Vue

3

Jour 2

Support réalisé par Pierrick Hauguel

01

Installation de Vue Router

Introduction à Vue Router

Vue Router est la bibliothèque officielle de routage pour Vue.js, permettant de créer des applications monopage avec des URL dynamiques.

```
import { createRouter, createWebHistory } from 'vue-router';

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

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

export default router;
```

Définir les routes de base

Les routes de base sont définies en associant des chemins d'URL à des composants.

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

const app = createApp(App);
app.use(router);
app.mount('#app');
```

Composants de route

Les composants de route sont rendus lorsque l'URL correspond à leur chemin défini.

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

```
<!-- Home.vue -->
<template>
  <h1>Accueil</h1>
</template>

<!-- About.vue -->
<template>
  <h1>À propos</h1>
</template>
```

Introduction au routage dynamique

Le routage dynamique permet de créer des routes qui incluent des paramètres, rendant les URL plus flexibles.

```
const routes = [
  { path: '/user/:id', component: User },
];
```

Syntaxe des routes dynamiques

Les routes dynamiques utilisent les paramètres dans le chemin pour rendre les composants dynamiques.

```
<!-- User.vue -->
<template>
  <div>User ID: {{ userId }}</div>
</template>

<script setup>
import { useRoute } from 'vue-router';

const route = useRoute();
const userId = route.params.id;
</script>
```

Routes dynamiques imbriquées

Les routes imbriquées permettent de structurer les routes hiérarchiquement, en utilisant des sous-routes.

```
const routes = [
  { path: '/user/:id', component: User, children: [
    { path: 'profile', component: UserProfile },
    { path: 'posts', component: UserPosts },
  ]},
];
```

Syntaxe de correspondance des routes

La correspondance des routes utilise des expressions pour définir des chemins plus complexes.

```
const routes = [
  { path: '/user/:id(\\d+)', component: User },
];
```

Routes nommées

Les routes peuvent être nommées pour faciliter la navigation et la gestion des routes.

```
const routes = [
  { path: '/user/:id', name: 'user', component: User },
];
```

Utilisation de noms de routes

Les noms de routes simplifient la navigation en permettant de référencer les routes par leur nom plutôt que par leur chemin.

```
<template>
  <router-link :to="{ name: 'user', params: { id: 123 }}">Go to User</router-link>
</template>
```

Vue imbriquée avec Nested Routes

Les vues imbriquées permettent d'afficher plusieurs composants de route dans une seule vue parent.

```
const routes = [
  { path: '/user/:id', component: User, children: [
    { path: 'profile', component: UserProfile },
    { path: 'posts', component: UserPosts },
  ]},
];
```

Vue imbriquée avec des noms de routes

Les noms de routes peuvent aussi être utilisés avec des routes imbriquées pour une navigation plus flexible.

```
const routes = [
  { path: '/user/:id', component: User, children: [
    { path: 'profile', name: 'user-profile', component: UserProfile },
    { path: 'posts', name: 'user-posts', component: UserPosts },
  ]},
];
```

Navigation programmée

La navigation programmée permet de naviguer vers différentes routes par programmation.

```
<script setup>
const router = useRouter();

const goToUser = (id) => {
  router.push({ name: 'user', params: { id } });
};
</script>

<template>
  <button @click="goToUser(123)">Go to User</button>
</template>
```

Utilisation du router et de la route

Les composables useRouter et useRoute fournissent des informations sur le routeur et la route actuelle.

Modes d'historique différents

Vue Router prend en charge différents modes d'historique pour gérer les URL, y compris le mode history et le mode hash.

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

Utilisation du mode history

Le mode history utilise l'API d'historique du navigateur pour gérer les URL sans le hash #.

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

Utilisation du mode hash

Le mode hash utilise le caractère # dans l'URL pour gérer les routes, ce qui est compatible avec des serveurs ne prenant pas en charge l'API d'historique.

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

Introduction aux gardes de navigation

Les gardes de navigation permettent d'exécuter des fonctions avant de naviguer vers une nouvelle route ou de quitter une route actuelle.

```
router.beforeEach((to, from, next) => {
  if (to.meta.requiresAuth && !isAuthenticated) {
    next('/login');
  } else {
    next();
  }
});
```

Gardes de navigation globales

Les gardes de navigation globales s'appliquent à toutes les routes de l'application et sont définies sur le routeur.

```
router.beforeEach((to, from, next) => {
  // Logic here
  next();
});
```

Gardes de navigation de composant

Les composants individuels peuvent également définir des gardes de navigation pour gérer leur propre logique de navigation.

```
<script setup>
import { onBeforeRouteLeave } from 'vue-router';

onBeforeRouteLeave((to, from, next) => {
  const answer = window.confirm('Do you really want to leave?');
  if (answer) {
    next();
  } else {
    next(false);
  }
});
</script>
```

Gardes de navigation avant et après

Vue Router propose des gardes de navigation avant et après pour exécuter des fonctions respectivement avant et après la navigation.

```
router.beforeEach((to, from, next) => {
  // Logic before navigation
  next();
});

router.afterEach((to, from) => {
  // Logic after navigation
});
```

Champs Meta de route

Les champs meta permettent de stocker des informations supplémentaires sur les routes, comme des exigences d'authentification.

```
const routes = [
  { path: '/dashboard', component: Dashboard, meta: { requiresAuth: true } },
];
```

Utilisation des champs Meta

Les champs meta peuvent être utilisés dans les gardes de navigation pour vérifier des conditions spécifiques avant d'accéder à une route.

```
router.beforeEach((to, from, next) => {
  if (to.meta.requiresAuth && !isAuthenticated) {
    next('/login');
  } else {
    next();
  }
});
```

Récupération des données avec Vue Router

Vue Router permet de récupérer des données avant la navigation pour s'assurer que les composants disposent des données nécessaires.

```
const routes = [
  {
    path: '/user/:id',
    component: User,
    beforeEnter: (to, from, next) => {
      fetchUserData(to.params.id).then(() => {
        next();
      });
    },
  },
];
```

Récupération des données avant la navigation

Les données peuvent être récupérées avant la navigation pour éviter de charger un composant sans les données nécessaires.

```
<!-- User.vue -->
<template>
  <div>
    <div>User ID: {{ userId }}</div>
    <div>User Data: {{ userData }}</div>
  </div>
</template>

<script setup>
import { ref, onMounted } from 'vue';
import { useRoute } from 'vue-router';

const route = useRoute();
const userId = ref(route.params.id);
const userData = ref(null);

// Récupérer les données à partir des paramètres de la route
userData.value = route.params.userData;
</script>
```

02

Les Watchers

Introduction aux Watchers

Les watchers permettent de réagir aux changements des données réactives. Ils sont utiles pour effectuer des opérations complexes ou asynchrones lorsque des données spécifiques changent.

```
<script setup>
import { ref, watch } from 'vue';

const data = ref(0);

watch(data, (newVal, oldVal) => {
  console.log(`Data changed from ${oldVal} to ${newVal}`);
});
</script>
```

Utilisation de watchEffect

watchEffect est un watcher qui s'exécute immédiatement et réagit automatiquement aux changements des dépendances réactives.

```
<script setup>
import { ref, watchEffect } from 'vue';

const data = ref(0);

watchEffect(() => {
  console.log(`Data is now ${data.value}`);
});
</script>
```

Watchers et Réactivité

Les watchers exploitent la réactivité de Vue pour surveiller les changements de données et exécuter des fonctions spécifiques en réponse.

```
<script setup>
import { ref, reactive, watch } from 'vue';

const state = reactive({ count: 0 });

watch(() => state.count, (newVal) => {
  console.log(`Count changed to ${newVal}`);
});
</script>
```

Syntaxe de base de watch

La syntaxe de base de watch comprend la source réactive à surveiller et une fonction callback à exécuter lorsque la source change.

```
<script setup>
import { ref, watch } from 'vue';

const message = ref('Hello');

watch(message, (newVal, oldVal) => {
  console.log(`Message changed from "${oldVal}" to "${newVal}"`);
});
</script>
```

Watchers multiples

Il est possible d'avoir plusieurs watchers pour surveiller différentes sources de données réactives dans un même composant.

```
<script setup>
import { ref, watch } from 'vue';

const count = ref(0);
const message = ref('Hello');

watch(count, (newVal) => {
  console.log(`Count is now ${newVal}`);
});

watch(message, (newVal) => {
  console.log(`Message is now ${newVal}`);
});
</script>
```

Surveillance des propriétés imbriquées

Les watchers peuvent surveiller des propriétés imbriquées d'objets réactifs, en utilisant une fonction pour accéder à la propriété à surveiller.

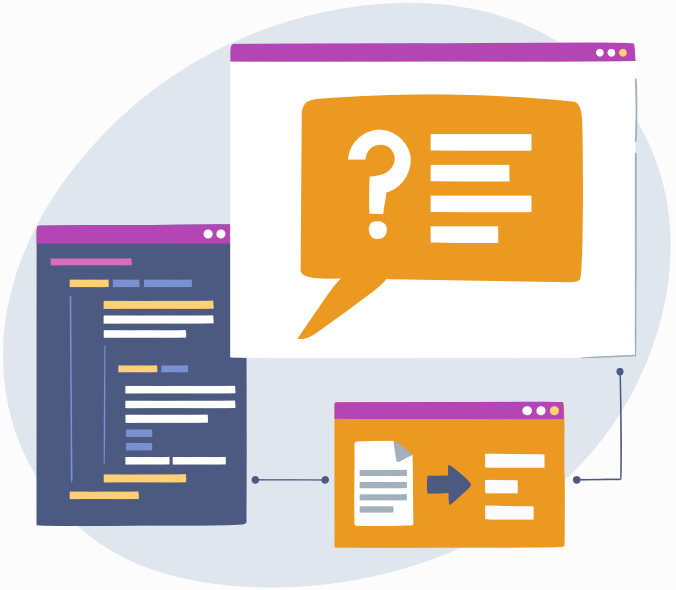
```
<script setup>
import { reactive, watch } from 'vue';

const user = reactive({ name: { first: 'John', last: 'Doe' } });

watch(() => user.name.last, (newVal) => {
  console.log(`Last name changed to ${newVal}`);
});
</script>
```

03

Les Computed



Introduction aux Computed Properties

Les propriétés calculées (computed) sont des valeurs dérivées de l'état réactif, mises en cache et recalculées uniquement lorsque leurs dépendances changent.

```
<script setup>
import { ref, computed } from 'vue';

const count = ref(1);
const double = computed(() => count.value * 2);
</script>
```

Différences entre Computed et Watch

computed est utilisé pour des valeurs dérivées mises en cache, tandis que watch est utilisé pour exécuter du code en réponse à des changements de données.

```
<script setup>
import { ref, computed, watch } from 'vue';

const count = ref(1);
const double = computed(() => count.value * 2);

watch(count, (newVal) => {
  console.log(`Count is now ${newVal}`);
});
</script>
```

Syntaxe de base des Computed Properties

Les propriétés calculées sont définies avec computed et peuvent être utilisées comme des variables réactives.

```
<script setup>
import { ref, computed } from 'vue';

const count = ref(3);
const triple = computed(() => count.value * 3);
</script>
```

Computed Properties et cache

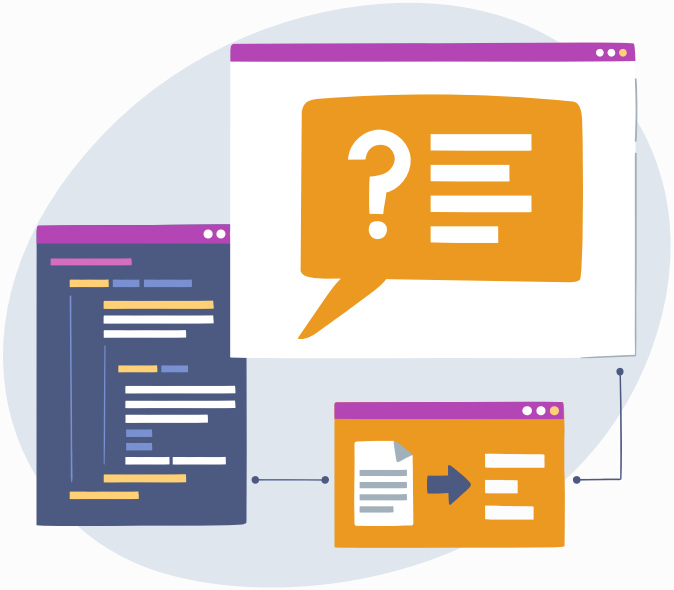
Les propriétés calculées sont mises en cache et ne sont recalculées que lorsque leurs dépendances réactives changent.

```
<script setup>
import { ref, computed } from 'vue';

const count = ref(4);
const quadruple = computed(() => count.value * 4);
</script>
```

04

Les Emits



Transfert d'événements avec emit

Les composants enfants peuvent envoyer des événements aux composants parents en utilisant emit.

```
<script setup>
const emit = defineEmits(['update']);

const updateParent = () => {
  emit('update', 'New Value');
};
</script>

<template>
  <button @click="updateParent">Update Parent</button>
</template>
```

Communication entre composants parents et enfants

Les composants parents et enfants peuvent communiquer via les props et les événements émis.

```vue
<!-- Parent.vue -->
<script setup>
import Child from './Child.vue';

const handleUpdate = (value) => {
  console.log(`Received from child: ${value}`);
};
</script>

<template>
  <Child @update="handleUpdate" />
</template>
```

```vue
<!-- Child.vue -->
<script setup>
const emit = defineEmits(['update']);

const updateParent = () => {
  emit('update', 'New Value');
};
</script>

<template>
  <button @click="updateParent">Update Parent</button>
</template>
```

Modèle de composant v-model

Le modèle v-model permet de lier une donnée du parent à une donnée du composant enfant de manière bidirectionnelle.

```
<template>
  <input :value="modelValue" @input="updateValue">
</template>

<script setup>
import { defineProps, defineEmits } from 'vue';

const props = defineProps({
  modelValue: String,
});

const emit = defineEmits(['update:modelValue']);

const updateValue = (event) => {
  emit('update:modelValue', event.target.value);
};
</script>
```

```
<template>
  <div>
    <ChildComponent v-model="parentValue" />
    <p>Valeur dans le composant parent: {{ parentValue }}</p>
  </div>
</template>

<script setup>
import { ref } from 'vue';
import ChildComponent from './ChildComponent.vue';

const parentValue = ref('');
</script>
```

05

Les Composables

Création de Composables

Les composables sont des fonctions qui encapsulent et réutilisent la logique réactive entre différents composants.

```
// useCounter.js
import { ref } from 'vue';

export function useCounter() {
  const count = ref(0);
  const increment = () => {
    count.value++;
  };

  return { count, increment };
}
```

Avantages des Composables

Les composables permettent une meilleure réutilisabilité et organisation de la logique réactive, rendant le code plus propre et maintenable.

```
<script setup>
import { useCounter } from './useCounter';

const { count, increment } = useCounter();
</script>

<template>
  <button @click="increment">Count is {{ count }}</button>
</template>
```

Partage de la logique réactive avec les Composables

Les composables permettent de partager facilement la logique réactive entre plusieurs composants.

```
<script setup>
import { useCounter } from './useCounter';

const { count, increment } = useCounter();
</script>

<template>
  <button @click="increment">Count is {{ count }}</button>
</template>
```

Composables et réactivité

Les composables utilisent la réactivité de Vue pour gérer et partager l'état de manière efficace.

```
// useToggle.js
import { ref } from 'vue';

export function useToggle(initial = false) {
  const state = ref(initial);
  const toggle = () => {
    state.value = !state.value;
  };

  return { state, toggle };
}
```

Exemples de Composables

Voici quelques exemples de composables pour des cas d'utilisation courants comme la gestion de compteurs ou de formulaires.

```
// useForm.js
import { ref } from 'vue';

export function useForm() {
  const form = ref({
    name: '',
    email: ''
  });

  const submit = () => {
    console.log(`Form submitted with: ${JSON.stringify(form.value)}`);
  };

  return { form, submit };
}
```

Utilisation de Composables dans les composants

Les composables peuvent être facilement utilisés dans les composants Vue pour partager la logique réactive.

```
<script setup>
import { useForm } from './useForm';

const { form, submit } = useForm();
</script>

<template>
  <form @submit.prevent="submit">
    <input v-model="form.name" placeholder="Name" />
    <input v-model="form.email" placeholder="Email" />
    <button type="submit">Submit</button>
  </form>
</template>
```

Composables et gestion d'état

Les composables peuvent être utilisés pour gérer l'état de l'application de manière centralisée et réactive.

```
// useStore.js
import { ref } from 'vue';

export function useStore() {
  const state = ref({
    user: { name: 'Alice' }
  });

  const setUser = (name) => {
    state.value.user.name = name;
  };

  return { state, setUser };
}
```

06

Les Directives custom

Création de directives personnalisées

Les directives personnalisées permettent d'ajouter un comportement personnalisé aux éléments du DOM.

```
// v-focus.js
import { ref, onMounted } from 'vue';

export const vFocus = {
  mounted(el) {
    el.focus();
  }
};
```

Utilisation de directives dans les composants

Les directives personnalisées peuvent être utilisées dans les composants pour ajouter des comportements spécifiques aux éléments du DOM.

```
<script setup>
import { vFocus } from './directives/v-focus';
</script>

<template>
  <input v-focus />
</template>
```

Directives et réactivité

Les directives peuvent interagir avec les données réactives pour modifier dynamiquement le comportement des éléments du DOM.

```
<script setup>
import { ref } from 'vue';

const isActive = ref(false);
</script>

<template>
  <div v-if="isActive" v-focus></div>
</template>
```

Exemples de directives personnalisées

Voici quelques exemples de directives personnalisées pour des cas d'utilisation courants comme la mise au point automatique ou la gestion de la visibilité.

```
// v-show.js
export const vShow = {
  beforeMount(el, binding) {
    el.style.display = binding.value ? '' : 'none';
  },
  updated(el, binding) {
    el.style.display = binding.value ? '' : 'none';
  }
};
```

Directives et Composition API

Les directives peuvent être créées et utilisées avec la Composition API pour ajouter des comportements dynamiques aux composants.

```
<script setup>
import { ref, onMounted } from 'vue';

const show = ref(true);

onMounted(() => {
  show.value = false;
});
</script>

<template>
  <div v-show="show">Visible</div>
</template>
```

Inscription globale des directives

Les directives peuvent être enregistrées globalement, rendant leur utilisation possible dans n'importe quel composant du projet.

```
import { createApp } from 'vue';
import App from './App.vue';
import { vFocus } from './directives/v-focus';

const app = createApp(App);
app.directive('focus', vFocus);
app.mount('#app');
```

07

Validation de formulaire avec Zod

Introduction à Zod

Zod est une bibliothèque de validation de schémas TypeScript-first qui permet de valider facilement les objets JavaScript.

```
import { z } from 'zod';

const userSchema = z.object({
  name: z.string(),
  age: z.number(),
});
```

Création d'un schéma de validation de base

Un schéma de validation de base avec Zod peut être créé pour valider différents types de données.

```
const schema = z.object({
  username: z.string(),
  password: z.string().min(8),
});
```

Validation des emails

Zod inclut des validations pour les emails afin de s'assurer qu'ils respectent le format standard.

```
const emailSchema = z.string().email('Email non valide');
```

Validation des champs conditionnels

Les champs conditionnels peuvent être validés en fonction des valeurs d'autres champs.

```
const schema = z.object({
  password: z.string().min(8),
  confirmPassword: z.string().min(8),
}).refine(data => data.password === data.confirmPassword, {
  message: "Les mots de passe doivent correspondre",
  path: ["confirmPassword"],
});
```

Exemple de vérification de champs dans un composant Vue avec Zod

Intégrer Zod dans un composant Vue pour valider les champs d'un formulaire. Cet exemple montre comment utiliser un schéma Zod pour vérifier les entrées de l'utilisateur et afficher les erreurs de validation.

```vue
<template>
  <div>
    <form @submit.prevent="handleSubmit">
      <div>
        <label for="name">Nom:</label>
        <input v-model="formData.name" id="name" />
        <span v-if="errors.name">{{ errors.name }}</span>
      </div>
      <div>
        <label for="email">Email:</label>
        <input v-model="formData.email" id="email" />
        <span v-if="errors.email">{{ errors.email }}</span>
      </div>
      <div>
        <label for="age">Âge:</label>
        <input v-model="formData.age" id="age" type="number" />
        <span v-if="errors.age">{{ errors.age }}</span>
      </div>
      <button type="submit">Soumettre</button>
    </form>
  </div>
</template>
```

```vue
<script setup>
import { ref } from 'vue';
import { z } from 'zod';

// Définition du schéma de validation avec Zod
const formSchema = z.object({
  name: z.string().min(1, "Le nom est requis"),
  email: z.string().email("Email non valide"),
  age: z.number().min(0, "L'âge doit être positif"),
});

// État du formulaire et erreurs
const formData = ref({
  name: '',
  email: '',
  age: null,
});

const errors = ref({});
```

```vue
// Gestion de la soumission du formulaire
const handleSubmit = () => {
  const result = formSchema.safeParse(formData.value);

  if (result.success) {
    // Les données sont valides
    console.log("Formulaire soumis avec succès :", result.data);
    errors.value = {};
  } else {
    // Les données ne sont pas valides
    errors.value = result.error.format();
  }
};
</script>
```

Création de types TypeScript à partir de schémas Zod

Les types TypeScript peuvent être créés directement à partir des schémas Zod pour garantir la cohérence des types.

```
const schema = z.object({
  title: z.string(),
});

type Schema = z.infer<typeof schema>;
```

Exemples pratiques d'inférence de schéma

Voici des exemples pratiques d'utilisation de l'inférence de schéma avec Zod et TypeScript.

```
const productSchema = z.object({
  id: z.number(),
  name: z.string(),
  price: z.number().positive(),
});

type Product = z.infer<typeof productSchema>;

const product: Product = {
  id: 1,
  name: "Laptop",
  price: 999.99,
};
```

08

Introduction à Pinia

Introduction à Pinia

Pinia est une bibliothèque de gestion d'état pour Vue.js, conçue comme une alternative moderne à Vuex, offrant une API intuitive et des fonctionnalités avancées.

Installation et configuration de Pinia

Pour utiliser Pinia, il faut l'installer et le configurer dans votre projet Vue.

```
npm install pinia
```

```
import { createApp } from 'vue';
import App from './App.vue';
import pinia from './pinia';

const app = createApp(App);
app.use(pinia);
app.mount('#app');
```

Définition des états (state) dans Pinia

Les états sont définis dans la propriété state du store et contiennent les données réactives de l'application.

```
import { defineStore } from 'pinia';

export const useStore = defineStore('main', {
  state: () => ({
    count: 0,
  }),
});
```

Création de getters dans Pinia

Les getters sont des fonctions dérivées des états, utilisées pour calculer des valeurs basées sur l'état du store.

```
import { defineStore } from 'pinia';

export const useStore = defineStore('main', {
  state: () => ({
    count: 0,
  }),
  getters: {
    doubleCount: (state) => state.count * 2,
  },
});
```

Utilisation des actions dans Pinia

Les actions sont des méthodes utilisées pour modifier l'état du store de manière synchrone ou asynchrone.

```
import { defineStore } from 'pinia';

export const useStore = defineStore('main', {
  state: () => ({
    count: 0,
  }),
  actions: {
    increment() {
      this.count++;
    },
  },
});
```

Utilisation de Pinia avec la Composition API

La Composition API permet d'utiliser Pinia de manière plus flexible et modulaire dans les composants Vue.

```
<script setup>
import { useStore } from './store';

const store = useStore();

function incrementCount() {
  store.increment();
}
</script>

<template>
  <div>
    <p>Count: {{ store.count }}</p>
    <button @click="incrementCount">Increment</button>
  </div>
</template>
```

Introduction aux Stores Setup dans Pinia

Les stores setup dans Pinia utilisent la Composition API pour définir l'état, les actions et les getters dans une fonction setup, offrant une flexibilité accrue.

```
import { defineStore } from 'pinia';

export const useStore = defineStore('main', () => {
  const count = ref(0);
  const doubleCount = computed(() => count.value * 2);
  function increment() {
    count.value++;
  }

  return { count, doubleCount, increment };
});
```

Définition des États dans un Store Setup

Les états sont définis en utilisant ref ou reactive dans la fonction setup du store.

```
import { defineStore } from 'pinia';

export const useStore = defineStore('main', () => {
  const state = reactive({
    count: 0,
    name: 'Vue.js',
  });

  return { state };
});
```

Création de Getters dans un Store Setup

Les getters sont définis en utilisant computed pour calculer des valeurs dérivées de l'état.

```
import { defineStore } from 'pinia';

export const useStore = defineStore('main', () => {
  const count = ref(0);
  const doubleCount = computed(() => count.value * 2);

  return { count, doubleCount };
});
```

Utilisation des Actions dans un Store Setup

Les actions sont définies comme des fonctions dans la fonction setup et peuvent muter l'état directement.

```
import { defineStore } from 'pinia';

export const useStore = defineStore('main', () => {
  const count = ref(0);

  function increment() {
    count.value++;
  }

  return { count, increment };
});
```

Accès au Store dans les Composants avec Setup

Les stores setup peuvent être accédés et utilisés dans les composants Vue en les important et en appelant la fonction store.

```
<script setup>
import { useStore } from './store';

const store = useStore();
</script>

<template>
  <div>
    <p>Count: {{ store.count }}</p>
    <button @click="store.increment">Increment</button>
  </div>
</template>
```

Utilisation de storeToRefs dans Pinia

storeToRefs est une fonction de Pinia qui permet de transformer les propriétés d'un store en références réactives individuelles. Cela facilite la réactivité et l'accès aux propriétés du store dans les composants Vue.

```
<script setup>
import { useStore } from './store';
import { storeToRefs } from 'pinia';

// Utilisation du store
const store = useStore();
const { count, doubleCount } = storeToRefs(store);

function incrementCount() {
  store.increment();
}
</script>

<template>
  <div>
    <p>Count: {{ count }}</p>
    <p>Double Count: {{ doubleCount }}</p>
    <button @click="incrementCount">Increment</button>
  </div>
</template>
```