed « dateFormat », je prépare mon tableau de dates formatés et je les affiche en utilisant l’index
* La méthode **map()** crée un nouveau tableau avec les résultats de l'appel d'une fonction fournie sur chaque élément du tableau appelant.
* Afficher le nombre d’utilisateurs trouvés et formater la date :

<caption> {{ numberOfUsers }} utilisateurs inscrits !</caption>

<tr v-for="(user, index) in users" :key="user.id">

<td> {{ dateFormat[index] }}</td>

computed :{

        numberOfUsers () {

            return this.users.length

        },

        dateFormat () {

            return this.users.map (u => u.createdAt.split('T')[0].split('-').reverse().join('/'))

        }

    },

**Editer un utilisateur en particulier :**

* L’on peut brancher le click aussi bien sur toute la ligne que sur le nom ou l’id par exemple

<tr v-for="(user, index) in users" :key="user.id" class="edit" @click="goEdit(user.id)">

* Créer la méthode en utilisant la méthode URL :

goEdit(id) {

            console.log (id)

            this.$router.push(`/admin/users/edit/${id}`)

        }

* Utiliser la propriété « name » du router :

goEdit(id) {

            this.$router.push({name:'userEdit', params:{id:id}})

        }

**Modifier un utilisateur :**

* Créer un formulaire des données à modifier et lier les champs avec le user récupéré avec « mounted »
* <div class="formFroup">
* <label for="last\_name"> Nom : </label>
* <input type="last\_name" id="last\_name" v-model="user.last\_name">
* </div> <br>
* <div class="formFroup"> <br>
* <button type="submit" class="button" @click="modify"> Modifier </button>
* </div>
* Créer la méthode « modify »

// Modifier l'utilisateur

    methods: {

        modify() {

            console.log ("Utilisateur à modifier:", this.user )

            userServices.modifyOneUser(this.user)

                .then (respose => {

                    console.log (' Response:', respose)

                    this.$router.push({name: 'usersList'})

                })

                .catch (error => {

                    console.log ("Error:", error)

                })

        }

    }

**Suppression d’un utilisateur :**

* Ajouter un bouton de suppression sur chaque utilisateur dans userIndex.vue
* Lier l’élément au click à une méthode

<td > <span class="del\_btn" @click="deleteUser(user.id, index)">x</span>  </td>

* Ajouter la méthode de suppression dans \_services/user\_services.js et l’exporter
* // -5 Supprimer un utilisateur
* let deleteOneUser = (userId) => {
* return Axios.delete ('users/trash/'+userId)
* }
* Créer la méthode de suppression :
* deleteUser(userId, index){
* console.log ('userId:', userId, 'Index:', index)
* userServices.deleteOneUser(userId)
* .then (response => {
* console.log ("Response:", response)
* this.users.splice(index,1)
* })
* .catch (error => {
* console.log ("Error:", error)
* })
* }

**Editer / Modifier un cocktail : (2ème méthode** *un seul composant***)**

* Le lien :

<li><router-link to='/admin/cocktails/edit'> Cocktail add </router-link></li>

* Rendre le paramètre optionnel pour naviguer de l’édition ou l’ajout au même composant:

**Le point d’interrogation**

{path:'cocktails/edit/:id(\\d+)?', name:'cocktailEdit', component: Admin.CocktailAdd, props: true},

**Utiliser un watcher :**

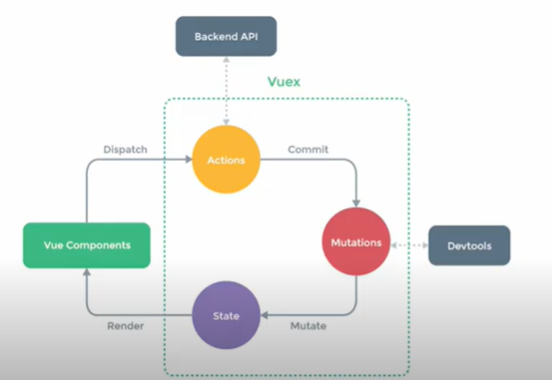
* Le watcher peut déterminer si l’on est dans « *CocktailEdit.vue* » ou dans ‘*CocktailAdd.vue* »
* Importer watch

import { watch } from 'vue'

* Le setup se met en place au début du composant et surveiller les props avec le watcher :
* setup (props) {
* console.log ("Props:", props)
* watch (props, (value, old) => {
* console.log ("value:", value, "old:", old)
* })
* },

**Utilisation du store Vuex :**

Distribue des données aux composants interdépendants ou même complètement indépendants.



Le store est un objet disponible et accessible sur l’entièreté de l’application (fichiers services, router, composants…)

// Fichier du store src/store/index.js

import { createStore } from 'vuex'

export default createStore({

  state: {

  },

  getters: {

  },

  mutations: {

  },

  actions: {

  },

  modules: {

  }

})

**State :** Contient les **variables.** Ces variables ne sont pas forcément synchronisées avec les vues contrairement aux mutations et getters. ***Correspond à*** *« data »* ***dans les composants.***

**Getters :** Les accesseurs. Méthodes que l’on peut appeler de n’importe où pour **accéder** aux variables du state. ***Correspond à*** *« computed »* ***dans les composants.***

**Mutations : *Les modificateurs synchrones***.Synchronisées avec les états des variables du state que l’on peut **modifier.** Commiter une modification que l’on pousse dans le store/state. ***Correspond à*** *« methods****» synchrones dans les composants.***

**Actions : *Les modificateurs Asynchrones.*** On peut s’y coller des fonctions généralistes. (Tels que les services, les appels API). ***Correspond à*** *« methods »* ***asynchrones dans les composants.***

**Modules :** Assez spécifique. Comme un state dans le state. Un sous-state qui contient ses propres state, getters…

**Remarques :**

* La voie normale est qu’un composant demande à un getter de lui fournir une data du store. *Dans les nouvelles versions l’on peut accéder directement aux data du state sans passer par un getter.*
* Pour commiter une modification, l’on utilise une mutation qui modifie une variable dans le state et peut retourner une valeur (variable modifiée) qui sera synchronisée avec les composants utilisant la même variable.
* Plutôt utiliser les actions pour les fonctions asynchrones (fetch, promesse,…) et les mutations pour les fonctions synchrones (modifier une variable, un string, …)
* Avec les mutations on **commit** et avec les actions l’on **dispatch**
* Utiliser l’extension chrome Vue Devtools

**Exemples :**

1. **Le state**

**Accéder à une variable du state depuis un composant**

  state: {

    lounis: "Lounis"

  },

**Views/public/publicLayout.vue :**

 <div class="public">

              {{ aff }}

data () {

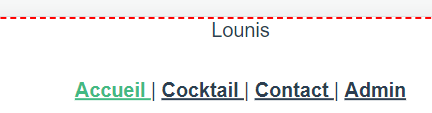
        return {

            aff: this.$store.state.lounis

        }

    }

**Résultat :**

****

1. **Les getters :**

* Fonctions pour récupérer proprement une valeur du state et la synchroniser dans tous les composants qui l’utilisent.
* state: {
* lounis: "Lounis"
* },
* getters: {
* getLounis(state) {
* return state.lounis
* }
* },

**Views/public/publicLayout.vue**

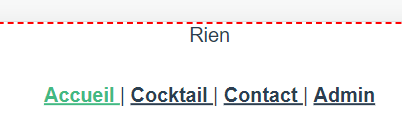
data () {

        return {

            aff: "Rien"

        }

    }

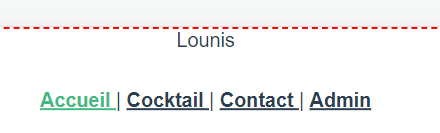


***A/ Modifier la valeur de la variable aff au montage avec un getter***

 mounted () {

        this.aff = this.$store.getters.getLounis

    }

******

***B/ Utiliser mapGetters pour synchroniser entièrement les données***

***mapGetters*** permet d’importer dans le composants tous les getters dont l’on a besoin

* Importer mapGetters dans le composant

import { mapGetters } from 'vuex';

* Insérer les getters dont j’ai besoin et connecter le composant au state par le getter. **La syntaxe** : ***…mapGetters([‘getter1’, ‘getter2’,…])***

computed: {

        ...mapGetters(['getLounis'])

    }

* Appeler la fonction dans le template

 <div class="public">

              {{ getLounis }}

* L’on a plus besoins d’appeler le getter dans « mounted » ni de la variable dans le « data »
* Si l’on modifie la variable dans le state, le composant utilisant le getter sera synchronisé

**Remarques :**

* L’on appelle un getter qui est une fonction telle une propriété et non une fonction

this.aff = this.$store.getters.getLounis

ou avec mapGetters selon la syntaxe : ***…mapGetters([‘getter1’, ‘getter2’,…])***

C’est une fonction qui reçoit un Array contenant les getters en String

computed: {

        ...mapGetters(['getLounis', ‘getLounis2’])

    }

* Le getter peut prendre des paramètres avec la méthode style access (méthode particulière)
* Pour récupérer un élément précis dans un tableau du state

Exemple :

state: {

    users: [

      {

        id: 0,

        firstName: 'Lounis',

        lastName: 'Daguelou'

      },

      {

        id: 1,

        firstName: 'Lina',

        lastName: 'Bouallouche'

      },

    ]

  },

  getters: {

    getLounis(state) {

      return state.users[0].firstName

    },

// Un getter avec paramètres

    getLina: (state) => (id) => {

      return state.users[id].firstName + ' ' + state.users[id].lastName

    }

**Template :** L’on passe le paramètre

  <p> {{ getLina(1) }} </p>

1. **Les mutations**

Permettent de modifier des données du state de manière synchronisée

1. **Utiliser Commit :**

**A/ Sans paramètres : (sans envoyer de données)**

* Créer la mutation dans le satate
* mutations: {
* changeLounis (state) {
* return state.users[0].firstName = ‘Rabah’
* }
* },
* Commiter la mutation à partir d’un composant

**Template :**

 <p> {{ getLounis }} <span> <button @click="changeName"> Changer </button></span> </p>

**Script :**

methods: {

      changeName () {

        this.$store.commit('changeLounis')

      }

    }

**B/ Avec paramètres (Envoyer des données)**

* Créer la mutation dans le satate
* mutations: {
* changeLounis (state, value) {
* return state.users[0].firstName = value
* }
* },
* Commiter la mutation à partir d’un composant

**Template :**

 <p> {{ getLounis }} <span> <button @click="changeName"> Changer </button></span> </p>

**Script :**

methods: {

      changeName () {

        this.$store.commit('changeLounis', ‘Rabah’)

      }

    }

**C/ Envoyer un objet : Par convention on l’appelle payload**

* Créer la mutation dans le satate
* mutations: {
* changeLounis (state, payload) {
* return state.users[0].firstName = payload.firstName
* }
* },
* Commiter la mutation à partir d’un composant

**Template :**

 <p> {{ getLounis }} <span> <button @click="changeName"> Changer </button></span> </p>

**Script :**

methods: {

      changeName () {

        this.$store.commit('changeLounis', {firstName: 'Lina'})

      }

    }

1. **Utiliser mapMutations :**

* Importer mapMutations,

import { mapGetters, mapMutations } from 'vuex'

* Inclure les mapMutations dans methods
* Appeler la mutation (en passant une variable ou un objet (payload)).

methods: {

      ...mapMutations(['changeLounis']),

      changeName () {

        this.changeLounis({firstName: 'Lina'})

      }

    }

1. **Les actions**

* On crée une action dans le store
* actions: {
* modify(context) {
* console.log("context:", context)
* }
* },
* Le context contient quasiment tout (les getters, le state les mutations, …)

**Exemple**s : je peux

Lancer une modification,

Commiter une mutation,

actions: {

    modify(context) {

      console.log("context:", context)

      context.state.users[0].firstName = 'Loulou'

      context.commit('changeLina', {lastName:"Daguelou"})

    }

  },

**1/ Utiliser dispatch :**

* Appeler l’action depuis un composant ( Les actiions se rangent dans « methhods : »

  action () {

        this.$store.dispatch('modify')

      }

**2/ Utiliser mapActions :**

 actions: {

    changeLina(context, payload) {

      context.state.users[0].firstName = 'Loulou'

      context.commit('changeLina', payload)

    }

  },

* Importer mapActions
* Envoyer une action depuis un composant
* methods: {
* ...mapMutations(['changeLounis']),
* ...mapActions(['changeLina']),
* changeName () {
* this.changeLounis({firstName: 'Lina'})
* },
* action () {
* this.changeLina({lastName:"Daguelou"})
* }
* }
* Pour la même action dans le store je peux écrire : (j’utilise uniquement le commit du context)
* actions: {
* changeLina({commit}, payload) {
* context.state.users[0].firstName = 'Loulou'
* commit('changeLina', payload)
* }
* },

**Gestion des erreurs API**

* Créer un composant qui va se greffer partout dans l’app