TP1 : Vue.js - Dr Mad et son laboratoire

0°/ Installer l'environnement de développement

Si ce n'est déjà fait, installez les outils nécessaires à la mise en place de votre environnement de développement :

- **Node.js**
- **npm** (le gestionnaire de paquets de Node)
- **serve** (pour exécuter l'application Vue.js en local)

Consultez l'article Installation de vue-cli & config. idea pour plus de détails.

1°/ Créer le projet avec vue-cli + vue-router + vuex

1.1°/ Créer un canevas de base

Dans un terminal, tapez la commande suivante pour créer un projet avec Vue CLI :

```
vue create drmad
```

Un menu interactif apparaîtra :

```
Vue CLI v4.5.14


  New version available 4.5.14 → 5.0.8
  Run npm i -g @vue/cli to update!
```

- Sélectionnez **"Manually select features"** pour personnaliser les fonctionnalités du projet.
- Choisissez les options suivantes :
  - **Vue version** : Vue 2.x (Important, ne choisissez pas Vue 3.x)
  - **Babel**
  - **Vue Router** (pour la gestion des routes internes)
  - **Vuex** (pour la gestion de l'état global de l'application)
  - **Linter / Formatter** (pour le formatage du code)
  - **History mode** pour le Router
  - **ESLint avec error prevention only**

Une fois ces options sélectionnées, appuyez sur **Entrée** pour lancer la création du projet. Vue CLI va ensuite télécharger les dépendances nécessaires et créer les fichiers par défaut du projet.

1.2°/ Tester le projet

Pour tester le projet, lancez la commande suivante dans le terminal pour démarrer le serveur de développement :

```
npm run serve
```

Par défaut, l'application sera disponible à l'adresse http://localhost:8080. Vérifiez que l'application fonctionne correctement.

Si tout est en ordre, vous pouvez arrêter le serveur avec **Ctrl+C**.

1.3°/ Passer le projet sous Idea

Si vous utilisez un IDE comme **IntelliJ IDEA**, vous pouvez l'utiliser pour compiler et exécuter le projet :

- Consultez le chapitre 3 de l'article Installation de vue-cli & config. idea pour configurer le projet dans IDEA.
- Après configuration, vous pouvez cliquer sur la flèche verte dans IDEA pour compiler et exécuter le projet. Le serveur se lancera automatiquement, et chaque modification du code déclenchera une nouvelle compilation avec mise à jour dans le navigateur.

2°/ Compléter la structure du projet

2.1°/ Les services d'accès aux données

Pour séparer la logique d'accès aux données de la vue, nous allons créer des services pour simuler l'accès aux données avant d'intégrer une API réelle.

Création des fichiers de service Dans le répertoire `src`, créez un sous-dossier `services` et ajoutez-y deux fichiers :

1. **shop.service.js** (fourni dans l'exemple)
2. **bankaccount.service.js** (vide, à compléter au fur et à mesure)

Dans **shop.service.js**, vous trouverez des exemples de fonctions pour récupérer les données de la boutique de virus. Ces fonctions utilisent des données locales et renvoient des objets structurés dans le format suivant :

```
{
  "error": 0,
  "status": 200,
  "data": {...}
}
```

Une fonction comme `getAllViruses()` pourra appeler soit une fonction locale `getAllVirusesFromLocalSource()`, soit une fonction future `getAllVirusesFromAPI()`.

2.2°/ Créer la source de données "locale"

Pour simuler l'accès à une API avant que celle-ci soit fonctionnelle, vous allez utiliser des fichiers locaux contenant des données et des fonctions pour y accéder.

Téléchargez et décompressez l'archive `datasource.tgz` dans le répertoire `src`. Cette archive contient deux fichiers principaux :

- **data.js** : contient des objets et tableaux de données représentant les informations manipulées par l'application.
- **controller.js** : contient des fonctions qui simulent l'accès à une API, comme la récupération des virus ou le login de l'utilisateur.

Ces fichiers sont conçus pour renvoyer des données structurées comme une API réelle, ce qui permet de tester l'application sans avoir besoin d'un back-end opérationnel.

2.3°/ Intermède sur la structuration des données pour le front (optionnel)

Bien que ce point ne soit pas directement utilisé dans ce TP, il est important de comprendre que l'application SPA peut avoir besoin de transformer ou de structurer les données reçues d'une API pour les rendre compatibles avec l'application front-end.

Cela peut inclure la création de classes pour manipuler les objets de manière plus flexible et organiser les données de façon cohérente. Cela permet de faciliter le travail avec des APIs de structures variées et de mieux gérer les transformations des données.

3°/ Une première application

3.1°/ Objectif L'objectif de cette version de l'application est de pouvoir afficher deux types de pages en fonction de l'URL, en utilisant **vue-router** :

- **Page 1** : Liste des virus disponibles à la vente.
- **Page 2** : Formulaire de login permettant à un utilisateur de se connecter à la boutique.

Ces pages seront affichées via **vue-router**, qui pilote l'affichage en fonction de l'URL saisie. Par exemple :

- `localhost:8080/shop/items` -> Affiche la liste des virus.
- `localhost:8080/shop/login` -> Affiche un formulaire de login.

Les données (comme les virus ou l'utilisateur) sont stockées en mémoire via le tableau `data.js`, et récupérées via des méthodes de service. Le stockage des données se fera principalement dans **Vuex Store**, afin d'apprendre à gérer un état global, bien que cela puisse être optimisé dans un cas réel.

3.2°/ Mise en place initiale

1. **Effacer les fichiers dans `components` et `views`.**
2. À la racine du projet, installez le module `uuid` via NPM :

   ```
   npm install uuid
   ```

3. Téléchargez l'archive `vuejs-tp1-src.tgz` et décompressez-la dans le répertoire `src`. Cette archive ajoutera/écrasera les fichiers suivants :

- `App.vue`
- `views/ShopLoginView.vue` et `views/VirusesView.vue`
- `router/index.js`
- `store/index.js`

Une fois ces fichiers mis en place, **l'application devrait se compiler sans erreur** et vous devriez voir une page centrale avec un logo et un message de bienvenue. Vous pouvez tester les URLs comme suit : - `localhost:8080/shop/items` : Liste des virus. - `localhost:8080/shop/login` : Formulaire de login.

Explication des fichiers :

- **`store/index.js`** : Utilisation de **Vuex** pour centraliser les données. Il définit des variables (state), des mutations (modifications synchrones de l'état), et des actions (modifications asynchrones de l'état via des appels API).
- **`router/index.js`** : Configure **vue-router**, en définissant des "routes" pour chaque URL. Chaque route associe un chemin (ex : `/shop/items`) à un composant.
- **`App.vue`** : Contient la balise `<router-view>` qui permet d'afficher dynamiquement un composant en fonction de la route activée.
- **`VirusesView.vue`** : Affiche la liste des virus en utilisant le **store** pour accéder aux données via `mapState()`.

3.3°/ Affichage du solde d'un compte bancaire Étapes à suivre :

1. **`controller.js`** :
   - Importer le tableau `bankaccounts` depuis `data.js`.
   - Créer une fonction `getAccountAmount(number)` qui vérifie si l'ID du compte est valide et retourne le solde du compte.

   Exemple de fonction dans `controller.js` :

```
function getAccountAmount(number) {
    if (!number) {
        return { error: 1, message: "Account number is required" };
    }
    const account = bankaccounts.find(acc => acc.number === number);
    if (account) {
        return { error: 0, data: account.amount };
    } else {
        return { error: 1, message: "Account not found" };
    }
}
```

2. **`bankaccount.service.js`** (à créer) :
   - Créer une fonction `getAccountAmountFromLocalSource(number)` qui appelle la fonction du **controller.js** et retourne les résultats.
   - Une autre fonction `getAccountAmount(number)` qui appelle `getAccountAmountFromLocalSource` et retourne un objet d'erreur ou de données.

   Exemple :

```
import { getAccountAmount } from './controller';

function getAccountAmountFromLocalSource(number) {
    return getAccountAmount(number);
}

export function getAccountAmount(number) {
    return getAccountAmountFromLocalSource(number);
}
```

3. **`store/index.js`** :
   - Ajouter une variable `accountAmount` dans le `state`, initialisée à 0.
   - Ajouter une mutation `updateAccountAmount()` pour mettre à jour le solde du compte dans le **state**.
   - Créer une action `getAccountAmount()` qui utilise la méthode `BankService.getAccountAmount()` pour récupérer le solde d'un compte.

Exemple :

```
state: {
    accountAmount: 0
},
mutations: {
    updateAccountAmount(state, amount) {
        state.accountAmount = amount;
    }
},
actions: {
    async getAccountAmount({ commit }, number) {
        const result = await BankService.getAccountAmount(number);
        if (result.error === 0) {
            commit('updateAccountAmount', result.data);
        } else {
            console.error(result.message);
        }
    }
}
```

4. **`views/BankAccountView.vue`** (à créer) :

   - Copiez et adaptez le code de `ShopLoginView.vue` pour afficher un seul champ de saisie pour l'identifiant du compte et le solde du compte.
   - Utilisez `mapState()` pour récupérer `accountAmount` du store.
   - Utilisez `mapActions()` pour appeler l'action `getAccountAmount()`.

   Exemple :

```
<template>
    <div>
        <input v-model="accountNumber" placeholder="Enter account number" />
        <button @click="getAccountAmount(accountNumber)">Get Account Amount</button>
        <p v-if="accountAmount !== null">Account Balance: {{ accountAmount }}</p>
    </div>
</template>

<script>
import { mapState, mapActions } from 'vuex';

export default {
    data() {
        return {
            accountNumber: ''
        };
    },
    computed: {
        ...mapState(['accountAmount'])
    },
    methods: {
        ...mapActions(['getAccountAmount'])
    }
};
</script>
```

5. **`router/index.js`** :

   - Ajouter une nouvelle route `/bank/amount` pour afficher le composant `BankAccountView`.

   Exemple :

```
{
    path: '/bank/amount',
```

```
        name: 'BankAccount',
        component: () => import('../views/BankAccountView.vue')
    }
```

3.4°/ Affichage des transactions liées à un compte bancaire

1. `controller.js` :
   - Ajouter une méthode `getAccountTransactions(number)` pour récupérer les transactions associées à un compte bancaire.
2. `bankaccount.service.js` :
   - Créer une fonction `getAccountTransactionsFromLocalSource(number)` qui appelle la fonction du **controller.js** pour récupérer les transactions.
3. `store/index.js` :
   - Ajouter un tableau `accountTransactions` dans le `state`.
   - Ajouter une mutation `updateAccountTransactions()` pour mettre à jour les transactions dans le **state**.
   - Créer une action `getAccountTransactions()` pour récupérer les transactions via le service.
4. `views/BankAccountView.vue` :
   - Ajouter un bouton pour récupérer les transactions et afficher la liste des transactions sous forme de liste à puces, uniquement si des transactions existent.

4°/ Conclusion

Cette première application montre comment la structuration avec **Vue.js**, **Vue Router**, et **Vuex** permet d'ajouter de nouvelles fonctionnalités de manière incrémentale tout en centralisant la gestion des données. La séparation des préoccupations entre les composants, le store, et les services rend l'application modulaire et facile à maintenir. Cela permet également de développer des fonctionnalités rapidement en réutilisant des composants existants.