Apprendre Vue.js par la pratique

Introduction à Vue.js

1.1. Qu'est-ce que Vue.js?

Vue.js est un framework progressif pour la construction d'interfaces utilisateur. Contrairement à d'autres frameworks monolithiques, Vue est conçu dès le départ pour être adopté de manière incrémentielle. La bibliothèque principale se concentre uniquement sur la couche vue, et il est facile d'intégrer Vue avec d'autres bibliothèques ou projets existants. D'autre part, Vue est également parfaitement capable de faire tourner des applications sophistiquées en mono-page lorsqu'il est combiné avec des outils modernes et des bibliothèques complémentaires.

1.2. Avantages de Vue.js

Vue.js offre une multitude d'avantages qui le rendent attrayant pour le développement d'applications web modernes :

- **Réactivité :** Vue.js assure une mise à jour fluide et réactive des éléments de l'interface utilisateur en réponse aux changements de données.
- **Composabilité :** Avec les composants, il est facile de réutiliser le code et de créer des applications complexes.
- **Facilité d'apprentissage :** Vue.js est connu pour sa courbe d'apprentissage douce, ce qui le rend accessible aux débutants.
- **Performances :** Il offre de très bonnes performances et une charge rapide des applications grâce à un système de virtual DOM optimisé.
- **Écosystème :** Un écosystème riche avec des outils comme Vue Router, Vuex pour la gestion d'état, et Vue CLI pour l'automatisation du workflow.

1.3. Comparaison avec d'autres frameworks frontend

Vue.js se compare favorablement à d'autres frameworks populaires comme React et Angular :

- **React :** Similaire à Vue dans son approche de composants réutilisables, mais Vue offre une intégration plus simple et une syntaxe moins verbeuse.
- **Angular :** Vue est souvent considéré comme plus léger et plus rapide que Angular, avec une configuration moins complexe.
- **Ember :** Tandis qu'Ember est hautement opinioné avec une courbe d'apprentissage plus raide, Vue offre plus de flexibilité et est plus facile à intégrer dans des projets existants.

Vue.js a gagné une grande popularité parmi les développeurs grâce à sa simplicité, ses performances et son approche progressive.

Installation et Configuration

Avant de commencer à travailler avec Vue.js, nous devons installer les outils nécessaires pour le développement. Dans ce tutoriel, nous allons utiliser Vue CLI, un outil de ligne de commande pour la création et la gestion de projets Vue.js.

2.1. Installation de Node.js

Pour installer `Vue CLI`, nous devons d'abord installer Node.js, qui est un environnement d'exécution JavaScript construit sur le moteur JavaScript V8 de Chrome. Node.js est nécessaire pour exécuter des applications JavaScript côté serveur. Il est également utilisé pour installer des outils de développement comme Vue CLI.

- Sur Linux, vous pouvez installer Node.js à l'aide de votre gestionnaire de paquets. Par exemple, sur Ubuntu, vous pouvez utiliser `apt` :

```
sudo apt install nodejs
```

Vérifiez que Node.js est installé avec succès en exécutant la commande suivante :

```
node -v
```

Cette commande affichera la version de Node.js installée sur votre système.

- Sur macOS, vous pouvez installer Node.js à l'aide de Homebrew :

```
brew install node
```

Vérifiez que Node.js est installé avec succès en exécutant la commande suivante :

```
node -v
```

Cette commande affichera la version de Node.js installée sur votre système.

- Sur Windows, vous pouvez installer Node.js depuis le site officiel.

2.1. Installation de Vue.js

L'installation de Vue.js peut être effectuée de plusieurs manières. La méthode la plus simple est d'inclure Vue avec une balise `<script>` en pointant vers un CDN comme `jsDelivr` ou `cdnjs`. Pour des projets plus conséquents ou pour une utilisation plus avancée, il est recommandé d'utiliser NPM ou Yarn :

Une fois que Node.js est installé, nous pouvons installer Vue CLI à l'aide de la commande suivante :

```
npm install -g @vue/cli
```

ou avec `Yarn` :

```
yarn global add @vue/cli
```

2.2. Création d'un nouveau projet

Pour créer un nouveau projet Vue.js, le plus simple est d'utiliser Vue CLI. Après l'avoir installé globalement avec NPM ou Yarn :

```
npm install -g @vue/cli
# ou
yarn global add @vue/cli
```

Vous pouvez créer un nouveau projet en exécutant :

```
vue create my-project
```

2.3. Structure du projet Vue.js

Un projet Vue.js typique créé avec Vue CLI aura la structure de dossier suivante :

- `node_modules/` : Dossiers contenant toutes les dépendances.
- `public/` : Fichiers statiques qui ne nécessitent pas de traitement webpack.
- `src/` : Code source de votre application, y compris les assets, composants et vues.
- `package.json` : Fichier de configuration pour les dépendances et scripts.
- `vue.config.js` : Un fichier optionnel pour configurer Vue CLI.

2.4. Configuration de Vue CLI

Vue CLI offre une grande flexibilité et permet une configuration approfondie à travers le fichier `vue.config.js`. Cela inclut des options pour le serveur de développement, la construction de production, et plus encore.

2.5. Vue DevTools

Pour une expérience de développement améliorée, Vue DevTools peut être installé comme une extension de navigateur pour Chrome et Firefox. Elle offre des fonctionnalités telles que l'inspection des composants, la débogage du state management, et des performances de timing pour les rendus de composants.

Les Fondamentaux de Vue.js

3.1. Instance Vue

Chaque application Vue commence par la création d'une nouvelle instance Vue avec la fonction `Vue`. Cette instance sert de point d'ancrage pour le montage de la structure réactive de Vue sur un élément du DOM :

```
var vm = new Vue({
  // options
});
```

3.2. Syntaxe de Template

Les templates de Vue.js utilisent du HTML valide avec des syntaxes spéciales pour lier le DOM rendu aux données de l'instance Vue. Vue.js compile les templates en fonctions de rendu Virtual DOM.

3.3. Data Binding

Le data binding est le mécanisme qui relie les données de l'instance Vue au template. Il permet de mettre à jour automatiquement le DOM lorsque les données changent.

3.3.1. Interpolations de texte

Utilisation de la syntaxe `{{ }}` pour afficher du texte :

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

D'où provient la valeur de `message` ? Elle est définie dans l'instance Vue :

```
var app = new Vue({
  el: "#app",
  data: {
    message: "Hello Vue!",
  },
});
```

L'instance Vue est l'objet principal de Vue qui contient les données de l'application. Elle peut être créée avec les options suivantes :

- `el` : Sélectionne l'élément DOM sur lequel monter l'instance Vue.
- `data` : Définit les données de l'application.
- `methods` : Définit les méthodes de l'application.
- `computed` : Définit les propriétés calculées de l'application.
- `watch` : Définit les watchers de l'application.
- `template` : Définit le template de l'application.
- `components` : Définit les composants de l'application.
- `props` : Définit les props de l'application.
- `mixins` : Définit les mixins de l'application.
- `directives` : Définit les directives de l'application.
- `filters` : Définit les filtres de l'application.
- `extends` : Définit l'extension de l'application. etc.

Exemple le code ci-dessus en détail :

- `var app = new Vue({})` : Crée une nouvelle instance Vue.
- `el: '#app'` : Sélectionne l'élément avec l'ID `app` pour le montage de l'instance Vue.
- `data: { message: 'Hello Vue!' }` : Définit la propriété `message` sur l'instance Vue.

3.3.2. Data Binding pour les attributs HTML

Data bidding sert également à lier des attributs HTML à des données de l'instance Vue.

Utilisation de la syntaxe `v-bind` pour lier des attributs HTML à des expressions JavaScript :

```
<div id="app">
  <span v-bind:title="message">
    Passez votre souris ici pour voir le titre dynamique !
  </span>
</div>
```

Dans ce code, `v-bind` est une directive qui lie l'attribut `title` de l'élément `<span>` à la propriété `message` de l'instance Vue :

```
var app = new Vue({
  // Crée une nouvelle instance Vue
  el: "#app", // Sélectionne l'élément DOM avec l'ID `app`
  data: {
    message: "Vous avez affiché cette page le " + new
Date().toLocaleString(), // Date et heure actuelles
  },
});
```

3.3.3. Utilisation de JavaScript Expressions

Vue.js permet l'utilisation d'expressions JavaScript simples à l'intérieur des doubles accolades

```
<span>{{ message.split('').reverse().join('') }}</span>
```

Exemple de code ci-dessus en détail :

- `message.split('')` : Divise la chaîne de caractères en un tableau de caractères.
- `reverse()` : Inverse l'ordre des éléments du tableau.
- `join('')` : Rassemble les éléments du tableau en une chaîne de caractères.

Les expressions peuvent également être utilisées dans les attributs HTML :

```
<div v-bind:id="'list-' + id"></div>
```

Exemple de code ci-dessus en détail :

- `v-bind:id` : Lie l'attribut `id` à une expression JavaScript.
- `"'list-' + id"` : Concatène la chaîne de caractères `'list-'` avec la valeur de la propriété `id`.

```
data: {
  id: 1;
}
```

Dans ce code, l'attribut `id` de l'élément `<div>` sera lié à la valeur de la propriété `id` de l'instance Vue.

3.4. Directives

Les directives sont des attributs spéciaux avec le préfixe `v-`. Elles appliquent des comportements réactifs au rendu du DOM.

- **3.4.1. v-bind**

La directive `v-bind` permet de lier des attributs HTML à des expressions JavaScript :

```
<a v-bind:href="url">Link</a>
```

Exemple de code ci-dessus en détail :

- `v-bind:href` : Lie l'attribut `href` à une expression JavaScript.

```
data: {
  url: "https://www.google.com";
}
```

Dans ce code, l'attribut `href` de l'élément `<a>` sera lié à la valeur de la propriété `url` de l'instance Vue.

- **3.4.2. v-model**

Crée une liaison bidirectionnelle sur un élément de formulaire :

```
<input v-model="message" placeholder="Entrez quelque chose" />
<p>Message : {{ message }}</p>
```

Exemple de code ci-dessus en détail :

- `v-model="message"` : Crée une liaison bidirectionnelle sur l'élément `<input>` avec la propriété `message` de l'instance Vue.

```
data: {
  message: "";
}
```

Dans ce code, la valeur de l'élément `<input>` sera liée à la propriété `message` de l'instance Vue. Lorsque la valeur de l'élément `<input>` change, la propriété `message` sera mise à jour et vice versa.

- **3.4.3. v-for**

La directive `v-for` permet d'itérer sur un tableau ou un objet :

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

Exemple de code ci-dessus en détail :

- `v-for="item in items"` : Itère sur le tableau `items` et crée un élément `<li>` pour chaque élément du tableau.

```
data: {
  items: ["Item 1", "Item 2", "Item 3"];
}
```

Dans ce code, la directive `v-for` crée un élément `<li>` pour chaque élément du tableau `items`.

- **3.4.4. v-if, v-else-if, v-else**

La directive `v-if` permet de conditionner l'affichage d'un élément :

```
<p v-if="seen">Maintenant vous me voyez</p>
```

Exemple de code ci-dessus en détail :

- `v-if="seen"` : Affiche l'élément `<p>` si la propriété `seen` de l'instance Vue est vraie.

```
data: {
  seen: true;
}
```

Dans ce code, l'élément `<p>` sera affiché si la propriété `seen` de l'instance Vue est vraie.

La directive `v-else` permet d'afficher un élément si la condition de `v-if` n'est pas remplie :

```
<p v-if="seen">Maintenant vous me voyez</p>
<p v-else>Vous ne me voyez plus</p>
```

Exemple de code ci-dessus en détail :

- `v-else` : Affiche l'élément `<p>` si la propriété `seen` de l'instance Vue est fausse.

```
data: {
  seen: false;
}
```

Dans ce code, l'élément `<p>` sera affiché si la propriété `seen` de l'instance Vue est fausse.

La directive `v-else-if` permet d'afficher un élément si la condition de `v-if` n'est pas remplie et qu'une autre condition est remplie :

```
<p v-if="type === 'A'">A</p>
<p v-else-if="type === 'B'">B</p>
```

Exemple de code ci-dessus en détail :

- `v-else-if="type === 'B'"` : Affiche l'élément `<p>` si la propriété `type` de l'instance Vue est égale à `'B'`.

```
data: {
  type: "B";
}
```

- **3.4.5. v-on**

La directive `v-on` permet d'écouter les événements du DOM et d'exécuter des méthodes JavaScript en réponse :

```
<button v-on:click="doSomething">Cliquez ici</button>
```

Exemple de code ci-dessus en détail :

- `v-on:click="doSomething"` : Exécute la méthode `doSomething` lorsque l'élément `<button>` est cliqué.

```
methods: {
    doSomething: function () {
        // On fait quelque chose ici
        'Hello World!'
    }
}
```

Dans ce code, la méthode `doSomething` sera exécutée lorsque l'élément `<button>` est cliqué. La méthode `doSomething` affichera la chaîne de caractères `'Hello World!'` dans la console.

3.5. Événements et Gestion des Événements

Les événements sont des actions qui se produisent sur une page web, comme le chargement d'une page ou le clic sur un bouton. Vue.js permet d'écouter les événements du DOM et d'exécuter des méthodes JavaScript en réponse.

Gérer les interactions utilisateur avec `v-on` ou le raccourci `@` pour attacher des événements aux méthodes :

```
<button @click="sayHello">Say hello</button>
```

Exemple de code ci-dessus en détail :

- `@click="sayHello"` : Exécute la méthode `sayHello` lorsque l'élément `<button>` est cliqué.

```
methods: {
    sayHello: function () {
        alert('Hello!')
    }
}
```

Composants en Vue.js

4.1. Introduction aux Composants

Les composants sont des instances de Vue réutilisables qui sont définies avec un nom et peuvent contenir leur propre template, logique et état. Ils permettent de construire des interfaces utilisateur complexes en encapsulant et en réutilisant des morceaux de code.

```
Vue.component("my-component", {
  // options
});
```

4.1. Introduction aux Composants

4.2. Communication entre Composants

Les composants Vue interagissent principalement via les props pour le passage des données et les événements pour les notifications.

- **4.2.1.** Props

Les props sont des attributs personnalisés que vous pouvez enregistrer sur un composant. Quand une valeur est passée à une prop attribut, elle devient une propriété de ce composant.

```
Vue.component("child", {
  // Déclare les props
  props: ["message"],
  // Comme les data, les props peuvent être utilisées à l'intérieur des
templates
  // et sont également accessibles à partir de `this`.
  template: "<span>{{ message }}</span>",
});
```

```
<child message="Hello!"></child>
```

Dans ce code, le composant `child` a une prop `message` qui est définie sur la chaîne de caractères `'Hello!'`. Le composant `child` affichera la valeur de la prop `message` dans le template.

- **4.2.2.** Événements personnalisés

Les composants peuvent émettre des événements en utilisant `$emit`, et ces événements peuvent être écoutés sur le composant parent.

```
Vue.component("button-counter", {
  data: function () {
    return {
      count: 0, // Définit la propriété `count` sur le composant `button-
counter`
    };
  },
  template: '<button v-on:click="incrementCounter">{{ count }}</button>',
  methods: {
    incrementCounter: function () {
      this.count += 1; // Incrémente la propriété `count` sur le composant
`button-counter`
      this.$emit("increment"); // Émet l'événement `increment`
    },
  },
});
```

Expliquons le code JavaScript ci-dessus en détail :

- `data: function () { return { count: 0 } }` : Définit la propriété `count` sur le composant `button-counter`.
- `template: '<button v-on:click="incrementCounter">{{ count }}</button>'` : Définit le template du composant `button-counter`.

- `methods: { incrementCounter: function () { this.count += 1 this.$emit('increment') } }` : Définit la méthode `incrementCounter` sur le composant `button-counter`. La méthode `incrementCounter` incrémente la propriété `count` et émet l'événement `increment`.

Ensuite, nous pouvons écouter l'événement `increment` et exécuter une méthode lorsque l'événement est émis dans le code :

```
<div id="app">
  <button-counter v-on:increment="incrementTotal"></button-counter>
  <p>Total clicks: {{ count }}</p>
</div>
```

Dans ce code, le composant `button-counter` émet l'événement `increment` lorsque le bouton est cliqué. L'événement `increment` est écouté sur le composant parent, et la méthode `incrementTotal` est exécutée lorsque l'événement est émis.

4.3. Slots

Les slots sont des points d'insertion dans un composant qui permettent d'insérer du contenu HTML à partir du composant parent.

```
Vue.component('alert-box', {
    template: `
        <div class="demo-alert-box">
            <strong>Erreur!</strong>
            <slot></slot>
        </div>
})
```

Dans le ci-dessus, le composant `alert-box` émet un slot qui permet d'insérer du contenu HTML à partir du composant parent.

```
<alert-box> Une erreur est survenue. Veuillez réessayer. </alert-box>
```

Dans ce code, le composant `alert-box` affichera le contenu HTML inséré dans le slot.

4.4. Composants Dynamiques

Avec la directive component et l'attribut spécial `is`, il est possible de basculer entre différents composants dynamiquement.

```
new Vue({
  el: "#app",
```

```
  data: {
    currentTabComponent: "tab-home",
  },
  components: {
    "tab-home": {
      template: "<div>Home component</div>",
    },
    "tab-posts": {
      template: "<div>Posts component</div>",
    },
  },
});
```

Expliquons ce code en détail :

- `currentTabComponent: 'tab-home'` : Définit la propriété `currentTabComponent` sur l'instance Vue.
- `components: { 'tab-home': { template: '<div>Home component</div>' }, 'tab-posts': { template: '<div>Posts component</div>' } }` : Définit les composants `tab-home` et `tab-posts` sur l'instance Vue.
- `template: '<div id="app"> <button @click="currentTabComponent = 'tab-home'">Home</button> <button @click="currentTabComponent = 'tab-posts'">Posts</button> <component v-bind:is="currentTabComponent"> </component> </div>'` : Définit le template de l'instance Vue.

```
<div id="app">
  <button @click="currentTabComponent = 'tab-home'">Home</button>
  <button @click="currentTabComponent = 'tab-posts'">Posts</button>
  <component v-bind:is="currentTabComponent"></component>
</div>
```

Dans ce code, le composant `tab-home` sera affiché si la propriété `currentTabComponent` de l'instance Vue est égale à `'tab-home'`. Le composant `tab-posts` sera affiché si la propriété `currentTabComponent` de l'instance Vue est égale à `'tab-posts'`.

4.5. Composants Asynchrones

Les composants peuvent être chargés de manière asynchrone avec `Vue.component` et `import`.

```
Vue.component({
  "async-component": () => import("./AsyncComponent.vue"),
});
```

Dans ce code, le composant `async-component` sera chargé de manière asynchrone.

Computed Properties et Watchers

5.1. Computed Properties

Les propriétés calculées (`computed properties`) sont des valeurs dérivées des données de l'instance Vue. Elles sont mises en cache et ne se recalculent que lorsque l'une des données dépendantes change, ce qui les rend très performantes pour des opérations qui n'ont pas besoin d'être exécutées à chaque re-render de la vue.

```
new Vue({
  el: "#example",
  data: {
    a: 1,
    b: 2,
  },
  computed: {
    sum: function () {
      return this.a + this.b;
    },
  },
});
```

Dans ce code, la propriété calculée `sum` est définie sur l'instance Vue. La propriété calculée `sum` retourne la somme des propriétés `a` et `b` de l'instance Vue.

```
<div id="example">
  <p>La somme de {{ a }} + {{ b }} est {{ sum }}.</p>
</div>
```

5.2. Watchers

Les observateurs (`watchers`) sont des fonctions qui vous permettent de réagir aux changements sur vos données Vue. Ils sont particulièrement utiles lorsque vous voulez effectuer des opérations asynchrones ou coûteuses en réponse à un changement de données.

```
new Vue({
  el: "#example",
  data: {
    a: 1,
    b: 2,
  },
  watch: {
    a: function (newVal, oldVal) {
      console.log(`a a changé de ${oldVal} à ${newVal}`);
    },
  },
});
```

Dans ce code, l'observateur `a` est défini sur l'instance Vue. L'observateur `a` affichera un message dans la console lorsque la propriété `a` de l'instance Vue change.

```
<div id="example">
  <p>La somme de {{ a }} + {{ b }} est {{ sum }}.</p>
</div>
```

Les propriétés calculées et les watchers sont deux des fonctionnalités les plus puissantes de Vue.js, permettant une gestion efficace des données réactives et des dépendances dans vos applications. Les propriétés calculées sont utilisées pour des calculs synchrones simples, tandis que les watchers sont préférés pour des tâches asynchrones ou plus complexes réagissant aux changements de données.

Gestion des États

6.1. État Local et Data Functions

Chaque instance de composant dans Vue.js a son propre état local, accessible via la fonction `data`. C'est ici que vous définirez les propriétés réactives qui contrôlent le comportement du composant.

```
new Vue({
  data() {
    return {
      counter: 0,
    };
  },
});
```

Dans ce code, la fonction `data` est définie sur l'instance Vue. La fonction `data` retourne un objet avec la propriété `counter` initialisée à `0`.

`counter` est une propriété de l'état local du composant, qui peut être liée au template et mise à jour en conséquence.

```
<div id="app">
  <p>Le compteur est à {{ counter }}</p>
</div>
```

6.2. Gestion des états globaux avec Vuex

Vuex est une bibliothèque de gestion d'état pour les applications Vue.js. Elle sert de magasin centralisé pour tous les composants dans une application, avec des règles garantissant que l'état peut uniquement être muté de manière prévisible.

- **6.2.1.** Installation de Vuex

Vuex peut être installé avec NPM ou Yarn :

```
npm install vuex --save
# ou
yarn add vuex
```

- **6.2.2.** Configuration de Vuex

Pour utiliser Vuex dans une application Vue.js, nous devons d'abord l'importer dans le fichier `main.js` :

```
import Vue from "vue";
import Vuex from "vuex";

Vue.use(Vuex);
```

Ensuite, nous pouvons créer une instance Vuex avec la fonction `Vuex.Store` :

```
const store = new Vuex.Store({
  // options
});
```

- **6.2.3.** State

Le `state` dans Vuex est l'objet où vous déclarez toutes les propriétés qui doivent être partagées entre plusieurs composants.

```
const store = new Vuex.Store({
  state: {
    count: 0,
  },
});
```

Dans ce code, le `state` est défini sur l'instance Vuex. Le `state` contient la propriété `count` initialisée à `0`.

`count` est une propriété de l'état global de l'application, qui peut être liée au template et mise à jour en conséquence.

```
<div id="app">
  <p>Le compteur est à {{ count }}</p>
</div>
```

- **6.2.4.** Getters Les `getters` sont comme des propriétés calculées pour le store Vuex. Ils peuvent être utilisés pour calculer des états dérivés basés sur le state du store.

```
const store = new Vuex.Store({
  state: {
    todos: [
      { id: 1, text: "Faire les courses", done: true },
      { id: 2, text: "Apprendre Vue.js", done: false },
    ],
  },
  getters: {
    doneTodos: (state) => {
      return state.todos.filter((todo) => todo.done);
    },
  },
});
```

Dans ce code, les `getters` sont définis sur l'instance Vuex. Les `getters` contiennent la méthode `doneTodos` qui retourne un tableau de todos terminés.

`doneTodos` est une propriété calculée pour le store Vuex, qui peut être liée au template et mise à jour en conséquence.

```
<div id="app">
  <p>Les todos terminés sont : {{ doneTodos }}</p>
</div>
```

- **6.2.5.** Mutations Les `mutations` sont des fonctions qui modifient le state du store Vuex. Elles peuvent être utilisées pour effectuer des modifications synchrones.

```
const store = new Vuex.Store({
  state: {
    count: 0,
  },
  mutations: {
    increment(state) {
      state.count++;
    },
  },
});
```

Dans ce code, les `mutations` sont définies sur l'instance Vuex. Les `mutations` contiennent la méthode `increment` qui incrémente la propriété `count` du state.

`increment` est une méthode pour le store Vuex, qui peut être appelée dans le template et mise à jour en conséquence.

```
<div id="app">
  <p>Le compteur est à {{ count }}</p>
```

```
  <button @click="increment">Incrémenter</button>
</div>
```

- **6.2.6.** Actions

Les `actions` sont des fonctions qui modifient le state du store Vuex. Elles peuvent être utilisées pour effectuer des modifications asynchrones.

Elle sont similaires aux mutations, à la différence qu'elles ne modifient pas directement l'état, mais peuvent contenir des opérations asynchrones avant de commettre des mutations.

```
const store = new Vuex.Store({
  state: {
    count: 0,
  },
  mutations: {
    increment(state) {
      state.count++;
    },
  },
  actions: {
    increment(context) {
      context.commit("increment");
    },
  },
});
```

Dans ce code, les `actions` sont définies sur l'instance Vuex. Les `actions` contiennent la méthode `increment` qui appelle la méthode `increment` du store Vuex.

`increment` est une méthode pour le store Vuex, qui peut être appelée dans le template et mise à jour en conséquence.

```
<div id="app">
  <p>Le compteur est à {{ count }}</p>
  <button @click="increment">Incrémenter</button>
</div>
```

- **6.2.5.** Modules

Les `modules` sont des espaces de noms pour les `state`, `getters`, `mutations` et `actions` dans le store Vuex. Ils permettent de diviser le store Vuex en modules plus petits et plus faciles à gérer.

```
const store = new Vuex.Store({
  modules: {
    a: {
      state: {
```

```
        count: 0,
      },
      mutations: {
        increment(state) {
          state.count++;
        },
      },
      actions: {
        increment(context) {
          context.commit("increment");
        },
      },
      getters: {
        doneTodos: (state) => {
          return state.todos.filter((todo) => todo.done);
        },
      },
    },
  },
});
```

Dans ce code, le module `a` est défini sur l'instance Vuex. Le module `a` contient le `state`, les `mutations`, les `actions` et les `getters` du store Vuex.

Ensuite, nous pouvons accéder aux propriétés du module `a` dans le template :

```
<div id="app">
  <p>Le compteur est à {{ a.count }}</p>
  <button @click="a.increment">Incrémenter</button>
</div>
```

Ce code affichera la propriété `count` du module `a` et appellera la méthode `increment` du module `a` lorsque le bouton est cliqué.

Routage avec Vue Router

Le routage dans une application Vue.js est géré par le Vue Router, qui permet de mapper des composants aux routes et de définir comment l'utilisateur navigue à travers l'application. Voici un aperçu détaillé en format markdown.

7.1. Introduction au Routage

Le routage est un processus essentiel dans les applications SPA (Single Page Applications) où l'URL reflète l'état actuel de l'application. Vue Router est la solution officielle pour le routage dans Vue.js, offrant une intégration profonde et des fonctionnalités telles que le lazy loading, la navigation, les route guards, et bien plus.

7.2. Configuration de Vue Router

Pour utiliser Vue Router, commencez par l'installer via npm ou yarn. Ensuite, créez une instance de Vue Router en définissant les routes et les composants correspondants.

```
import Vue from "vue";
import VueRouter from "vue-router";
import HomeComponent from "./components/HomeComponent";
import AboutComponent from "./components/AboutComponent";

Vue.use(VueRouter);

const routes = [
  { path: "/", component: HomeComponent },
  { path: "/about", component: AboutComponent },
];

const router = new VueRouter({
  routes,
  mode: "history",
});
```

Expliquons ce code en détail :

- `import Vue from 'vue'` : Importe Vue.
- `import VueRouter from 'vue-router'` : Importe Vue Router.
- `import HomeComponent from './components/HomeComponent'` : Importe le composant `HomeComponent`.
    - HomeComponent est un composant Vue qui sera affiché lorsque l'utilisateur navigue vers la route `/`.
- `import AboutComponent from './components/AboutComponent'` : Importe le composant `AboutComponent`.
- `Vue.use(VueRouter)` : Installe Vue Router.
- `const routes = [ { path: '/', component: HomeComponent }, { path: '/about', component: AboutComponent }, ]` : Définit les routes et les composants correspondants.
- `const router = new VueRouter({ routes, mode: 'history', })` : Crée une instance de Vue Router.
- `routes` : Définit les routes et les composants correspondants.
- `mode: 'history'` : Définit le mode de routage sur `history`.

7.3. Routes et Composants

7.3.1 Routes :

Les routes sont définies dans l'instance Vue Router. Une route est mappée à un composant. Quand une URL correspond à une route, le composant associé est rendu à l'endroit spécifié dans le template de l'application.

Chaque route est mappée à un composant. Quand une URL correspond à une route, le composant associé est rendu à l'endroit spécifié dans le template de l'application.

- Route `/` : Affiche le composant `HomeComponent` lorsque l'utilisateur navigue vers la route `/`.
- Route `/about` : Affiche le composant `AboutComponent` lorsque l'utilisateur navigue vers la route `/about`.

Ensuite, nous pouvons utiliser le composant `router-view` pour afficher le composant correspondant à la route :

```
<template>
  <div id="app">
    <router-view></router-view>
  </div>
</template>
```

Les deux composants `HomeComponent` et `AboutComponent` seront affichés dans le composant `router-view` lorsque l'utilisateur navigue vers la route `/` ou `/about`.

7.3.2 Composants :

Les composants sont des instances Vue qui définissent le comportement de l'application. Ils peuvent être définis dans le fichier `main.js` ou dans des fichiers séparés.

- Comment créer un composant dans le fichier `main.js` :

Pour créer un composant dans vue.js, créer un fichier `HomeComponent.vue` dans le dossier `components` et ajouter le code suivant dans le fichier :

```
<template>
  <div>Home component</div>
</template>
```

Enfin, nous pouvons utiliser le composant `HomeComponent` dans le fichier `main.js` :

```
Vue.component("home-component", {
  template: "<div>Home component</div>",
});
```

7.4. Navigation

La navigation peut être gérée programmablement en utilisant router.push, ou en utilisant le composant pour une navigation déclarative.

```
<template>
  <nav>
    <router-link to="/">Home</router-link>
    <router-link to="/about">About</router-link>
```

```
    </nav>
  </template>
```

7.5. Sous-routes et Route Guards

Les sous-routes permettent de définir des routes enfants, tandis que les route guards sont des fonctions qui peuvent être exécutées avant ou après des changements de route, permettant de sécuriser certaines routes ou de faire des vérifications.

```
const router = new VueRouter({
  routes: [
    {
      path: "/user/:id",
      component: UserComponent,
      children: [
        {
          path: "profile",
          component: UserProfileComponent,
        },
        {
          path: "posts",
          component: UserPostsComponent,
        },
      ],
    },
  ],
});

router.beforeEach((to, from, next) => {
  // effectue des actions avant chaque changement de route
  next();
});
```

Expliquons ce code en détail :

- `path: '/user/:id'` : Définit la route `/user/:id`.
- `component: UserComponent` : Définit le composant `UserComponent` pour la route `/user/:id`.
- `children: [ { path: 'profile', component: UserProfileComponent, }, { path: 'posts', component: UserPostsComponent, }, ]` : Définit les sous-routes `profile` et `posts` pour la route `/user/:id`.
- `router.beforeEach((to, from, next) => { // effectue des actions avant chaque changement de route next(); })` : Exécute la fonction `beforeEach` avant chaque changement de route.
- `to` : Définit la route vers laquelle l'utilisateur navigue.
- `from` : Définit la route depuis laquelle l'utilisateur navigue.
- `next` : Définit la fonction qui doit être appelée pour confirmer la navigation.

Pour mettre en place un routeur dans une application Vue.js, il faut l'intégrer dans l'instance racine de Vue :

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

En configurant et en utilisant Vue Router, vous pouvez gérer efficacement la navigation dans votre application Vue.js, en fournissant une expérience utilisateur fluide et en gardant l'URL synchronisée avec l'état de l'application.

Gestion des Formulaires

La gestion des formulaires est une partie essentielle du développement front-end. Vue.js simplifie ce processus avec des directives telles que `v-model`, qui permettent de créer facilement des liaisons bidirectionnelles entre les éléments de formulaire et les données de l'application. La validation des formulaires est également cruciale pour garantir que les données reçues sont valides et sécurisées.

8.1. Binding de Formulaire avec v-model

La directive `v-model` est utilisée pour créer une liaison bidirectionnelle sur les éléments de formulaire. Cela permet de lier les champs de formulaire aux données de l'instance Vue et est particulièrement utile pour réaliser des formulaires réactifs.

```
<template>
  <form>
    <input v-model="message" placeholder="Entrez un message" />
    <p>Le message est : {{ message }}</p>
  </form>
</template>
```

Lorsque l'utilisateur saisit du texte dans l'élément `<input>`, la propriété `message` de l'instance Vue est mise à jour et le message est affiché dans le template.

```
export default {
  data() {
    return {
      message: "",
    };
  },
};
```

Dans ce template ci-dessus, la propriété `message` est définie sur l'instance Vue. La propriété `message` est liée à l'élément `<input>` avec la directive `v-model`.

8.2. Validation de Formulaires

La validation de formulaire est une étape nécessaire pour prévenir les erreurs de l'utilisateur et assurer que les données soumises correspondent aux attentes. Vue.js ne fournit pas de système intégré de validation de formulaires, mais il existe des bibliothèques tierces telles que VeeValidate ou Vuelidate qui offrent des fonctionnalités de validation complètes.

Exemple de validation de formulaire avec VeeValidate :

```
<template>
  <form @submit.prevent="submitForm">
    <input v-model="email" v-validate="'required|email'" name="email" />
    <span>{{ errors.first('email') }}</span>
    <button type="submit">Submit</button>
  </form>
</template>
```

Dans ce code, la directive `v-validate` est utilisée pour valider l'élément `<input>`. La directive `v-validate` prend une chaîne de caractères qui définit les règles de validation. Dans ce cas, la règle `required` est utilisée pour vérifier que l'élément `<input>` n'est pas vide, et la règle `email` est utilisée pour vérifier que l'élément `<input>` contient une adresse email valide.

```
import { required, email } from "vee-validate/dist/rules";
import { extend } from "vee-validate";

extend("required", required);
extend("email", email);

export default {
  data() {
    return {
      email: "",
    };
  },
  methods: {
    submitForm() {
      this.$validator.validateAll().then((result) => {
        if (result) {
          alert("Form submitted!");
        }
      });
    },
  },
};
```

Dans ce code, les règles de validation `required` et `email` sont étendues avec la fonction `extend` de VeeValidate. Ensuite, la méthode `submitForm` est définie sur l'instance Vue. La méthode `submitForm` est appelée lorsque le formulaire est soumis. La méthode `submitForm` appelle la méthode `validateAll` de VeeValidate pour valider tous les éléments du formulaire. Si la validation réussit, le formulaire est soumis.

Expliquons ce code en détail :

- `import { required, email } from 'vee-validate/dist/rules'` : Importe les règles de validation `required` et `email` de VeeValidate.
- `import { extend } from 'vee-validate'` : Importe la fonction `extend` de VeeValidate.
- `extend('required', required)` : Étend la règle de validation `required` avec la fonction `extend` de VeeValidate.
- `extend('email', email)` : Étend la règle de validation `email` avec la fonction `extend` de VeeValidate.
- `extend('required',
- `data() { return { email: '' } }` : Définit la propriété `email` sur l'instance Vue. `required', required)` : Étend la règle de validation `required` avec la fonction `extend` de VeeValidate.
- submitForm() { this.$validator.validateAll().then((result) => { if (result) { alert('Form submitted!'); } }); } : Définit la méthode `submitForm` sur l'instance Vue. La méthode `submitForm` est appelée lorsque le formulaire est soumis. La méthode `submitForm` appelle la méthode `validateAll` de VeeValidate pour valider tous les éléments du formulaire. Si la validation réussit, le formulaire est soumis.
- `this.$validator.validateAll().then((result) => { if (result) { alert('Form submitted!'); } });` : Valide tous les éléments du formulaire avec la méthode `validateAll` de VeeValidate. Si la validation réussit, le formulaire est soumis.
- `if (result) { alert('Form submitted!'); }` : Si la validation réussit, le formulaire est soumis.
- `alert('Form submitted!')` : Affiche une alerte lorsque le formulaire est soumis.

En combinant v-model avec une bibliothèque de validation, vous pouvez créer des expériences utilisateur robustes et réactives dans vos applications Vue.js.

Mixins et Plugins

Vue.js offre des mixins et des plugins comme des moyens puissants pour réutiliser et étendre la fonctionnalité à travers votre application. Les mixins permettent de définir des méthodes, des hooks de cycle de vie, des propriétés, etc., qui peuvent être réutilisés dans différents composants. Les plugins ajoutent des fonctionnalités globales à Vue qui peuvent bénéficier à toute l'application, comme ajouter des méthodes globales ou des directives personnalisées.

9.1. Utilisation des Mixins

Les mixins sont un moyen flexible de distribuer des fonctionnalités réutilisables pour les composants Vue. Un mixin peut contenir n'importe quelle option de composant. Lorsqu'un composant utilise un mixin, toutes les options du mixin sont "mixées" dans les options du composant.

Exemple de mixin :

```
// mixin.js
export const myMixin = {
  created() {
    this.hello();
```

```
  },
  methods: {
    hello() {
      console.log("hello from mixin!");
    },
  },
};

// Dans un composant Vue
import { myMixin } from "./mixin";

export default {
  mixins: [myMixin],
  created() {
    this.hello();
  },
};
```

Dans ce code, le mixin `myMixin` est défini dans le fichier `mixin.js`. Le mixin `myMixin` contient la méthode `hello` qui affiche un message dans la console.

Ensuite, le mixin `myMixin` est utilisé dans le composant Vue. Le mixin `myMixin` est utilisé dans le composant Vue avec la propriété `mixins`.

```
mixins: [myMixin];
```

9.2. Création et Utilisation des Plugins

Les plugins permettent de créer de nouvelles fonctionnalités qui peuvent être réutilisées dans toute l'application Vue. Par exemple, vous pouvez créer un plugin pour installer une bibliothèque externe, définir des composants globaux ou ajouter des fonctions globales.

Exemple de création et utilisation d'un plugin :

```
// loggerPlugin.js
export default {
  install(Vue, options) {
    Vue.prototype.$log = function (message) {
      console.log(message);
    };
  },
};

// Dans main.js ou un point d'entrée similaire
import Vue from "vue";
import LoggerPlugin from "./loggerPlugin";

Vue.use(LoggerPlugin);
```

```
// Utilisation dans les composants
this.$log("Some message");
```

Animation et Transition

Vue fournit une série d'outils pour appliquer des transitions et des animations aux éléments du DOM lorsqu'ils sont ajoutés/retirés ou lorsque leur état change. Cela est réalisé via le composant `<transition>` et les `hooks` de transition.

10.1. Transition entre Éléments

Le composant `<transition>` de Vue enveloppe un élément ou un groupe d'éléments et applique des transitions d'entrée et de sortie en suivant des étapes spécifiques, telles que `enter`, `leave`, et d'autres.

Exemple simple d'utilisation de `<transition>` :

```
<template>
  <div>
    <button @click="show = !show">Toggle</button>
    <transition name="fade">
      <p v-if="show">Hello</p>
    </transition>
  </div>
</template>
```

Dans ce code, le composant `<transition>` enveloppe l'élément `<p>`. Lorsque l'utilisateur clique sur le bouton, la propriété `show` est mise à jour et l'élément `<p>` est ajouté ou supprimé avec une transition.

```
export default {
  data() {
    return {
      show: false,
    };
  },
};
```

Dans ce code, la propriété `show` est définie sur l'instance Vue. La propriété `show` est mise à jour lorsque l'utilisateur clique sur le bouton.

```
.fade-enter-active,
.fade-leave-active {
  transition: opacity 0.5s;
}

.fade-enter,
.fade-leave-to {
```

```
  opacity: 0;
}
```

Dans ce code, les classes CSS `fade-enter-active`, `fade-leave-active`, `fade-enter`, et `fade-leave-to` sont définies. Ces classes CSS sont utilisées pour définir les transitions d'entrée et de sortie.

10.2. Animation avec CSS

Vous pouvez également utiliser des animations CSS pour contrôler le comportement des transitions. Cela peut être fait en référençant des classes CSS spécifiques dans vos feuilles de style.

10.3. Utilisation de Bibliothèques d'Animation de Tiers

En utilisant le composant `<transition>` avec le mode `out-in` ou `in-out`, vous pouvez définir comment les composants entrent et sortent du DOM, permettant des transitions fluides entre différents états ou pages de votre application.

10.4. Utilisation des hooks

Vue.js offre la possibilité d'intercepter les moments clés des transitions avec des hooks JavaScript, vous donnant un contrôle total sur le déroulement des transitions et animations.

Chaque aspect des mixins, des plugins et des transitions dans Vue.js peut être ajusté avec précision pour créer une expérience utilisateur riche et interactive.

En utilisant les mixins, les plugins et les transitions dans Vue.js, vous pouvez créer des expériences utilisateur riches et interactives dans vos applications Vue.js.

Déploiement d'Applications Vue.js

Le déploiement est l'étape finale du cycle de développement d'une application, où le code est mis à disposition sur un serveur web et accessible aux utilisateurs. Vue.js, étant un framework principalement frontal, le déploiement concerne le code compilé statique (HTML, CSS, et JavaScript).

11.1. Préparation pour le déploiement

Avant de déployer votre application, il y a quelques étapes de préparation à suivre :

- **11.1.1. Build de production** : Exécutez le build de production avec Vue CLI, qui optimise et minimise votre application pour le déploiement.

  ```
  npm run build
  ```

  Cela crée un dossier dist/ contenant les fichiers statiques optimisés pour la production.

- **11.1.2. Tests** : Exécutez les tests pour vous assurer que votre application fonctionne correctement avant de la déployer.

```
npm run test
```

- **11.1.3. Linting** : Exécutez le linting pour vous assurer que votre code est conforme aux règles de style.

```
npm run lint
```

- **11.1.4. Révision de la configuration** : Vérifiez la configuration de votre application pour vous assurer qu'elle est correcte et qu'elle ne contient pas de données sensibles.

11.2. Déploiement sur un Serveur Web

Il existe plusieurs plateformes où vous pouvez déployer votre application Vue.js :

- **Services d'hébergement statique** : Comme Netlify, Vercel, GitHub Pages, ou Firebase Hosting.
- **Serveurs traditionnel**s : Vous pouvez utiliser n'importe quel serveur web capable de servir des fichiers statiques, comme Apache ou Nginx.
- **Services cloud** : AWS, Azure, ou Google Cloud Platform offrent des services pour héberger des applications web.

11.3. Automatisation du déploiement

Pour faciliter le processus de déploiement, vous pouvez mettre en place un pipeline CI/CD (Intégration Continue / Déploiement Continu) avec des outils comme Jenkins, GitLab CI/CD, ou GitHub Actions. Ceci automatisera le test et le déploiement de votre application à chaque fois que vous poussez du code sur la branche principale.

11.4. Mise à jour du domaine et du SSL

Si vous déployez votre application sur un nouveau domaine, vous devrez mettre à jour le domaine dans les paramètres de votre application. Si vous utilisez un certificat SSL, vous devrez également mettre à jour le domaine dans les paramètres de votre `certificat SSL`.

11.5. Surveillance et maintenance

Une fois déployée, l'application doit être surveillée pour s'assurer qu'elle fonctionne correctement. Des outils de surveillance peuvent vous aider à détecter et à répondre rapidement aux problèmes.

De plus, il est important de mettre à jour régulièrement les dépendances et d'appliquer les corrections de sécurité pour protéger votre application contre les vulnérabilités.

11.6. Exemple de fichier de configuration Nginx

Si vous déployez votre application Vue.js sur un serveur Nginx, vous pouvez utiliser le fichier de configuration suivant :

```
```

```
server {
    listen 80;
    server_name example.com www.example.com;

    location / {
        root /path/to/your/dist;
        try_files $uri $uri/ /index.html;
    }

    # Redirection vers HTTPS si vous avez SSL
    # listen 443 ssl;
    # ssl_certificate /path/to/your/fullchain.pem;
    # ssl_certificate_key /path/to/your/privkey.pem;
    # include /path/to/your/options-ssl-nginx.conf;
    # ssl_dhparam /path/to/your/ssl-dhparams.pem;
}
```

11.7. Déploiement d'une Application Vue.js avec GitHub Pages et GitHub Actions

Déployer une application Vue.js sur GitHub Pages peut être un moyen simple et efficace de partager votre projet avec le monde. En utilisant GitHub Actions, vous pouvez automatiser le processus de déploiement à chaque push ou merge dans votre branche principale. Voici les étapes à suivre :

- **11.7.1. Création d'un dépôt GitHub**
  - Assurez-vous que votre projet Vue.js est hébergé sur GitHub et que vous avez accès aux paramètres du repository.
  - Activez GitHub Pages dans les paramètres du repository et choisissez la branche qui servira à déployer votre site (par exemple, `gh-pages`).
- **11.7.2. Préparation de Votre Projet Vue.js**
  - Configurez le `vue.config.js` de votre projet pour définir la base publique de votre application. Si votre dépôt s'appelle `my-vue-app`, la base publique devrait être `/my-vue-app/`.

    ```
    module.exports = {
      publicPath: process.env.NODE_ENV === "production" ? "/my-vue-app/" : "/",
    };
    ```

  - Assurez-vous que votre projet est configuré pour produire un build de production dans un dossier que GitHub Pages peut utiliser (habituellement `dist`).

11.7.2 Création d'un Fichier de Workflow GitHub Actions

- À la racine de votre projet, créez un dossier `.github` puis un sous-dossier `workflows`.
- À l'intérieur du dossier `workflows`, créez un fichier de workflow, par exemple `deploy.yml`.

Voici un exemple de fichier `deploy.yml` pour le déploiement sur GitHub Pages :

```yaml
name: Deploy to GitHub Pages

on:
  push:
    branches:
      - main # Set this to your default branch

jobs:
  build-and-deploy:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout
        uses: actions/checkout@v2

      - name: Install and Build
        run: |
          npm install
          npm run build

      - name: Deploy to GitHub Pages
        uses: JamesIves/github-pages-deploy-action@releases/v3
        with:
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
          BRANCH: gh-pages # The branch the action should deploy to.
          FOLDER: dist # The folder the action should deploy.
```

11.7.3 Activation de l'Action et Déploiement

- Poussez le fichier de workflow dans votre dépôt pour activer GitHub Actions.
- Les changements seront construits et déployés automatiquement à chaque push.

11.7.4 Vérification et Débogage

- Consultez l'onglet "Actions" de votre dépôt pour vérifier que tout fonctionne.
- Votre site devrait être accessible à l'URL fournie par GitHub Pages.
- En cas d'erreur, utilisez les logs pour résoudre les problèmes.

11.7.5 Conseils Supplémentaires

- Ajoutez `.nojekyll` pour empêcher GitHub Pages de traiter votre site comme un projet Jekyll.
- Configurez un fichier `CNAME` dans votre dossier de déploiement pour les domaines personnalisés.

Suivez ces étapes pour une mise en ligne automatique et facile de votre application Vue.js.

Ressources et Communauté

Documentation Officielle

- Vue.js Docs
- Vuex Documentation
- Vue Router Documentation

Tutoriels et Articles

- Vue Mastery
- Vue School
- Dev.to Vue Tag

Communautés et Forums

- Vue.js Forum
- Vue.js Chat on Discord
- Vue Land on Discord
- Stack Overflow

Conclusion

En résumé, Vue.js est un framework progressif et flexible pour construire des interfaces utilisateur. Avec la communauté dynamique et les ressources abondantes, il est accessible aux débutants tout en étant puissant pour les développeurs expérimentés. En vous engageant avec la communauté et en profitant des ressources disponibles, vous pouvez continuer à apprendre, à partager et à contribuer à l'écosystème Vue.js.