



ADVANCED VUE 3: FROM JUNIOR TO EVAN YOU LEVEL

Pierrick Hauguel



L'API de Composition : rappels

- La Composition API est une nouvelle façon d'écrire des composants en Vue.js 3.
- Elle permet une meilleure organisation du code en séparant les préoccupations liées à l'état, aux effets et aux calculs.
- La Composition API est basée sur le concept de composition de fonctions.

Option

```
<template>
  <h1></h1>
  <button @click="incrCounter">Click Me</button>
</template>
<script>
export default {
  data() {
    return {
      counter: 0
    }
  },
  methods: {
    incrCounter: function () {
      this.counter += 1
    }
  }
}
</script>
```

Composition

```
<template>
  <h1></h1>
  <button @click="incrCounter">Click Me</button>
</template>
<script setup>
import { ref } from "vue";

let counter = ref(0);

const incrCounter = function () {
  counter.value += 1;
}
</script>
```

Quelles différences ?

- > Nous avons importé quelque chose appelé "ref"
cela nous permet de créer des variables réactives
- > Lorsque nous augmentons le compteur, nous augmentons en fait counter.value, car "ref" renvoie un objet.
- > Nous évitons d'avoir à utiliser un prototype entier, et n'avons à la place qu'une seule fonction "incrCounter"

Quels avantages ?

- > Meilleure prise en charge des types , car il utilise principalement des fonctions et des variables standards.
- > Fichiers plus petits, l'API de composition nécessite moins de code.

Petite note : quelles différences entre ref et reactive ?

-> Une différence simple : reactive n'accepte que les objets contrairement à ref.

```
const x = reactive({ name: "John" });
x.name = "Ammy";
// x -> { name: 'Ammy' }

const x = ref({ name: "John" });
x.value.name = "Ammy";
// x.value -> { name: 'Ammy' }
```

-> Impossible de changer l'instance d'un reactive

```
import { reactive, ref } from 'vue';

// INVALID - changes of x are NOT recorded by Vue
let x = reactive({ name: "John" });
x = reactive({ todo: true });

// VALID
const x = ref({ name: "John" });
x.value = { todo: true };

```

Pourquoi donc utiliser reactive ?



- > reactive() suit chaque propriété individuellement par opposition à ref()
- > chaque propriété dans reactive() est traitée de façon autonome comme une ref() lorsque Vue essaie de déterminer si elle doit mettre à jour quelque chose qui en dépend.

Parce que ref est (presque) une arnaque

```
1  class RefImpl<T> {
2    private _value: T
3    private _rawValue: T
4
5    public dep?: Dep = undefined
6    public readonly __v_isRef = true
7
8    constructor(value: T, public readonly __v_isShallow: boolean) {
9      this._rawValue = __v_isShallow ? value : toRaw(value)
10     this._value = __v_isShallow ? value : toReactive(value)
11   }
12
13   get value() {
14     trackRefValue(this)
15     return this._value
16   }
17
18   set value(newVal) {
19     newVal = this.__v_isShallow ? newVal : toRaw(newVal)
20     if (hasChanged(newVal, this._rawValue)) {
21       this._rawValue = newVal
22       this._value = this.__v_isShallow ? newVal : toReactive(newVal)
23       triggerRefValue(this, newVal)
24     }
25   }
26 }
```



Les composables

Les composables sont des fonctions qui exploitent l'API de Composition de Vue pour encapsuler et réutiliser la logique d'état

```
// mouse.js
import { ref, onMounted, onUnmounted } from 'vue'

export function useMouse() {
  const x = ref(0)
  const y = ref(0)

  function update(event) {
    x.value = event.pageX
    y.value = event.pageY
  }

  onMounted(() => window.addEventListener('mousemove', update))
  onUnmounted(() => window.removeEventListener('mousemove', update))

  return { x, y }
}
```



Utilisation des Composables

Les composables permettent une réutilisation aisée de la logique dans différents composants

```
<script setup>
import { useMouse } from './mouse.js'

const { x, y } = useMouse()
</script>

<template>Mouse position is at: {{ x }}, {{ y }}</template>
```



Les valeurs calculées

Les propriétés calculées permettent de créer des propriétés dérivées qui seront recalculées seulement lorsque les dépendances auront changé.

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

const a = ref(1);
const b = ref(2);

const sum = computed(() => {
  return a.value + b.value;
});

</script>
```



inject / provide

Vue 3 propose un mécanisme pour la communication entre les composants parent-enfant via provide et inject.

-> provide permet à un composant d'exposer une propriété à ses descendants.

```
// Composant Parent
<template>
  ...
</template>

<script>
import { provide } from 'vue';

export default {
  setup() {
    const message = 'Bonjour le monde!';
    provide('message', message);
  }
}
</script>
```



inject / provide

-> inject permet aux composants descendants d'accéder à cette propriété.

```
// Composant Enfant
<template>
  <p>{{ message }}</p>
</template>

<script>
import { inject } from 'vue';

export default {
  setup() {
    const message = inject('message');
    return { message };
  }
}
</script>
```



Vue router : Un des meilleurs !

Un peu de syntaxe :

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



Vue router : History mode

History Mode Activation :

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

const router = createRouter({
  history: createWebHistory(),
  routes: [
    //...
  ],
})
```



Vue router : Les paramètres d'URL

- Vue Router 4 offre la capacité de définir des paramètres dans les URL, permettant ainsi une correspondance dynamique de routes.
- Syntaxe inspirée par celle utilisée par Express, supportant des motifs de correspondance avancés tels que des paramètres optionnels, des exigences zéro ou plus / un ou plus, et même des motifs regex personnalisés.

```
// Exemple de base
import { createRouter, createWebHistory } from 'vue-router'

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

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



Vue router : Accès aux Paramètres d'URL

- Les paramètres d'URL peuvent être accédés via la propriété params du routeur.

```
// Dans le composant User
import { ref, onBeforeRouteUpdate } from 'vue';
import { useRoute } from 'vue-router';

export default {
  setup() {
    const route = useRoute();
    const userId = ref(route.params.id);

    onBeforeRouteUpdate((to, from) => {
      userId.value = to.params.id;
    });

    return { userId };
  }
}
```



Vue router : Navigation Programmatique avec Paramètres d'URL

- Pour spécifier des paramètres lors de la navigation programmatique, fournissez une chaîne ou un nombre (ou un tableau de ces éléments pour des paramètres répétables). Les autres types seront automatiquement convertis en chaînes.

```
// Exemple
import { ref } from 'vue';
import { useRouter } from 'vue-router';

export default {
  setup() {
    const router = useRouter();
    const navigateToUser = () => {
      router.push({ name: 'User', params: { id: 123 }});
    };

    return { navigateToUser };
  }
}
```



Vue router : Utilisation Avancée des Paramètres d'URL

- Les paramètres d'URL peuvent aussi être utilisés avec des expressions régulières pour des correspondances plus complexes.

```
// Exemple

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

const routes = [
    // correspondra à tout et le mettra sous '$route.params.pathMatch'
    { path: '/:pathMatch(.*)*', name: 'NotFound', component: NotFound },
]

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



Vue router : Routes imbriquées / Nested routes

History Mode Activation :

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

const router = createRouter({
  history: createWebHistory(),
  routes: [
    //...
  ],
})
```



Vue router : Utilisation de router.onError

Cette méthode permet de capturer les erreurs survenant lors du routage, par exemple, lors d'un échec de navigation

```
// Handle router errors
import { createRouter } from 'vue-router'

const router = createRouter({ /* options */ })

router.onError((err) => {
  // Handle the router error here
  console.error("Router error:", err);
  // Add code for reporting or other error handling logic
});
```



Vue router : Route générique pour capturer les routes non définies

Cette méthode permet de capturer les erreurs survenant lors du routage, par exemple, lors d'un échec de navigation

```
// Handle router errors
import { createRouter } from 'vue-router'

const router = createRouter({ /* options */ })

router.onError((err) => {
  // Handle the router error here
  console.error("Router error:", err);
  // Add code for reporting or other error handling logic
});
```



Vue router : Route générique pour capturer les routes non définies

```
// Add a wildcard route
const routes = [
  // ...other routes,
  { path: '/:pathMatch(.*)', redirect: { name: 'NotFound' } },
]

const router = createRouter({
  // ...other options,
  routes,
})
```



Vue router : Gestion des Échecs de Navigation

```
router.push('/path').catch(err => {
  // Handle navigation failure
  console.error('Navigation failed:', err);
});
```



Les options des guards

1.to - Route de destination

- L'objet to contient des informations sur la route de destination.
- Vous pouvez accéder aux propriétés telles que path, params, query, etc.

2.from - Route d'origine

- L'objet from contient des informations sur la route d'origine.
- Vous pouvez accéder aux propriétés telles que path, params, query, etc.

3.next - Fonction de contrôle de la navigation

- La fonction next est appelée pour permettre ou annuler la navigation.
- Utilisez next() pour permettre la navigation.
- Utilisez next(false) pour annuler la navigation.
- Utilisez next('/path') pour rediriger vers une autre route.



Les cas d'utilisation des guards

1. Authentification requise

1. Utilisation d'un garde pour vérifier si l'utilisateur est authentifié avant de permettre la navigation vers des routes protégées.

2. Autorisation des rôles

1. Utilisation d'un garde pour vérifier si l'utilisateur a les autorisations nécessaires avant de permettre la navigation vers certaines routes.

3. Validation des formulaires

1. Utilisation d'un garde pour vérifier si les données d'un formulaire sont valides avant de permettre la navigation.

4. Gestion des états non sauvegardés

1. Utilisation d'un garde pour afficher une confirmation à l'utilisateur s'il quitte une page avec des modifications non sauvegardées.



Vue router : Utilisation de onBeforeRouteLeave

```
import { onBeforeRouteLeave } from 'vue-router';

export default {
  setup() {
    onBeforeRouteLeave((to, from) => {
      const answer = window.confirm('Voulez-vous vraiment quitter ? Vous avez des modifications non enregistrées !')
      if (!answer) return false; // Annuler la navigation et rester sur la même page
    });
  },
};
```

Vue router : Utilisation de onBeforeRouteUpdate

```
import { onBeforeRouteUpdate, ref } from 'vue-router';

export default {
  setup() {
    const userData = ref();

    onBeforeRouteUpdate(async (to, from) => {
      if (to.params.id !== from.params.id) { // Vérifier si l'ID utilisateur a changé
        userData.value = await fetchUser(to.params.id); // Récupérer les données utilisateur
      }
    });
  },
};
```



Vue router : Gards Globaux

```
import { createRouter } from 'vue-router'

const router = createRouter({
  // ...options
})

// Gard global exécuté avant chaque navigation
router.beforeEach((to, from, next) => {
  // ... logique de gard
  next() // procéder à la navigation
})

// Gard global exécuté avant la confirmation de chaque navigation, mais après tous les gards de composant
router.beforeResolve((to, from, next) => {
  // ... logique de gard
  next() // procéder à la navigation
})

// Gard global exécuté après chaque navigation confirmée
router.afterEach((to, from) => {
  // ... logique de gard
})
```



Vue router : Gards par route

```
const routes = [
  {
    path: '/protected',
    component: ProtectedComponent,
    beforeEnter: (to, from, next) => {
      // ... logique de gard
      next() // procéder à la navigation
    },
    // ...autres routes
  }

const router = createRouter({
  // ...options
  routes,
})
```



Vue router : Gards par route

```
const routes = [
  {
    path: '/protected',
    component: ProtectedComponent,
    beforeEnter: (to, from, next) => {
      if (isUserAuthenticated()) {
        next() // utilisateur authentifié, procéder à la navigation
      } else {
        next('/login') // utilisateur non authentifié, rediriger vers la page de connexion
      }
    },
    // ...autres routes
  }
]

const router = createRouter({
  // ...options
  routes,
})
```



Définition et utilité des slots

- Introduction au concept des slots dans les frameworks JavaScript/Vue.js.
- Les slots permettent d'injecter du contenu dynamiquement dans les composants.
- Ils offrent une flexibilité accrue en termes de personnalisation et de réutilisation des composants.



Les slots : Définition et utilité des slots

Les slots vont nous permettre de passer des enfants à notre composant.

```
1 <template>
2   <button class="fancy-btn">
3     <slot />
4     <!-- slot outlet -->
5   </button>
6 </template>
7
8 <style>
9   .fancy-btn {
10     color: #fff;
11     background: linear-gradient(315deg, #42d392 25%, #647eff);
12     border: none;
13     padding: 5px 10px;
14     margin: 5px;
15     border-radius: 8px;
16     cursor: pointer;
17   }
18 </style>
```

```
1 <template>
2   <FancyButton>
3     Click me
4     <!-- slot content -->
5   </FancyButton>
6 </template>
7
```



Injection de contenu dans un template

Les slots permettent d'injecter du contenu dans le template d'un composant.

```
<template>
  <div>
    <h1>Titre du composant</h1>
    <slot></slot>
  </div>
</template>
```

Le contenu passé entre les balises `<slot></slot>` sera inséré à l'endroit où le composant est utilisé.



Slots et composants génériques



Les slots sont particulièrement utiles pour créer des composants génériques.

```
<template>
  <button>
    <slot></slot>
  </button>
</template>
```

Ce composant de bouton générique permet d'injecter du contenu personnalisé à l'intérieur du bouton.



La directive v-slot

La directive v-slot est utilisée pour déclarer et utiliser des slots dans les templates Vue.js.

```
<template>
  <div>
    <slot name="content"></slot>
  </div>
</template>

<template>
  <div>
    <div v-slot:content>
      Contenu du slot
    </div>
  </div>
</template>
```

Le slot avec le nom "content" est déclaré avec `<slot name="content"></slot>`.

Dans le composant parent, on utilise la directive v-slot pour assigner du contenu au slot nommé.



Les slots nommés

Les slots peuvent être nommés pour permettre l'injection de contenu spécifique dans plusieurs emplacements.

Ce composant a trois slots nommés : "header", "default" (utilisé pour le contenu principal) et "footer".

Lorsqu'il est utilisé, le contenu peut être assigné à un slot spécifique en utilisant la directive v-slot avec le nom approprié.

```
<template>
  <div>
    <header>
      <slot name="header"></slot>
    </header>
    <main>
      <slot></slot>
    </main>
    <footer>
      <slot name="footer"></slot>
    </footer>
  </div>
</template>
```



La portée des slots

Les slots peuvent également être utilisés pour transmettre des données entre composants.

La donnée "myData" est passée au slot en utilisant une propriété dynamique :data.

Le composant parent peut ensuite accéder à cette donnée à l'intérieur du slot.

```
<template>
  <div>
    <slot :data="myData"></slot>
  </div>
</template>
```



Les slots à nom dynamique

Il est possible d'utiliser des noms de slots dynamiques en utilisant des expressions JavaScript.

```
<template>
  <div>
    <slot :name="slotName"></slot>
  </div>
</template>

<template>
  <div>
    <div v-for="slot in slots" :key="slot.name">
      <slot :name="slot.name">{{ slot.content }}</slot>
    </div>
  </div>
</template>
```



Cas d'usage

Cas d'usage sur un composant tableau dynamique

```
<template>
  <div>
    <table>
      <thead>
        <tr>
          <!-- Utilisation d'un v-for pour générer les en-têtes de colonne -->
          <th v-for="column in columns" :key="column.name">
            {{ column.label }}
          </th>
        </tr>
      </thead>
      <tbody>
        <!-- Utilisation d'un v-for pour générer les lignes du tableau -->
        <tr v-for="item in items" :key="item.id">
          <!-- Utilisation d'un v-for pour générer les cellules de la ligne -->
          <td v-for="column in columns" :key="column.name">
            <!-- Utilisation du slot à nom dynamique correspondant à la colonne -->
            <slot :name="column.name" :item="item">
              {{ item[column.name] }}
            </slot>
          </td>
        </tr>
      </tbody>
    </table>
  </div>
</template>

<script>
export default {
  props: {
    columns: {
      type: Array,
      required: true,
    },
    items: {
      type: Array,
      required: true,
    },
  },
}
</script>
```

```
<template>
  <div>
    <my-table :columns="tableColumns" :items="tableData">
      <!-- Slot à nom dynamique pour personnaliser la colonne "name" -->
      <template v-slot:name="{ item }">
        <strong>{{ item.name }}</strong>
      </template>

      <!-- Slot à nom dynamique pour personnaliser la colonne "age" -->
      <template v-slot:age="{ item }">
        {{ item.age }} ans
      </template>

      <!-- Slot à nom dynamique pour personnaliser la colonne "email" -->
      <template v-slot:email="{ item }">
        <a :href="'mailto:' + item.email">{{ item.email }}</a>
      </template>
    </my-table>
  </div>
</template>

<script>
import MyTable from './MyTable.vue';

export default {
  components: {
    MyTable,
  },
  data() {
    return {
      tableColumns: [
        { name: 'name', label: 'Nom' },
        { name: 'age', label: 'Âge' },
        { name: 'email', label: 'Email' },
      ],
      tableData: [
        { id: 1, name: 'John Doe', age: 25, email: 'john@example.com' },
        { id: 2, name: 'Jane Smith', age: 32, email: 'jane@example.com' },
        { id: 3, name: 'Bob Johnson', age: 45, email: 'bob@example.com' },
      ],
    };
  },
}
```



Utilité de composants asynchrones

- Dans les applications Vue 3, les composants asynchrones jouent un rôle clé pour améliorer les performances et l'expérience utilisateur.
- Ils permettent de retarder le chargement des composants non essentiels jusqu'à ce qu'ils soient réellement nécessaires.
- Cela réduit le temps de chargement initial de l'application et améliore la réactivité globale.



Chargement des composants à la demande

- Le chargement des composants à la demande consiste à charger dynamiquement un composant uniquement lorsque celui-ci est requis par l'application.
- Cela permet d'optimiser les performances en évitant de charger tous les composants en une seule fois.
- La composition API de Vue 3 fournit une fonction appelée "defineAsyncComponent" pour faciliter le chargement asynchrone des composants.

```
// Définition d'un composant asynchrone
const AsyncComponent = defineAsyncComponent(() => import('./AsyncComponent.vue'))

// Utilisation du composant asynchrone
<template>
  <AsyncComponent />
</template>
```



Configuration

```
<script setup>
import { defineAsyncComponent } from 'vue'

const AdminPage = defineAsyncComponent(() =>
  import('./components/AdminPageComponent.vue')
)
</script>

<template>
  <AdminPage />
</template>
```

```
const AsyncComp = defineAsyncComponent({
  // La fonction de chargement
  loader: () => import('./Foo.vue'),
  // Un composant à utiliser pendant le chargement du composant asynchrone
  loadingComponent: LoadingComponent,
  // Délai avant l'affichage du composant de chargement. Par défaut : 200 ms.
  delay: 200,
  // Un composant à utiliser si le chargement échoue
  errorComponent: ErrorComponent,
  // Le composant d'erreur sera affiché si un délai d'attente est
  // fourni et dépassé. Par défaut : Infini.
  timeout: 3000
})
```



Exemple de composant asynchrone

```
// AsyncComponent.vue

<template>
  <div>
    <h2>Composant Asynchrone</h2>
    <p>{{ message }}</p>
  </div>
</template>

<script>
export default {
  data() {
    return {
      message: '',
    };
  },
  async created() {
    try {
      const response = await fetch('https://api.example.com/data');
      const data = await response.json();
      this.message = data.message;
    } catch (error) {
      console.error(error);
      this.message = 'Erreur lors du chargement des données';
    }
  },
}
</script>
```

Gestion d'erreur

```
<template>
  <div>
    <h1>Mon application Vue</h1>
    <div v-if="loading">Chargement en cours...</div>
    <div v-else-if="error">Erreur lors du chargement du composant</div>
    <div v-else>
      <AsyncComponent />
    </div>
  </div>
</template>

<script>
import { ref } from 'vue';
const AsyncComponent = defineAsyncComponent(() => import('./AsyncComponent.'))

export default {
  setup() {
    const loading = ref(true);
    const error = ref(false);

    AsyncComponent().then(() => {
      loading.value = false;
    }).catch(() => {
      loading.value = false;
      error.value = true;
    });
  }

  return {
    loading,
    error,
  };
}
</script>
```





Gestion d'erreur



Vuex ou Pinia ?



Pinia

"Type Safe, Extensible, and Modular by design. Forget you are even using a store."



Initialisation du store Pinia

Importez Pinia dans le fichier principal main.js.

```
import { createApp } from 'vue'  
import { createPinia } from 'pinia'  
import App from './App.vue'  
  
const app = createApp(App)  
const pinia = createPinia()  
app.use(pinia)  
app.mount('#app')
```



Création d'un store basique

Créez un fichier store.js et déclarez un store avec quelques états et actions simples.

```
import { defineStore } from 'pinia'

export const useCounterStore = defineStore('counter', {
  state: () => ({
    count: 0,
  }),
  actions: {
    increment() {
      this.count++
    },
    decrement() {
      this.count--
    },
  }
})
```



Utilisation du store dans un composant Vue

Importez le store dans un composant Vue et utilisez les méthodes et les états dans le template et les méthodes.

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

<script>
import { useCounterStore } from './store'

export default {
  setup() {
    const counterStore = useCounterStore()

    return {
      count: counterStore.count,
      increment: counterStore.increment,
      decrement: counterStore.decrement,
    }
  },
}
</script>
```



Utilisation des getters et des actions (mutations)

Utilisez les getters pour récupérer les états du store et les mutations pour effectuer des modifications asynchrones sur les états.

```
export const useCounterStore = defineStore('counter', {
  state: () => ({
    count: 0,
  }),
  getters: {
    doubleCount() {
      return this.count * 2
    },
  },
  actions: {
    increment() {
      this.count++
    },
    decrement() {
      this.count--
    },
  },
})
```



Utilisation des actions avec des paramètres

Utilisez des paramètres dans les actions pour effectuer des opérations spécifiques.

```
export const useCounterStore = defineStore('counter', {  
  state: () => {  
    count: 0,  
  },  
  actions: {  
    incrementBy(value) {  
      this.count += value  
    },  
  },  
})
```



Test dans Vue : Pourquoi tester les applications Vue.js

History Mode Activation :

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

const router = createRouter({
  history: createWebHistory(),
  routes: [
    //...
  ],
})
```



Test dans Vue : Les différents types de test

History Mode Activation :

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

const router = createRouter({
  history: createWebHistory(),
  routes: [
    //...
  ],
})
```



Test dans Vue : Tester uniquement avec Vue.js

History Mode Activation :

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

const router = createRouter({
  history: createWebHistory(),
  routes: [
    //...
  ],
})
```



Test dans Vue : Les frameworks à disposition (Jest...)

History Mode Activation :

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

const router = createRouter({
  history: createWebHistory(),
  routes: [
    //...
  ],
})
```



Performance : Le code splitting

—

History Mode Activation :

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

const router = createRouter({
  history: createWebHistory(),
  routes: [
    //...
  ],
})
```



Performance : Pourquoi le lazy loading est important dans Vue.js

History Mode Activation :

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

const router = createRouter({
  history: createWebHistory(),
  routes: [
    //...
  ],
})
```



Performance : Le chargement à la demande

History Mode Activation :

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

const router = createRouter({
  history: createWebHistory(),
  routes: [
    //...
  ],
})
```



Performance : Chargement lazy des routes avec "dynamic import"

History Mode Activation :

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

const router = createRouter({
  history: createWebHistory(),
  routes: [
    //...
  ],
})
```



Performance : Les directives v-once et v-memo

History Mode Activation :

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

const router = createRouter({
  history: createWebHistory(),
  routes: [
    //...
  ],
})
```



Pinia et l'authentification

History Mode Activation :

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

const router = createRouter({
  history: createWebHistory(),
  routes: [
    //...
  ],
})
```



Internationaliser sa webapp

—

History Mode Activation :

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

const router = createRouter({
  history: createWebHistory(),
  routes: [
    //...
  ],
})
```



Les bases - utilisations de fichiers statiques

History Mode Activation :

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

const router = createRouter({
  history: createWebHistory(),
  routes: [
    //...
  ],
})
```



Comment rendre les langues dynamiques ?

History Mode Activation :

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

const router = createRouter({
  history: createWebHistory(),
  routes: [
    //...
  ],
})
```



Comment rendre les langues dynamiques ?

History Mode Activation :

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

const router = createRouter({
  history: createWebHistory(),
  routes: [
    //...
  ],
})
```



Introduction à Vue Query

Vue Query : une bibliothèque pour gérer les requêtes de données dans Vue.js

```
// main.js
import { createApp } from 'vue'
import { createVueQuery } from 'vue-query'
import App from './App.vue'

const app = createApp(App)
const vueQuery = createVueQuery()

app.use(vueQuery)
app.mount('#app')
```



Utilisation de Vue Query pour effectuer une requête GET

Vue Query : une bibliothèque pour gérer les requêtes de données dans Vue.js

- Créer un composant simple pour afficher les données
- Utiliser le hook useQuery pour effectuer une requête GET vers une API
- Gérer les différents états de la requête (loading, error, success)
- Afficher les données dans le composant



Utilisation de Vue Query pour effectuer une requête GET

```
<template>
  <div>
    <div v-if="isLoading">Chargement...</div>
    <div v-else-if="isError">Erreur lors de la récupération des données</div>
    <div v-else>
      <ul>
        <li v-for="user in data" :key="user.id">{{ user.name }}</li>
      </ul>
    </div>
  </div>
</template>

<script>
import { useQuery } from 'vue-query'

export default {
  name: 'UserList',
  setup() {
    const { isLoading, isError, data } = useQuery('users', () => {
      return fetch('https://api.example.com/users').then((response) =>
        response.json()
      )
    })

    return { isLoading, isError, data }
  }
}
</script>
```



Gestion des erreurs de requête

```
<template>
  <div>
    <div v-if="isError">Erreur lors de la récupération des données: {{ error }}
    <div v-else>
      <ul>
        <li v-for="user in data" :key="user.id">{{ user.name }}</li>
      </ul>
    </div>
  </div>
</template>

<script>
import { useQuery } from 'vue-query'

export default {
  name: 'UserList',
  setup() {
    const { isError, error, data } = useQuery('users', () => {
      return fetch('https://api.example.com/users').then((response) => {
        if (!response.ok) {
          throw new Error('Échec de la requête')
        }
        return response.json()
      })
    })

    return { isError, error, data }
  },
}
</script>
```



Ajout de mutations pour le CRUD

```
<template>
  <div>
    <button @click="createUser">Créer un utilisateur</button>
  </div>
</template>

<script>
import { useMutation } from 'vue-query'

export default {
  name: 'UserForm',
  setup() {
    const createUserMutation = useMutation((userData) => {
      return fetch('https://api.example.com/users', {
        method: 'POST',
        body: JSON.stringify(userData),
        headers: {
          'Content-Type': 'application/json',
        },
      }).then((response) => response.json())
    })

    const createUser = async () => {
      const newUser = { name: 'John Doe', email: 'john@example.com' }
      try {
        await createUserMutation.mutateAsync(newUser)
        console.log('Utilisateur créé avec succès')
      } catch (error) {
        console.error('Erreur lors de la création de l\'utilisateur', error)
      }
    }

    return { createUser }
  },
}
</script>
```



Mise en cache et invalidation de cache

```
<template>
  <div>
    <button @click="invalidateCache">Invalider le cache</button>
    <ul>
      <li v-for="user in data" :key="user.id">{{ user.name }}</li>
    </ul>
  </div>
</template>

<script>
import { useQuery, useQueryClient } from 'vue-query'

export default {
  name: 'UserList',
  setup() {
    const queryClient = useQueryClient()

    const invalidateCache = () => {
      queryClient.invalidateQueries('users')
    }

    const { data } = useQuery('users', () => {
      return fetch('https://api.example.com/users').then((response) =>
        response.json()
      )
    })

    return { invalidateCache, data }
  },
}
</script>
```



Personnalisation des requêtes avec des intercepteurs

```
// main.js
import { createApp } from 'vue'
import { createVueQuery } from 'vue-query'
import App from './App.vue'

const app = createApp(App)
const vueQuery = createVueQuery({
  base: 'https://api.example.com',
  interceptors: {
    request: [
      (config) => {
        // Ajouter des en-têtes personnalisées
        config.headers['Authorization'] = 'Bearer token'
        return config
      },
    ],
  },
})
app.use(vueQuery)
app.mount('#app')
```



Utilisation des événements de mutation

```
<template>
  <div>
    <button @click="deletePost">Supprimer le post</button>
  </div>
</template>

<script>
import { useMutation, useQueryClient } from 'vue-query'

export default {
  name: 'PostDetail',
  setup() {
    const queryClient = useQueryClient()

    const deletePostMutation = useMutation((postId) => {
      return fetch(`https://api.example.com/posts/${postId}`, {
        method: 'DELETE',
      }).then((response) => response.json())
    })

    const deletePost = async () => {
      const postId = '123'
      try {
        await deletePostMutation.mutateAsync(postId)
        console.log('Post supprimé avec succès')
        queryClient.invalidateQueries('posts')
      } catch (error) {
        console.error('Erreur lors de la suppression du post', error)
      }
    }

    return { deletePost }
  },
}
</script>
```



Mise en place de rafraîchissements automatiques des données

```
<template>
  <div>
    <ul>
      <li v-for="post in data" :key="post.id">{{ post.title }}</li>
    </ul>
  </div>
</template>

<script>
import { useQuery } from 'vue-query'

export default {
  name: 'PostList',
  setup() {
    const { data } = useQuery('posts', () => {
      return fetch('https://api.example.com/posts').then((response) =>
        response.json()
      )
    }, {
      refetchInterval: 5000 // Rafraîchir toutes les 5 secondes
    })

    return { data }
  },
}
</script>
```



Utilisation de la Composition API avec Vue Query et Pinia

```
<template>
  <div>
    <ul>
      <li v-for="user in users" :key="user.id">{{ user.name }}</li>
    </ul>
  </div>
</template>

<script>
import { useUserStore } from '@/store'
import { useQuery } from 'vue-query'
import { computed } from 'vue'

export default {
  name: 'UserList',
  setup() {
    const userStore = useUserStore()
    const { data: users } = useQuery('users', () => {
      return fetch('https://api.example.com/users').then((response) =>
        response.json()
      )
    })

    return { users: computed(() => userStore.users), users }
  }
}
</script>
```



VUE 3 ADVANCED

Pierrick Hauguel



v-once

Rend l'élément et le composant une seule fois, et ignore les mises à jour futures.

```
<!-- élément simple -->

```

template



v-memo

Mémorise une sous-arborescence du template. Peut être utilisée à la fois sur les éléments et les composants

```
<div v-memo="[valueA, valueB]">  
  ...  
</div>
```

template

```
<div v-for="item in list" :key="item.id" v-memo="[item.id === selected]">  
  <p>ID: {{ item.id }} - selected: {{ item.id === selected }}</p>  
  <p>...more child nodes</p>  
</div>
```

template



Introduction aux Tests Unitaires

- Définition: Les tests unitaires permettent de vérifier le bon fonctionnement des différentes unités de code (fonctions, méthodes, classes, modules, etc.) de manière isolée.
- Importance:
 - Assurer que le code fonctionne comme prévu.
 - Faciliter la maintenance et l'évolution du code.
 - Déetecter et corriger les bugs tôt dans le cycle de développement.



Introduction à Vitest

- Qu'est-ce que Vitest?

Un framework de tests unitaires conçu spécifiquement pour Vue.js.

Créé et maintenu par les membres de l'équipe Vue/Vite.
Recommandé officiellement pour les projets Vue.js utilisant Vite



Exemple

Fichier de Fonction (sum.js):

```
// sum.js
export function sum(a, b) {
    return a + b;
}
```

Fichier de Test (sum.test.js):

```
// sum.test.js
import { expect, test } from 'vitest';
import { sum } from './sum';

test('adds 1 + 2 to equal 3', () => {
    expect(sum(1, 2)).toBe(3);
});
```

Exécution du Test:

```
npm run test
```



Structuration des Tests

- Bloc describe: Permet de regrouper plusieurs tests relatifs à une même fonctionnalité ou composant.
- Bloc test: Définit un test unitaire individuel.

```
describe('Sum Function', () => {
  test('adds 1 + 2 to equal 3', () => {
    expect(sum(1, 2)).toBe(3);
  });
  test('adds 3 + 4 to equal 7', () => {
    expect(sum(3, 4)).toBe(7);
  });
});
```



Utilisation de expect pour les Assertions

expect: Permet de vérifier le résultat d'une opération.

```
test('adds 1 + 2 to equal 3', () => {
  expect(sum(1, 2)).toBe(3);
});

test('returned value not be greater than 3', () => {
  expect(sum(1, 2)).not.toBeGreaterThan(3);
});
```



Gestion du Code Asynchrone

Tests asynchrones: Utilisation de `async` et `await` pour gérer les opérations asynchrones.

```
test('async test example', async () => {
  const data = await fetchData();
  expect(data).toBe('hello');
});
```



Groupement des Tests

Nesting describe Blocks: Organiser les tests en sous-groupes pour une meilleure structuration et lisibilité.

```
describe('Arithmetic Functions', () => {
  describe('Sum Function', () => {
    test('adds 1 + 2 to equal 3', () => {
      expect(sum(1, 2)).toBe(3);
    });
  });
  describe('Subtract Function', () => {
    test('subtracts 2 - 1 to equal 1', () => {
      expect(subtract(2, 1)).toBe(1);
    });
  });
});
```



Tests avec des données variées:

```
describe.each([
  { a: 1, b: 1, expected: 2 },
  { a: 1, b: 2, expected: 3 },
  { a: 2, b: 1, expected: 3 },
])('addition de $a + $b', ({ a, b, expected }) => {
  test(`retourne ${expected}`, () => {
    expect(a + b).toBe(expected);
  });
});
```



Tests de Composables

```
// composables/useCounter.js
export default function useCounter() {
  const count = ref(0);

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

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



Tests de Composables

```
// tests/composables/useCounter.spec.js
import { ref } from 'vue';
import useCounter from '@/composables/useCounter';

describe('useCounter Composable', () => {
  test('should increment count', () => {
    const { count, increment } = useCounter();
    expect(count.value).toBe(0);
    increment();
    expect(count.value).toBe(1);
  });
});
```



Tests de Composants Setup

```
<!-- components/Counter.vue -->
<template>
  <button @click="increment">{{ count }}</button>
</template>

<script>
import useCounter from '@/composables/useCounter';

export default {
  setup() {
    const { count, increment } = useCounter();
    return {
      count,
      increment,
    };
  },
};
</script>
```



Tests de Composants Setup

```
// tests/components/Counter.spec.js
import { mount } from '@vue/test-utils';
import Counter from '@/components/Counter.vue';

describe('Counter Component', () => {
  test('should increment count on button click', async () => {
    const wrapper = mount(Counter);
    expect(wrapper.html()).toContain('0');
    await wrapper.find('button').trigger('click');
    expect(wrapper.html()).toContain('1');
  });
});
```



Introduction à TypeScript



TypeScript comprend les types primitifs de JavaScript : number, string, boolean.

- Vous pouvez annoter les variables avec des types explicites.
- Exemple : `let age: number = 30;`
- TypeScript peut déduire le type d'une variable si vous ne l'avez pas annotée explicitement.
- Exemple : `let score = 100;` (ici, score est de type number).



Introduction à Typescript

- TypeScript prend en charge des opérations spécifiques aux types.
- Exemple : let x: number = 10; let y: number = 5; let somme: number = x + y;

```
function addition(a: number, b: number): number {  
    return a + b;  
}  
  
const resultat: number = addition(10, 5);
```



Introduction à Typescript

- Les tableaux sont des collections de valeurs de même type.
- Les tuples sont des tableaux avec une taille fixe et des types spécifiques pour chaque élément.

```
let nombres: number[] = [1, 2, 3, 4, 5];
```

```
let coordonnees: [number, number] = [3, 7];
```



Introduction à Typescript

- Les énumérations sont des types qui définissent un ensemble nommé de constantes.
- Exemple :

```
enum Couleur {  
    Rouge,  
    Vert,  
    Bleu  
}
```

```
function afficherCouleur(couleur: Couleur): void {  
    console.log("La couleur sélectionnée est " + Couleur[couleur]);  
}  
  
afficherCouleur(Couleur.Verte);
```



Introduction à Typescript

- Vous pouvez annoter les paramètres et la valeur de retour des fonctions.
- Exemple :

```
function addition(a: number, b: number): number {  
    return a + b;  
}
```

```
function saluer(nom: string, age?: number): string {  
    return `Bonjour, ${nom}${age ? `, tu as ${age} ans` : ""} !`;  
}
```

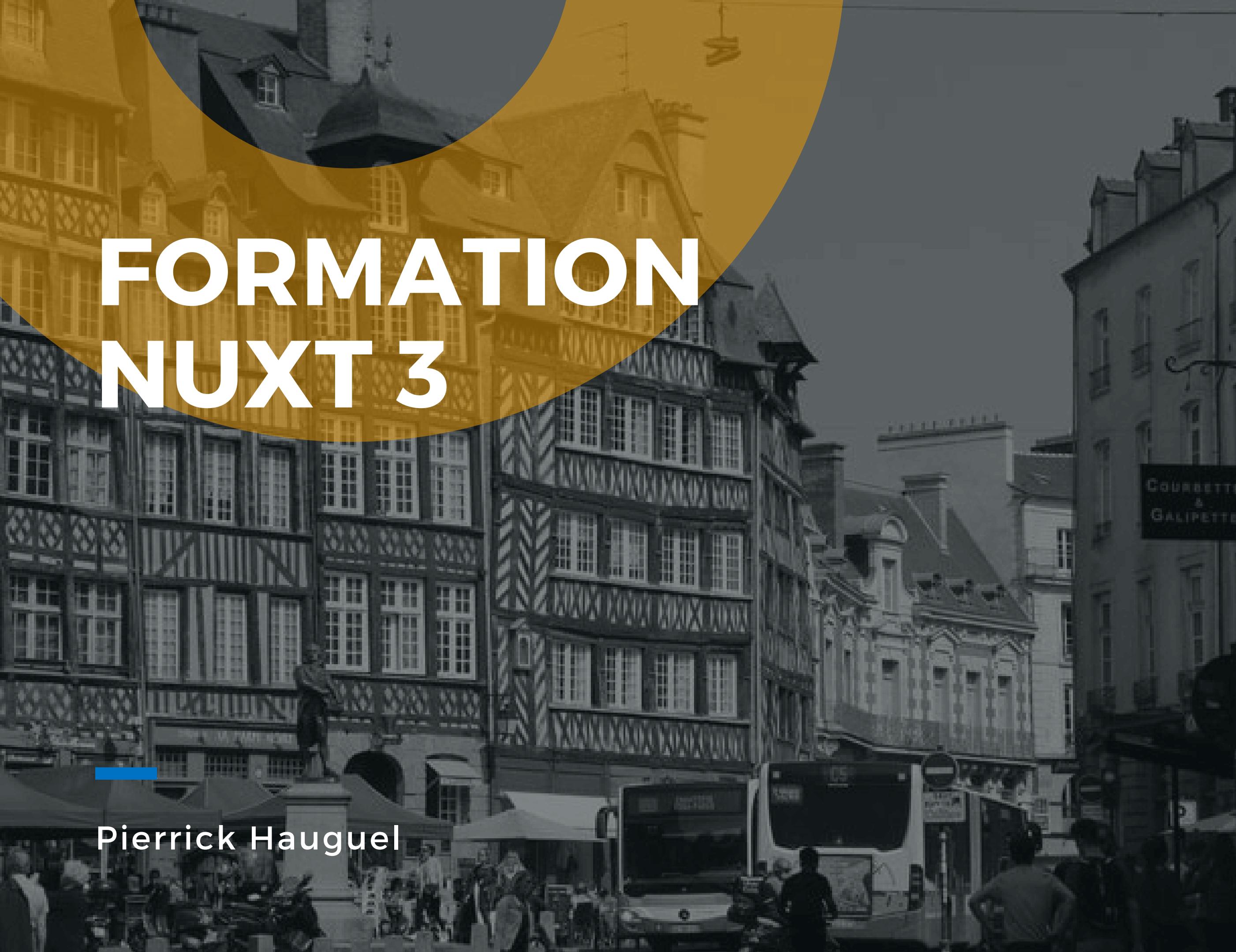


Introduction à TypeScript

- Les fonctions fléchées offrent une syntaxe concise.
- Exemple :

```
const multiplier = (a: number, b: number) => a * b;
```

```
function afficherMessage(message: string): void {  
    console.log(message);  
}  
  
afficherMessage("Bonjour, TypeScript !");
```



FORMATION NUXT3

Pierrick Hauguel



Nuxt 3

Vue d'ensemble de Nuxt 3

- Framework basé sur Vue.js pour le développement d'applications universelles (SSR, SPA, Static).
- Intégration complète avec Vue 3 et la Composition API.
- Performances améliorées et optimisations automatiques.



Comparaison avec les Versions Précédentes

Vue d'ensemble de Nuxt 3

- Performance et réactivité améliorées: Vue 3 apporte une réactivité plus fine et une performance accrue grâce à son nouveau système de réactivité basé sur les proxies.
- Composition API: Cette nouvelle API offre plus de flexibilité et de réutilisabilité dans la composition des composants.
- Meilleure intégration TypeScript: Vue 3 a été conçu avec un support TypeScript natif, permettant une expérience de développement plus robuste et typée.



Comparaison avec les Versions Précédentes

```
// Vue 2: Options API
export default {
  data() {
    return {
      message: 'Bonjour Nuxt 2!'
    }
  },
  methods: {
    greet() {
      console.log(this.message);
    }
  }
}
```

```
// Vue 3: Composition API dans Nuxt 3
import { ref } from 'vue';

export default {
  setup() {
    const message = ref('Bonjour Nuxt 3!');
    const greet = () => {
      console.log(message.value);
    };

    return { message, greet };
  }
}
```



Introduction de la Composition API

Avantages de la Composition API dans Nuxt
3

- Organisation du code: Regroupez la logique relative à une même fonctionnalité, rendant le code plus lisible et maintenable.
- Réutilisation du code: Créez des fonctions composable pour partager la logique entre les composants.
- Flexibilité accrue: Construisez des composants complexes avec moins de contraintes et plus de clarté.



Exemple de code

```
import { ref, computed } from 'vue';

export default {
  setup() {
    const count = ref(0);
    const doubleCount = computed(() => count.value * 2);

    const increment = () => {
      count.value++;
    };

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



Améliorations en termes de vitesse et d'optimisation

Optimisations de Performance dans Nuxt 3

- Vitesse de démarrage: Temps de démarrage réduit grâce à une meilleure optimisation du code et des dépendances.
- Optimisation du bundle: Réduction de la taille des bundles avec l'analyse statique et le tree-shaking.
- Lazy loading et code splitting: Chargement paresseux des composants et séparation du code pour une meilleure performance.



Exemple de code

```
// Lazy loading d'un composant dans Nuxt 3

<template>
  <Suspense>
    <AsyncComponent />
  </Suspense>
</template>

<script>
export default {
  components: {
    AsyncComponent: () => import '~/components/MyAsyncComponent.vue'
  }
}
</script>
```



Installation de Nuxt 3

- Prérequis: Node.js (version 14 ou supérieure).
- Utilisation de npm ou yarn: Commandes pour installer Nuxt 3.

```
npx nuxi@latest init nuxtaccessit
```



Configuration de l'Environnement de Développement

- IDE recommandé: Visual Studio Code avec extensions (Vetur, ESLint, Prettier).
- Configuration de l'éditeur: Paramètres pour linter et formater le code.

```
// settings.json dans VSCode
{
  "editor.codeActionsOnSave": {
    "source.fixAll.eslint": true
  },
  "vetur.validation.template": true,
  "eslint.validate": ["vue", "javascript"]
}
```



Structure de Base du Projet Nuxt 3

Répertoires et fichiers principaux:

- assets pour les ressources statiques.
- components pour les composants Vue.
- layouts pour les mises en page.
- pages pour les vues et routes.
- nuxt.config.js pour la configuration globale.

```
mon-projet-nuxt3/
  ├── assets/
  ├── components/
  ├── layouts/
  ├── pages/
  └── nuxt.config.js
```



Configuration de Nuxt 3

- Centralisation de la configuration: Le fichier `nuxt.config.js` est le cœur de la configuration de votre projet Nuxt 3.
- Paramètres configurables: Importation de modules, définition des plugins, configuration du rendu, personnalisation du build, etc.



Exemple de code

```
// Exemple basique de nuxt.config.js
export default {
  head: {
    title: 'Mon Projet Nuxt 3',
    meta: [
      { charset: 'utf-8' },
      { name: 'viewport', content: 'width=device-width, initial-scale=1' }
    ],
    modules: ['@nuxtjs/axios'],
    plugins: ['~/plugins/my-plugin.js']
}
```



Fichier nuxt.config.js

- Personnalisation du Head: Configurer les balises meta, les scripts, et les liens CSS globaux.
- Modules et Plugins: Ajouter des fonctionnalités et des comportements personnalisés à votre application Nuxt.
- Configuration du rendu: Gérer les paramètres de rendu côté serveur et client.



Exemple de code

```
// Configuration avancée dans nuxt.config.js
export default {
  head: {
    // Personnalisation du head
  },
  modules: [
    // Modules Nuxt
  ],
  plugins: [
    // Plugins Nuxt
  ],
  build: {
    // Options de build
  },
  server: {
    // Configuration du serveur
  }
}
```



Configuration du Serveur

- Port et Hôte: Définir le port et l'hôte pour le serveur de développement local.
- Middleware: Ajouter des middleware personnalisés pour le traitement des requêtes.

```
// Configurer le serveur dans nuxt.config.js
export default {
  server: {
    port: 3000, // par défaut : 3000
    host: '0.0.0.0' // par défaut : localhost
  }
}
```



Personnalisation de l'Environnement

- Variables d'environnement: Utiliser .env pour gérer les variables d'environnement.
- Configuration en fonction de l'environnement: Adapter la configuration selon l'environnement de développement, de test, ou de production.

```
// Utilisation des variables d'environnement dans nuxt.config.js
export default {
  env: {
    baseUrl: process.env.BASE_URL || 'http://localhost:3000'
  }
}
```



Création de Pages et Layouts

- Pages: Créez des fichiers .vue dans le dossier pages pour définir les vues de votre application.
- Layouts: Définissez des mises en page globales dans le dossier layouts.
- Utilisation de <slot/> Pour injecter les pages dans les layouts.



Utilisation des Templates et Composants

- Templates: Utilisez le langage de template de Vue pour créer vos vues.
- Composants: Réutilisez des composants Vue dans vos pages et layouts.
- Importation automatique: Nuxt 3 importe automatiquement vos composants.

```
<!-- pages/index.vue -->
<template>
  <div>
    <MonComposant />
    <p>Bienvenue sur ma page Nuxt!</p>
  </div>
</template>
```



Navigation dans Nuxt 3

Utilisation de `<NuxtLink>`: Utilisez `<NuxtLink>` pour la navigation entre les pages, ce qui améliore la performance grâce au prefetching.

```
<!-- Lien vers une autre page -->
<NuxtLink to="/about">À propos</NuxtLink>
```



Utilisation des Paramètres de Route avec useRoute

- Accès aux détails de la route: Utilisez le composable useRoute dans un bloc `<script setup>` ou une méthode `setup()` pour accéder aux détails de la route actuelle.
- Extraction des paramètres: Accédez facilement aux paramètres de la route, comme l'ID dans un chemin dynamique.

```
<script setup lang="ts">
  const route = useRoute();
  console.log(route.params.id); // Affiche l'ID de la route
</script>
```



Middleware de Route dans Nuxt 3

- Trois types de middleware: Anonyme, nommé, et global.
- Exécution dans la partie Vue de Nuxt: Différent des middleware serveur, ils sont exécutés côté client.
- Cas d'utilisation: Par exemple, pour protéger une page d'accès non autorisé.

```
// middleware/auth.ts
export default defineNuxtRouteMiddleware((to, from) => {
  if (isAuthenticated() === false) {
    return navigateTo('/login');
  }
});
```



Validation des Routes dans Nuxt 3

- Fonction validate: Utilisez validate dans definePageMeta pour déterminer si une route doit être rendue.
- Gestion des erreurs: Retournez false ou un objet avec statusCode/statusMessage pour gérer les erreurs de route.

```
<script setup lang="ts">
definePageMeta({
  validate: async (route) => {
    return /^\d+$/.test(route.params.id); // Vérifie si l'ID est numérique
  }
});
</script>
```



Dynamic Routing

- Crédit de routes: Les fichiers dans pages deviennent des routes.
- Routes dynamiques: Utilisez les fichiers avec des crochets (ex: [id].vue) pour les routes dynamiques.

```
<!-- pages/posts/[id].vue -->
<template>
  <div>
    <h1>Mon Post</h1>
  </div>
</template>
```



Nested Routes

- Structure des dossiers: Utilisez des sous-dossiers dans pages pour créer des routes imbriquées.
- Utilisation de <NuxtChild>: Pour afficher les composants enfants dans une route parente.

```
<!-- pages/blog/index.vue -->  
<template>  
  <div>  
    <h1>Blog</h1>  
    <NuxtChild/>  
  </div>  
</template>
```



Middleware de Routing

- Définition de middleware: Créez des fonctions middleware pour exécuter du code avant de rendre une page ou un groupe de pages.
- Application globale ou locale: Appliquez le middleware globalement dans nuxt.config.js ou localement dans les composants/pages.

```
// middleware/check-auth.js
export default function ({ store, redirect }) {
  if (!store.state.authenticated) {
    return redirect('/login');
  }
}

// nuxt.config.js ou dans un composant/page spécifique
export default {
  middleware: 'check-auth'
}
```



Créer des Transitions Entre les Pages dans Nuxt 3

- Utilisation de <Nuxt>: Le composant <Nuxt/> dans vos layouts permet de définir des transitions pour les changements de page.
- CSS pour transitions: Définissez des transitions CSS globales ou spécifiques à une page.

```
/* assets/main.css */  
.  
.page-transition-enter-active, .page-transition-leave-active {  
    transition: opacity 0.5s;  
}  
  
.page-transition-enter, .page-transition-leave-to {  
    opacity: 0;          <!-- layouts/default.vue -->  
}  
  
<template>  
    <div>  
        <Nuxt class="page-transition"/>  
    </div>  
</template>
```



Gestion des Assets dans Nuxt 3

- Répertoire assets: Stockez vos ressources statiques telles que les feuilles de style, les images et les polices.
- Importation des ressources: Utilisez le système de modules de Nuxt pour intégrer des ressources dans vos composants.

```
<!-- Exemple d'intégration d'une image -->
<template>
  <div>
    
  </div>
</template>
```



Optimisation des Images et des Fichiers

- Optimisation automatique des images:
Utilisation de @nuxt/image pour une optimisation automatique des images.
- Lazy loading: Chargez les images à la demande pour améliorer les performances.

```
<!-- Utilisation de @nuxt/image pour une image optimisée -->
<nuxt-img src="mon-image.jpg" alt="Image Optimisée"/>
```



Intégration des Ressources Statiques et Dynamiques

- Ressources statiques: Accédez directement via le dossier static.
- Ressources dynamiques: Utilisez Webpack pour importer des ressources depuis assets.

```
<!-- Exemple pour une ressource statique -->


<!-- Exemple pour une ressource dynamique -->

```



Animer les Composants dans Nuxt 3

- Utilisation de Vue's transition: Emballez les composants avec la balise `<transition>` pour appliquer des animations.
- Animation à l'entrée et à la sortie: Contrôlez les états de début et de fin des animations avec des classes CSS.

```
<!-- components/MyComponent.vue -->
<template>
  <transition name="fade">
    <div v-if="show">Mon composant animé</div>
  </transition>
</template>

<style>
  .fade-enter-active, .fade-leave-active {
    transition: opacity 0.5s;
  }
  .fade-enter, .fade-leave-to {
    opacity: 0;
  }
</style>
```



Pourquoi Utiliser des Composables Spécifiques pour des appels REST ?

- Réduction de la duplication des appels réseau:
Eviter de répéter les mêmes appels réseau sur le client et le serveur.
- Suspense et useFetch: Gérer efficacement les appels asynchrones et l'état de chargement avec Suspense.
- \$fetch et useAsyncData: Faciliter la récupération de données et la gestion de l'état.
- Options comme lazy et client-only: Contrôler quand et comment les données sont récupérées.



Optimisation et Gestion de la Charge Utile

- Minimiser la taille de la charge utile: Charger uniquement les données nécessaires pour améliorer les performances.
- Caching et re-fetching: Utiliser des stratégies de mise en cache pour optimiser les appels réseau.
- Fetching not immediate: Retarder le fetching jusqu'à ce qu'il soit nécessaire.

```
const { data } = useFetch('/api/data', { immediate: false });
```



Gestion des En-têtes et Cookies

- Passer des en-têtes clients à l'API: Transmettre des informations spécifiques au client lors des appels API.
- Passer des cookies du serveur: Gérer les cookies pour les authentifications ou préférences utilisateurs.

```
const { data } = useFetch('/api/data', {  
  headers: { 'Authorization': 'Bearer token' },  
  credentials: 'include' // pour inclure les cookies  
});
```



Utilisation de \$fetch dans Nuxt 3

- Simplicité et polyvalence: \$fetch est une fonction globale pour effectuer des requêtes HTTP, avec une syntaxe simple et intuitive.
- Gestion des erreurs intégrée: Offre une gestion des erreurs et des statuts de réponse HTTP.
- Réactivité: Les données récupérées sont réactives et se mettent à jour automatiquement dans l'interface utilisateur.

```
<script setup>
const data = await $fetch('/api/data');
</script>
```



Utilisation de useAsyncData dans Nuxt 3

- Fetching côté serveur et client: useAsyncData gère le fetching de données à la fois côté serveur et client, idéal pour le SSR.
- Synchronisation avec l'interface utilisateur: Les données sont synchronisées avec l'UI, permettant une mise à jour dynamique des composants.
- Options avancées: Supporte des options comme la mise en cache, le re-fetching et la détermination de la stratégie de fetching.

```
<script setup>
const { data, refresh, pending, error } = useAsyncData(() => $fetch('/api'))
</script>
```



Utilisation de Vue Query dans Nuxt 3

- Vue Query: Une bibliothèque pour gérer les requêtes, le caching, la synchronisation et la mise à jour des données.
- Intégration avec Nuxt: Installez et configurez Vue Query pour l'utiliser dans vos composants Nuxt.

```
# Installation de Vue Query
```

```
npm install @tanstack/vue-query
```



Utilisation de Vue Query dans Nuxt 3

```
// Utilisation de Vue Query dans un composant
import { useQuery } from '@tanstack/vue-query';

export default {
  setup() {
    const { data, isLoading, error } = useQuery(['repoData'], () =>
      fetch('https://api.github.com/repos/tannerlinsley/react-query').then(
        res.json()
      )
    );

    return { data, isLoading, error };
  }
}
```



Utilisation de useState dans Nuxt 3

- Accès direct aux composables: useState et d'autres composables Nuxt sont accessibles directement sans import.
- État partagé réactif et SSR-friendly: Idéal pour partager des données entre composants, tout en étant compatible avec le SSR.
- Facile à utiliser: Simplifie la gestion de l'état partagé dans l'application.

```
<script setup>
const counter = useState('counter', () => 0);

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

</script>
```



Avantages pour l'Application

- Cette approche permet une gestion d'état centralisée et cohérente, utile pour des éléments comme des indicateurs d'état, des compteurs, ou des données utilisateur.
- Parfait pour les applications où plusieurs composants dépendent de la même source de données.



Introduction à Pinia

- Qu'est-ce que Pinia ?: Pinia est la bibliothèque de state management recommandée pour Vue 3, offrant une API intuitive et flexible.
- Avantages de Pinia: Facile à utiliser, typage TypeScript natif, devtools puissants, et intégration parfaite avec la Composition API.

```
// Création d'un store Pinia
import { defineStore } from 'pinia';

export const useMyStore = defineStore('myStore', {
  state: () => ({
    counter: 0
}),
actions: {
  increment() {
    this.counter++;
  }
}
});
```



Configuration de Pinia dans Nuxt 3

- Installation de Pinia: Ajoutez Pinia comme dépendance dans votre projet Nuxt 3.
- Configuration dans Nuxt: Configurez Pinia dans nuxt.config.js ou en créant un plugin.

```
# Installation de Pinia
npm install @pinia/nuxt
```

```
// nuxt.config.js
export default {
  buildModules: [
    '@pinia/nuxt',
  ]
}
```



Concepts de SSR et SSG

- Server-Side Rendering (SSR): Génère le HTML de chaque page côté serveur à chaque requête, ce qui est idéal pour le SEO et la performance initiale.
- Static Site Generation (SSG): Génère des pages HTML statiques lors du build, permettant un hébergement facile et de meilleures performances.

```
# Commandes pour SSR et SSG
nuxt build # pour SSR
nuxt generate # pour SSG
```



Configuration de SSR et SSG

- Configuration dans nuxt.config.js: Définissez le mode de rendu (SSR ou SSG) dans votre configuration.
- Stratégie de déploiement: Choisissez entre SSR et SSG en fonction de vos besoins en matière de SEO, de performance, et d'hébergement.

```
// nuxt.config.js
export default {
  target: 'static', // pour SSG
  // ou
  ssr: true, // pour SSR
}
```



Optimisation et Meilleures Pratiques pour SSR et SSC

- Optimisation des performances: Utilisez le lazy loading, le code splitting et optimisez vos assets.
 - SEO efficace: Assurez-vous que vos meta tags et contenus sont bien configurés pour le SEO.
 - Cache et CDN: Pour les sites SSR, utilisez un cache et un CDN pour améliorer les performances de rendu.
-
- Utilisez nuxt/image pour une optimisation automatique des images.
 - Exploitez les possibilités de caching de votre serveur ou service d'hébergement.



Utilisation du Composant `<ClientOnly>` dans Nuxt 3

Le composant `<ClientOnly>` de Nuxt 3 est utilisé pour rendre les composants exclusivement côté client. Ceci est particulièrement utile pour les composants qui ne doivent être exécutés que dans le navigateur, comme ceux qui dépendent de l'API du navigateur ou des plugins côté client.



Utilisation du Composant `<ClientOnly>` dans Nuxt 3

Fonctionnalités clés:

1. Rendu Client-Seul: Le contenu à l'intérieur de `<ClientOnly>` ne s'affiche que du côté client.
2. Props 'placeholderTag' et 'fallbackTag': Permet de spécifier une balise à rendre côté serveur.
3. Props 'placeholder' et 'fallback': Définit un contenu de remplacement à afficher côté serveur.
4. Slot '#fallback': Permet de spécifier un contenu de remplacement côté serveur avec plus de flexibilité.



Utilisation du Composant <ClientOnly> dans Nuxt 3

Utilisation Basique avec Props:

```
<template>
  <div>
    <Sidebar />
    <ClientOnly fallback-tag="span" fallback="Loading comments...">
      <Comment />
    </ClientOnly>
  </div>
</template>
```



Utilisation du Composant `<ClientOnly>` dans Nuxt 3

Utilisation Avancée avec Slot 'fallback':

```
<template>
  <div>
    <Sidebar />
    <ClientOnly>
      <!-- Contenu client -->
      <template #fallback>
        <!-- Contenu côté serveur -->
        <p>Loading comments...</p>
      </template>
    </ClientOnly>
  </div>
</template>
```



Le Composant <NuxtPage> dans Nuxt 3

Le composant <NuxtPage> est essentiel pour afficher les pages situées dans le répertoire `pages/`. Il s'agit d'un composant intégré dans Nuxt, agissant comme un enveloppeur autour du composant <RouterView> de Vue Router. <NuxtPage> facilite l'affichage des pages de niveau supérieur ou imbriquées, avec une gestion automatique des noms et des routes.



Le Composant <NuxtPage> dans Nuxt 3

1. Props:

- name: Indique à RouterView de rendre le composant correspondant dans l'option components de l'itinéraire.
- route: Localisation de l'itinéraire avec tous ses composants résolus.
- pageKey: Contrôle le re-rendu du composant <NuxtPage>.
- transition: Définit les transitions globales pour toutes les pages rendues avec <NuxtPage>.
- keepalive: Gère la préservation de l'état des pages.



Le Composant <NuxtPage> dans Nuxt 3

- Ref de Page: Accès au ref d'un composant de page via `ref.value.pageRef`.
- Props Personnalisées: <NuxtPage> accepte également des props personnalisés pour des besoins spécifiques.



Le Composant <NuxtPage> dans Nuxt 3

Utilisation Basique avec Clé Statique:

```
<template>
  <NuxtPage page-key="static" />
</template>
```

Utilisation Basique avec Clé Statique:

```
<NuxtPage :page-key="route => route fullPath" />
```

Utilisation Basique avec Clé Statique:

```
<script setup lang="ts">
definePageMeta({
  key: route => route fullPath
})
</script>
```



Le Composant <NuxtPage> dans Nuxt 3

Accès au Ref de Page:

```
<script setup lang="ts">  
const page = ref()  
  
function logFoo() {  
  page.value.pageRef.foo()  
}  
</script>  
  
<template>  
  <NuxtPage ref="page" />  
</template>
```

Utilisation de Props Personnalisées:

```
<NuxtPage :foobar="123" />
```



Utilisation du Composant `<NuxtLayout>` dans Nuxt 3

Le composant `<NuxtLayout>` est utilisé pour définir des mises en page sur les pages et les pages d'erreur dans Nuxt 3. Il permet de spécifier une mise en page pour envelopper le contenu de la page, offrant une grande flexibilité dans la personnalisation de l'interface utilisateur.



Utilisation du Composant `<NuxtLayout>` dans Nuxt 3

1. Définition de Layouts: Spécifiez un nom de layout qui correspond à un fichier dans le répertoire `layouts/`.
2. Props Dynamiques: Possibilité d'utiliser une référence réactive ou une propriété calculée pour le nom du layout.
3. Utilisation de Props Personnalisées: `<NuxtLayout>` accepte des props supplémentaires qui sont accessibles en tant qu'attributs dans le layout.
4. Transitions: `<NuxtLayout>` gère les transitions entre les layouts grâce au composant `<Transition>` de Vue.
5. Ref de Layout: Accès au ref d'un composant de layout via `ref.value.layoutRef`.



Utilisation du Composant <NuxtLayout> dans Nuxt 3

Utilisation Basique dans app.vue:

```
<template>
  <NuxtLayout>
    Contenu de la page
  </NuxtLayout>
</template>
```



Utilisation du Composant <NuxtLayout> dans Nuxt 3

Spécification d'un Layout Personnalisé:

```
<script setup lang="ts">
  const layout = 'custom' // Correspond à layouts/custom.vue
</script>

<template>
  <NuxtLayout :name="layout">
    <NuxtPage />
  </NuxtLayout>
</template>
```



Utilisation du Composant <NuxtLayout> dans Nuxt 3

Passage de Props Personnalisées:

```
<template>
  <div>
    <NuxtLayout name="custom" title="Je suis un layout personnalisé">
      <!-- Contenu ici -->
    </NuxtLayout>
  </div>
</template>
```

```
<script setup lang="ts">
  const layoutCustomProps = useAttrs()
  console.log(layoutCustomProps.title) // Je suis un layout personnalisé
</script>
```



Le Composant `<NuxtLoadingIndicator>`

Le composant `<NuxtLoadingIndicator>` est utilisé pour afficher une barre de progression entre les navigations de page dans Nuxt 3. Ce composant est idéal pour améliorer l'expérience utilisateur en fournissant un retour visuel pendant le chargement des pages.



Le Composant `<NuxtLoadingIndicator>`

1. Affichage de la Barre de Progression: Visible lors des changements de page pour indiquer le chargement.
2. Personnalisation Facile: Possibilité de personnaliser la couleur, la hauteur, et la durée d'affichage.
3. Slots: Permet de passer du HTML personnalisé ou des composants à travers le slot par défaut de l'indicateur de chargement.



Le Composant `<NuxtLoadingIndicator>`

Utilisation Basique dans app.vue:

```
<template>
  <NuxtLoadingIndicator />
<NuxtLayout>
  <NuxtPage />
</NuxtLayout>
</template>
```



Le Composant `<NuxtLoadingIndicator>`

Personnalisation de l'Indicateur:

```
<template>
  <NuxtLoadingIndicator color="#ff0000" height="4" duration="1500">
    <NuxtLayout>
      <NuxtPage />
    </NuxtLayout>
  </NuxtLoadingIndicator>
</template>
```



Gestion des Erreurs avec `<NuxtErrorBoundary>`

Le composant `<NuxtErrorBoundary>` est conçu pour gérer les erreurs côté client qui surviennent dans son contenu par défaut (slot). Utilisant le crochet `onErrorCaptured` de Vue sous le capot, ce composant est essentiel pour une gestion robuste des erreurs dans une application Nuxt 3.



Gestion des Erreurs avec `<NuxtErrorBoundary>`

Fonctionnalités Principales:

1. Capture d'Erreurs: Gère les erreurs survenant dans les composants enfants.
2. Event `@error`: Événement émis lorsqu'une erreur se produit dans le slot par défaut.
3. Slot `#error`: Permet de spécifier un contenu de repli en cas d'erreur.



Gestion des Erreurs avec <NuxtErrorBoundary>

Utilisation Basique avec Event @error:

```
<template>
  <NuxtErrorBoundary @error="logSomeError">
    <!-- Contenu susceptible de générer une erreur --&gt;
  &lt;/NuxtErrorBoundary&gt;
&lt;/template&gt;</pre>
```



Gestion des Erreurs avec <NuxtErrorBoundary>

Utilisation du Slot #error pour Afficher un Message d'Erreur:

```
<template>
  <NuxtErrorBoundary>
    <!-- Contenu principal -->
    <template #error="{ error }">
      <p>Une erreur s'est produite : {{ error }}</p>
    </template>
  </NuxtErrorBoundary>
</template>
```



Optimisation SEO avec useSeoMeta

Le composable `useSeoMeta` permet de définir les balises méta SEO de votre site sous forme d'objet plat avec un support complet de TypeScript. Cela aide à éviter les erreurs courantes, comme la confusion entre `name` et `property`, et les fautes de frappe, avec plus de 100+ balises méta entièrement typées.



Optimisation SEO avec useSeoMeta

Fonctionnalités Principales:

1. Définition Simple et Sécurisée: Définissez facilement les balises méta SEO de votre site de manière sûre et structurée.
2. Support TypeScript Complet: Profitez de l'auto-complétion et de la vérification de types pour éviter les erreurs.
3. Balises Réactives: Utilisez la syntaxe de getter calculé pour les balises dynamiques.



Optimisation SEO avec useSeoMeta

Utilisation Basique dans app.vue:

```
<script setup lang="ts">
useSeoMeta({
  title: 'Mon Site Incroyable',
  ogTitle: 'Mon Site Incroyable',
  description: 'Voici mon site incroyable, laissez-moi vous en parler.',
  ogDescription: 'Voici mon site incroyable, laissez-moi vous en parler.',
  ogImage: 'https://exemple.com/image.png',
  twitterCard: 'summary_large_image',
})
</script>
```



Optimisation SEO avec useSeoMeta

Utilisation avec Balises Réactives:

```
<script setup lang="ts">  
const title = ref('Mon titre')  
  
useSeoMeta({  
  title,  
  description: () => `description: ${title.value}`  
})  
</script>
```



Gestion des Cookies avec useCookie

useCookie est un composable compatible avec le rendu côté serveur (SSR) pour lire et écrire des cookies. Il permet de gérer les cookies de manière simple et efficace dans les pages, composants et plugins d'une application Nuxt 3.

Fonctionnalités Principales:

1. SSR-Friendly: Fonctionne aussi bien en rendu côté serveur que côté client.
2. Sérialisation Automatique: Les valeurs des cookies sont automatiquement sérialisées et déserialisées en JSON.
3. Facilité d'Utilisation: Permet de lire et d'écrire des cookies avec une syntaxe simple et directe.



Personnalisation des Métadonnées de Page avec `useHead`

`useHead` est un composable qui permet de personnaliser les propriétés de la balise `<head>` de pages individuelles dans une application Nuxt. Il offre une gestion programmatique et réactive des balises de tête, alimentée par `Unhead`. Pour les données provenant d'utilisateurs ou d'autres sources non fiables, il est recommandé d'utiliser `useHeadSafe`.



Personnalisation des Métadonnées de Page avec useHead

Fonctionnalités Principales:

1. Gestion Réactive: Permet de gérer les balises de tête de manière dynamique.
2. Personnalisation Complète: Contrôle des titres, des liens, des méta-tags, et plus encore.
3. Types pour useHead: Support complet pour divers types de contenu de la balise <head>.



Personnalisation des Métadonnées de Page avec useHead

Utilisation Basique de useHead:

```
<script setup lang="ts">  
useHead({  
    title: 'Titre de ma page',  
    titleTemplate: titre => `MonSite.com - ${titre}`,  
    meta: [  
        { name: 'description', content: 'Description de ma page' },  
        { property: 'og:title', content: 'Titre pour OpenGraph' }  
    ],  
    link: [{ rel: 'icon', href: '/favicon.ico' }]  
})  
</script>
```



Gestion des Interruptions de Navigation avec `abortNavigation`

`abortNavigation` est une fonction utilitaire utilisée pour empêcher la navigation de se produire dans une application Nuxt. Elle est principalement utilisée dans les gestionnaires de middleware de route et peut déclencher une erreur si nécessaire. Cette fonction est utile pour contrôler l'accès aux routes en fonction de certaines conditions, comme l'autorisation de l'utilisateur.



Gestion des Interruptions de Navigation avec `abortNavigation`

Fonctionnalités Principales:

1. Prévention de Navigation: Permet d'arrêter la navigation vers une route spécifique.
2. Gestion d'Erreurs: Peut lancer une erreur personnalisée lors de l'interruption.
3. Utilisation dans Middleware: Fonctionne uniquement dans le contexte d'un middleware de route.



Gestion des Interruptions de Navigation avec `abortNavigation`

Utilisation dans un Middleware pour la Sécurité d'Accès:

```
export default defineNuxtRouteMiddleware((to, from) => {
  const user = useState('user')

  if (!user.value.isAuthorized) {
    return abortNavigation('Utilisateur non autorisé')
  }

  if (to.path !== '/edit-post') {
    return navigateTo('/edit-post')
  }
})
```



Ajout Dynamique de Middleware de Route avec `addRouteMiddleware`

`addRouteMiddleware` est une fonction utilitaire conçue pour ajouter dynamiquement des middlewares de route dans une application Nuxt. Les middlewares de route sont des gardiens de navigation qui peuvent être stockés dans le répertoire `middleware/` de votre application Nuxt (à moins d'une configuration différente).



Ajout Dynamique de Middleware de Route avec `addRouteMiddleware`

Fonctionnalités Principales:

1. Ajout Dynamique de Middleware: Permet d'ajouter des middlewares à la volée, utile pour les plugins et les configurations personnalisées.
2. Support des Middleware Anonymes et Nommés: Prise en charge des middlewares sans nom et avec nom.
3. Option de Middleware Global: Possibilité de définir un middleware comme global, affectant ainsi toutes les routes.



Ajout Dynamique de Middleware de Route avec `addRouteMiddleware`

Middleware Anonyme:

```
// plugins/my-plugin.ts
export default defineNuxtPlugin(() => {
  addRouteMiddleware((to, from) => {
    if (to.path === '/forbidden') {
      return false
    }
  })
})
```



Ajout Dynamique de Middleware de Route avec addRouteMiddleware

Middleware Nommé:

```
// plugins/my-plugin.ts
export default defineNuxtPlugin(() => {
  addRouteMiddleware('named-middleware', () => {
    console.log('Middleware nommé ajouté dans le plugin Nuxt')
  })
})
```



Ajout Dynamique de Middleware de Route avec addRouteMiddleware

Middleware Global:

```
// plugins/my-plugin.ts
export default defineNuxtPlugin(() => {
  addRouteMiddleware('global-middleware', (to, from) => {
    console.log('Middleware global s'exécutant à chaque changement')
  },
  { global: true }
})
})
```



Invalidation des Données Mises en Cache avec clearNuxtData

clearNuxtData est une fonction qui permet de supprimer les données mises en cache, les statuts d'erreur et les promesses en attente générés par useAsyncData et useFetch. Cette méthode est particulièrement utile pour invalider les données récupérées pour une autre page, permettant ainsi une actualisation ou une mise à jour des données affichées.



Invalidation des Données Mises en Cache avec clearNuxtData

1. Suppression de Données en Cache: Efface les données mises en cache par `useAsyncData` et `useFetch`.
2. Gestion Flexible des Clés: Permet de spécifier une ou plusieurs clés pour lesquelles les données doivent être invalidées.
3. invalide Toutes les Données si Aucune Clé n'est Fournie: Par défaut, invalide toutes les données si aucun paramètre n'est passé.



Invalidation des Données Mises en Cache avec clearNuxtData

Invalider des Données Spécifiques:

```
// Invalidation d'une clé spécifique  
clearNuxtData('maCleDeDonnees')
```

Invalider des Données Spécifiques:

```
// Invalidation de plusieurs clés  
clearNuxtData(['cle1', 'cle2', 'cle3'])
```

Invalider Toutes les Données:

```
// Invalidation de toutes les données  
clearNuxtData()
```



Invalidation de l'État Mis en Cache avec clearNuxtState

clearNuxtState est une fonction qui permet de supprimer l'état mis en cache par useState. Cette méthode est utile pour invalider l'état stocké par useState, notamment lorsqu'une actualisation ou une réinitialisation de l'état de l'application est nécessaire.



Invalidation de l'État Mis en Cache avec clearNuxtState

1. Suppression de l'État en Cache: Efface l'état stocké par useState.
2. Gestion Flexible des Clés: Permet de cibler une ou plusieurs clés spécifiques pour lesquelles l'état doit être invalidé.
3. Invalidate Tout l'État si Aucune Clé n'est Fournie: Par défaut, invalide tout l'état si aucun paramètre n'est passé.



Invalidation de l'État Mis en Cache avec clearNuxtState

Invalider des Données Spécifiques:

```
// Invalidation d'une clé spécifique  
clearNuxtState('maCleEtat')
```

Invalider des Données Spécifiques:

```
// Invalidation de plusieurs clés  
clearNuxtState(['etat1', 'etat2', 'etat3'])
```

Invalider Toutes les Données:

```
// Invalidation de tout l'état  
clearNuxtState()
```



Création de Middleware de Route avec `defineNuxtRouteMiddleware`

`defineNuxtRouteMiddleware` est une fonction utilitaire permettant de créer des middlewares de route nommés dans Nuxt 3. Ces middlewares de route, généralement stockés dans le dossier `middleware/`, sont utilisés pour gérer la navigation et les autorisations avant le chargement des routes.



Création de Middleware de Route avec defineNuxtRouteMiddleware

1. Définition de Middleware de Route: Permet de créer des middlewares de route personnalisés.
2. Gestion de la Navigation et des Autorisations: Idéal pour contrôler l'accès aux routes en fonction de certaines conditions.
3. Utilisation des Fonctions de Navigation Globales: Intègre les fonctions d'aide telles que `navigateTo` et `abortNavigation`.



Création de Middleware de Route avec defineNuxtRouteMiddleware

Middleware pour Afficher une Page d'Erreur:

```
// middleware/error.ts
export default defineNuxtRouteMiddleware((to) => {
  if (to.params.id === '1') {
    throw createError({ statusCode: 404, statusMessage: 'Page
  }
})
```



Création de Middleware de Route avec defineNuxtRouteMiddleware

Middleware pour la Redirection Basée sur l'Authentification:

```
// middleware/auth.ts
export default defineNuxtRouteMiddleware((to, from) => {
  const auth = useState('auth')

  if (!auth.value.isAuthenticated) {
    return navigateTo('/login')
  }

  if (to.path !== '/dashboard') {
    return navigateTo('/dashboard')
  }
})
```



Définir la Métadonnée des Pages avec `definePageMeta`

`definePageMeta` est une macro de compilation utilisée pour définir des métadonnées personnalisées pour les composants de page dans Nuxt 3. Cette fonctionnalité est particulièrement utile pour configurer des métadonnées spécifiques à chaque route statique ou dynamique de votre application Nuxt, comme la gestion des layouts, les transitions, ou encore les middlewares de route.



Définir la Métadonnée des Pages avec `definePageMeta`

1. Personnalisation des Métadonnées de Page: Permet de définir des propriétés spécifiques pour chaque page.
2. Configuration Flexible: Inclut des options pour les transitions, la validation des routes, le keep-alive, et plus.
3. Définition Directe dans les Pages: Peut être utilisé directement dans les scripts setup des composants de page.



Définir la Métadonnée des Pages avec `definePageMeta`

Utilisation Basique pour une Page Spécifique:

```
// pages/some-page.vue
<script setup lang="ts">
  definePageMeta({
    layout: 'default'
  })
</script>
```



Définir la Métadonnée des Pages avec definePageMeta

Définition d'un Middleware de Route:

```
// pages/some-page.vue
<script setup lang="ts">
definePageMeta({
  middleware: [
    function (to, from) {
      const auth = useState('auth')
      if (!auth.value.authenticated) {
        return navigateTo('/login')
      }
      if (to.path !== '/checkout') {
        return navigateTo('/checkout')
      }
    }
  ]
})
</script>
```



Définir la Métadonnée des Pages avec `definePageMeta`

Définition d'un Layout Personnalisé:

```
// pages/some-page.vue
<script setup lang="ts">
  definePageMeta({
    layout: 'admin'
  })
</script>
```



Exécution de Code Après Initialisation avec `onNuxtReady`

`onNuxtReady` est un composable de Nuxt 3 qui permet d'exécuter un callback après que votre application ait terminé son initialisation. Cette fonction est exécutée uniquement côté client et est idéale pour du code qui ne doit pas bloquer le rendu initial de votre application, tel que le chargement de bibliothèques lourdes ou l'exécution de scripts d'analyse.



Exécution de Code Après Initialisation avec `onNuxtReady`

1. Exécution Côté Client: Fonctionne uniquement après le chargement complet de l'application côté client.
2. Non-Blocant: Idéal pour les opérations qui ne doivent pas interférer avec le rendu initial.
3. Exécution Sécurisée: Même si l'application est déjà initialisée, le code sera exécuté lors du prochain callback d'inactivité.



Exécution de Code Après Initialisation avec `onNuxtReady`

Utilisation dans un Plugin Client-Side:

```
// plugins/ready.client.ts
export default defineNuxtPlugin(() => {
  onNuxtReady(async () => {
    const myAnalyticsLibrary = await import('my-big-analytics-library')
    // Utilisation de myAnalyticsLibrary
  })
})
```



Rafraîchir les Données du Serveur avec `refreshNuxtData`

`refreshNuxtData` est un composable dans Nuxt 3 qui permet de rafraîchir et de mettre à jour toutes les données récupérées depuis le serveur. Il ré-exécute toutes les requêtes de données faites par `useAsyncData`, `useLazyAsyncData`, `useFetch`, et `useLazyFetch`, et invalide également leur cache. Cette fonction est particulièrement utile pour actualiser les données de la page en réponse à certains événements ou actions de l'utilisateur.



Rafraîchir les Données du Serveur avec refreshNuxtData

1. Rafraîchissement des Données: Recherche à nouveau toutes les données depuis le serveur.
2. Invalidate le Cache: Invalidate le cache des composables de récupération de données.
3. Sélection des Données à Rafraîchir: Option de rafraîchir toutes les données ou seulement celles spécifiées par des clés.



Rafraîchir les Données du Serveur avec refreshNuxtData

Rafraîchir Toutes les Données:

```
// pages/some-page.vue
<script setup lang="ts">
const refreshing = ref(false)
const refreshAll = async () => {
  refreshing.value = true
  try {
    await refreshNuxtData()
  } finally {
    refreshing.value = false
  }
}
</script>

<template>
  <div>
    <button :disabled="refreshing" @click="refreshAll">
      Rafraîchir Toutes les Données
    </button>
  </div>
</template>
```



Rafraîchir les Données du Serveur avec refreshNuxtData

Rafraîchir des Données Spécifiques:

```
// pages/some-page.vue
<script setup lang="ts">
  const { pending, data: count } = await useLazyAsyncData('count', () =>
    refresh()
  )
  const refresh = () => refreshNuxtData('count')
</script>

<template>
  <div>
    {{ pending ? 'Chargement' : count }}
  </div>
  <button @click="refresh">Rafraîchir</button>
</template>
```



Changement Dynamique de Layout avec setPageLayout

setPageLayout est un outil dans Nuxt 3 qui permet de changer dynamiquement le layout d'une page. Cette fonctionnalité est particulièrement utile pour modifier l'apparence d'une page en fonction du contexte ou des actions de l'utilisateur. setPageLayout doit être utilisée dans le contexte Nuxt, comme dans les fonctions setup des composants, les plugins, ou les middlewares de route.



Changement Dynamique de Layout avec setPageLayout

1. Changement Dynamique de Layout: Permet de modifier le layout d'une page pendant la navigation.
2. Utilisation dans le Contexte Nuxt: Doit être appelée dans des contextes spécifiques tels que les composants, les plugins ou les middlewares.
3. Adaptabilité du Design de Page: Idéal pour adapter le design d'une page en fonction de critères dynamiques.



Changement Dynamique de Layout avec setPageLayout

Utilisation dans un Middleware de Route:

```
// middleware/custom-layout.ts
export default defineNuxtRouteMiddleware((to) => {
    // Définit le layout pour la route vers laquelle on navigue
    setPageLayout('other')
})
```



Scaffold d'Entités avec nuxi add

nuxi add est une commande du terminal utilisée pour générer rapidement différentes entités dans une application Nuxt 3. Elle simplifie la création de composants, composables, layouts, plugins, pages, middleware, et API en structurant automatiquement les fichiers selon les conventions de Nuxt.



Scaffold d'Entités avec nuxi add

Fonctionnalités Principales:

1. Génération Rapide d'Entités: Crée des composants, composables, layouts, etc., en une seule commande.
2. Support des Modificateurs: Certains templates acceptent des drapeaux modificateurs pour ajouter des suffixes ou préciser le mode.
3. Personnalisation via les Options: Permet de spécifier le nom, le chemin, et d'autres options lors de la génération.



```
npx nuxi add api hello --get  
# Génère `server/api/hello.ts`
```

Scaffold d'Entités avec nuxi add

```
npx nuxi add page "category/[id]"  
# Génère `pages/category/[id].vue`  
  
npx nuxi add plugin sockets --client  
# Génère `/plugins/sockets.client.ts`  
  
npx nuxi add middleware auth --global  
# Génère `middleware/auth.ts`  
  
npx nuxi add component TheHeader  
# Génère `components/TheHeader.vue`  
  
npx nuxi add composable foo  
# Génère `composables/foo.ts`  
  
npx nuxi add layout custom  
# Génère `layouts/custom.vue`
```



Nettoyage des Fichiers Générés avec nuxi cleanup

nuxi cleanup est une commande du terminal conçue pour supprimer les fichiers générés et les caches courants dans une application Nuxt. Cette commande est utile pour résoudre des problèmes liés à des fichiers obsolètes ou corrompus et pour assurer que l'application fonctionne avec les fichiers et configurations les plus récents.



Nettoyage des Fichiers Générés avec nuxi cleanup

1. Suppression des Fichiers Générés: Nettoie les dossiers et fichiers automatiquement générés par Nuxt.
2. Ciblage de Caches Spécifiques: Inclut la suppression des caches de Nuxt, Vite, et autres caches de node_modules.
3. Utilisation Facile: Simple commande à exécuter dans le terminal.



Nettoyage des Fichiers Générés avec nuxi cleanup

Exécution de nuxi cleanup:

```
npx nuxi cleanup
```

Cette commande supprime les fichiers et dossiers suivants :

- .nuxt: Dossier de build de Nuxt.
- .output: Dossier de sortie de production.
- node_modules/.vite: Cache de Vite.
- node_modules/.cache: Autres caches dans node_modules.



Génération de Pages Statiques avec nuxi generate

nuxi generate est une commande utilisée pour pré-
rendre chaque route de votre application Nuxt en
fichiers HTML statiques. Cette fonctionnalité est
essentielle pour déployer des applications Nuxt sur des
services d'hébergement statiques, offrant des
performances optimales et un meilleur référencement
SEO.



Génération de Pages Statiques avec nuxi generate

Fonctionnalités Principales:

1. Pré-rendu de Routes: Convertit chaque route de l'application en fichiers HTML statiques.
2. Compatibilité avec l'Hébergement Statique: Permet de déployer l'application sur n'importe quel service d'hébergement statique.
3. Simplicité d'Exécution: Commande facile à utiliser dans le terminal.



Gestion des Modules Nuxt avec nuxi module

Les commandes nuxi module add et nuxi module search facilitent l'ajout et la recherche de modules dans les applications Nuxt. Ces utilitaires permettent une gestion efficace et simplifiée des modules, en réduisant le besoin de configuration manuelle.



Gestion des Modules Nuxt avec nuxi module

Fonctionnalités Principales:

1. Ajout Automatisé de Modules: nuxi module add installe les modules Nuxt et met à jour les fichiers de configuration automatiquement.
2. Recherche Facile de Modules: nuxi module search permet de trouver des modules compatibles avec votre version de Nuxt.
3. Simplicité et Gain de Temps: Réduit le besoin de manipulation manuelle des fichiers de configuration.



Gestion des Modules Nuxt avec nuxi module

Ajouter un Module:

```
npx nuxi module add pinia
```

Rechercher un Module:

```
npx nuxi module search pinia
```