

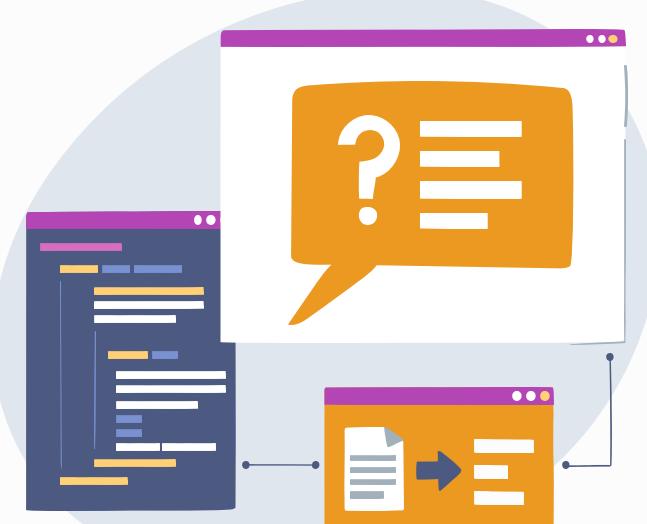
Vue 3



01

Rappels

ES6



Variables et Types de Données

Les variables en JavaScript peuvent être déclarées avec let, const, ou var (moins recommandé). let permet de déclarer des variables dont la valeur peut changer, tandis que const est pour des valeurs constantes.

Les types de données incluent les types primitifs (string, number, boolean, null, undefined, symbol) et les types non primitifs (objets, tableaux).

```
let age = 25; // Variable qui peut changer
const name = "John"; // Constante
let isStudent = false; // Type booléen
let score = null; // Type null
let x; // Undefined
const person = { firstName: "Alice", lastName: "Doe" }; // Objet
let numbers = [1, 2, 3]; // Tableau
```

Maîtriser les Opérateurs

```
let x = 10;
let y = 5;
let z = x + y; // 15
x += y; // x = x + y
let isEqual = x === y; // false
let isGreaterThan = x > y; // true
let andOperation = (x > 5) && (y < 10); // true
let orOperation = (x < 5) || (y > 2); // true
```

Les opérateurs en JavaScript incluent les opérateurs arithmétiques (+, -, *, /, %), d'assignation (=, +=, -=, etc.), de comparaison (==, ===, !=, !==, >, <, >=, <=) et logiques (&&, ||, !).

Structures de Contrôle

Les structures de contrôle dirigent le flux d'exécution du programme. Les conditions utilisent if, else if, else, et switch pour exécuter différents blocs de code basés sur des conditions. Les boucles for, while, et do...while permettent de répéter l'exécution d'un bloc de code.

```
if (x > 5) {
    console.log("x est supérieur à 5");
} else {
    console.log("x est inférieur ou égal à 5");
}

for (let i = 0; i < 5; i++) {
    console.log(i);
}

let i = 0;
while (i < 5) {
    console.log(i);
    i++;
}
```

Tout sur les Fonctions

Les fonctions peuvent être définies de plusieurs manières: déclarations de fonction, expressions de fonction, et fonctions fléchées.

```
function sayHello() {  
  console.log("Hello!");  
}  
  
const sayGoodbye = function() {  
  console.log("Goodbye!");  
};  
  
const add = (x, y) => x + y;  
  
sayHello(); // Affiche "Hello!"  
sayGoodbye(); // Affiche "Goodbye!"  
console.log(add(5, 3)); // 8
```

Manipulation d'Objets

```
const person = {  
    firstName: "John",  
    lastName: "Doe",  
    greet: function() {  
        console.log("Hello, " + this.firstName);  
    }  
};  
  
console.log(person.firstName); // John  
person.lastName = "Smith";  
person.greet(); // Hello, John
```

Les objets en JavaScript sont des collections de paires clé/valeur. La notation littérale permet de créer des objets, et on accède ou modifie leurs propriétés à l'aide de la notation pointée ou des crochets. Les méthodes d'objet sont des fonctions associées à des objets.

Exploiter les Tableaux

Les tableaux en JavaScript peuvent être manipulés et itérés à l'aide de méthodes telles que `.map()`, `.filter()`, `.reduce()`, et `.forEach()`. Ces méthodes permettent de traiter les éléments d'un tableau de manière efficace et concise.

```
const numbers = [1, 2, 3, 4, 5];

const squared = numbers.map(x => x * x);
console.log(squared); // [1, 4, 9, 16, 25]

const even = numbers.filter(x => x % 2 === 0);
console.log(even); // [2, 4]
```

```
numbers.forEach(x => console.log(x));
```

Fonctions Fléchées (=>)

Les fonctions fléchées offrent une syntaxe plus concise pour écrire des fonctions en JavaScript, permettant d'écrire des expressions de fonction plus courtes et directes. Elles sont particulièrement utiles pour les fonctions anonymes et les callbacks.

```
const numbers = [1, 2, 3, 4, 5];
// Exemple avec map utilisant une fonction fléchée
const squared = numbers.map(x => x * x);
console.log(squared); // Affiche [1, 4, 9, 16, 25]

// Fonction fléchée sans argument
const sayHello = () => console.log("Hello!");
sayHello();

// Fonction fléchée avec plusieurs arguments
const add = (a, b) => a + b;
console.log(add(5, 7)); // Affiche 12

// Fonction fléchée avec corps étendu
const multiply = (a, b) => {
  const result = a * b;
  return result;
};
console.log(multiply(2, 3)); // Affiche 6
```

Modularité avec les modules ES6

```
// file: math.js
export const add = (a, b) => a + b;
export const subtract = (a, b) => a - b;

// file: app.js
import { add, subtract } from './math.js';
console.log(add(2, 3)); // 5
console.log(subtract(5, 2)); // 3
```

Les modules permettent une meilleure organisation du code en séparant les fonctionnalités en différents fichiers, favorisant la réutilisation et la maintenance.

Promesses et Async/Await

```
// Function to perform a GET request using Fetch and return a promise
function fetchExample(url) {
  return fetch(url)
    .then(response => {
      // Check if the response is OK (status 200)
      if (response.ok) {
        // Return the response in JSON format
        return response.json();
      } else {
        // Throw an error with an appropriate message
        throw new Error(`Error ${response.status}: ${response.statusText}`);
      }
    })
    .then(data => {
      // Resolve the promise with the retrieved data
      return data;
    })
    .catch(error => {
      // Reject the promise with the encountered error
      throw error;
    });
}

// Example of using the fetchExample function
const url = 'https://api.example.com/data';

fetchExample(url)
  .then(data => {
    console.log('Data retrieved successfully:', data);
    // Do something with the retrieved data
  })
  .catch(error => {
    console.error('An error occurred while retrieving data:', error);
    // Handle the error appropriately
  });
}
```

Les promesses et la syntaxe `async/await` simplifient l'écriture de code asynchrone, rendant le traitement des opérations asynchrones plus lisible et facile à gérer.

Déstructuration pour une Meilleure Lisibilité

La destructuration permet d'extraire facilement des données des objets et des tableaux, rendant le code plus propre et plus lisible.

```
const person = { name: 'John', age: 30 };
const { name, age } = person;
console.log(name); // John
console.log(age); // 30

const numbers = [1, 2, 3];
const [first, , third] = numbers;
console.log(first); // 1
console.log(third); // 3
```

Paramètres par Défaut, Rest et Spread

Les valeurs par défaut pour les fonctions, l'opérateur Rest pour regrouper les arguments, et l'opérateur Spread pour étaler/expander des éléments offrent plus de flexibilité dans la gestion des données.

```
function greet(name, greeting = "Hello") {
  console.log(`#${greeting}, ${name}!`);
}

greet('John'); // Hello, John!
greet('John', 'Good morning'); // Good morning, John!

const parts = ['shoulders', 'knees'];
const body = ['head', ...parts, 'toes'];
console.log(body); // ["head", "shoulders", "knees", "toes"]
```

Template Literals

Les template literals permettent de construire des chaînes de caractères interpolés avec des expressions, rendant le code plus expressif et facilitant la création de chaînes multilignes.

```
const name = "John";
const greeting = `Hello, ${name}!
How are you today?`;
console.log(greeting);
```

Nouvelles Méthodes pour Objets et Tableaux

```
const numbers = [1, 2, 3, 4, 5];
console.log(numbers.find(x => x > 3)); // 4
console.log(numbers.includes(2)); // true

const person = { name: 'John', age: 30 };
console.log(Object.keys(person)); // ["name", "age"]
console.log(Object.values(person)); // ["John", 30]
```

ES6 et les versions ultérieures ont introduit de nouvelles méthodes pour travailler avec les objets et les tableaux, facilitant la recherche, la transformation, et la vérification des données.

L'Importance de l'Immutabilité

```
// Mauvaise pratique : modification directe d'un objet
let livre = { titre: "Programmation JavaScript" };
livre.titre = "Programmation Avancée"; // Modification directe

// Bonne pratique : immutabilité
const livreImmutable = Object.freeze({ titre: "Programmation JavaScript" });
const nouveauLivre = { ...livreImmutable, titre: "Programmation Avancée" };
```

L'immutabilité est un principe fondamental qui consiste à ne pas modifier directement les données. Au lieu de cela, toute modification produit une nouvelle instance des données. Cela aide à éviter les effets de bord et rend le code plus prévisible et facile à débugger.

Principes de la Programmation Fonctionnelle

La programmation fonctionnelle est un paradigme de programmation qui traite les calculs comme l'évaluation de fonctions mathématiques et évite les données changeantes ou mutables. Les principes clés incluent l'utilisation de fonctions pures et la composition de fonctions pour construire des logiciels.

```
// Fonction pure
const ajouter = (x, y) => x + y;

// Composition de fonctions
const multiplierParDeux = x => x * 2;
const ajouterEtMultiplier = (x, y) => multiplierParDeux(ajouter(x, y));

console.log(ajouterEtMultiplier(2, 3)); // 10
```

Stratégies de Gestion des Erreurs

La gestion des erreurs est cruciale pour développer des applications robustes.

L'utilisation de blocs try / catch permet de gérer les exceptions de manière élégante, tandis que les promesses offrent un mécanisme pour traiter les erreurs dans les opérations asynchrones.

```
// Utilisation de try / catch
try {
  const resultat = operationRisquee();
  console.log(resultat);
} catch (erreur) {
  console.error("Une erreur s'est produite:", erreur);
}

// Gestion des erreurs dans les promesses
fetch("https://api.exemple.com/data")
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(erreur => console.error("Erreur lors de la récupération des données:"))
```

Maîtriser la Gestion des Paquets

```
// package.json avec un script personnalisé
{
  "name": "mon-projet",
  "version": "1.0.0",
  "scripts": {
    "start": "webpack --mode development && node server.js",
    "build": "webpack --mode production"
  },
  "dependencies": {
    "react": "^17.0.1"
  },
  "devDependencies": {
    "webpack": "^5.22.0",
    "babel-loader": "^8.2.2"
  }
}
```

NPM et Yarn sont des gestionnaires de paquets pour JavaScript qui simplifient l'installation, la mise à jour et la gestion des dépendances de projets. Ils permettent également de définir et d'exécuter des scripts personnalisés pour automatiser les tâches de développement courantes.

Linting et Formatage avec ESLint et Prettier

```
// .eslintrc.json
{
  "extends": "eslint:recommended",
  "rules": {
    "no-unused-vars": "warn",
    "eqeqeq": ["error", "always"]
  }
}

// .prettierrc
{
  "semi": false,
  "singleQuote": true
}
```

ESLint et Prettier sont des outils essentiels pour maintenir la qualité du code JavaScript. ESLint analyse le code pour détecter les erreurs et les problèmes de style, tandis que Prettier reformate automatiquement le code selon des règles définies, assurant une cohérence stylistique.

Nouveautés Javascript

ECMAScript 6 (ES6) - 2015

Apport majeur :

Classes : Introduction de la syntaxe de classe pour la programmation orientée objet, qui est plus propre et facile à utiliser.

Modules : Standardisation du support des modules ES6 pour une meilleure gestion du code en facilitant l'import et l'export de composants.

Promesses : Pour la gestion asynchrone, facilitant l'écriture de code asynchrone et la gestion des opérations asynchrones.

Nouveautés Javascript

Autres apports :

let et const : Nouveaux mots-clés pour les déclarations de variables, offrant des portées de bloc, réduisant ainsi les erreurs communes dues aux variables globales.

Fonctions fléchées : Syntaxe plus concise pour l'écriture de fonctions, et ne crée pas de nouveau contexte this.

Paramètres par défaut, Rest et Spread : Amélioration de la gestion des paramètres de fonctions.

Destructuration : Permet une affectation plus facile et plus claire des données à partir d'arrays ou d'objets.

Templates littéraux : Intégration facile des variables dans les chaînes de caractères avec les templates littéraux.

ECMAScript 7 (ES7) - 2016

Opérateur d'exponentiation (**): Pour calculer la puissance d'un nombre.

Méthode Array.prototype.includes : Vérifie si un tableau inclut un élément donné, renvoyant true ou false.

ECMAScript 8 (ES8) - 2017

async et await : Simplification de l'écriture de fonctions asynchrones pour rendre le code asynchrone aussi facile à lire et à écrire que le code synchrone.

Méthodes de l'objet : Object.values(), Object.entries(), et Object.getOwnPropertyDescriptors() pour une meilleure manipulation des objets.

ECMAScript 9 (ES9) - 2018

Opérateur de décomposition pour objets : Étend les fonctionnalités de décomposition aux objets.

Promesses finally() : Méthode ajoutée aux promesses pour exécuter du code une fois que la promesse est réglée, indépendamment du résultat.

ECMAScript 10 (ES10) - 2019

`Array.prototype.{flat, flatMap}` : Méthodes pour aplatiser des tableaux imbriqués et appliquer une fonction, puis aplatisir le résultat.

`Object.fromEntries()` : Transforme une liste de paires clé-valeur en un objet.

Chaînes de caractères et modifications de `Array.sort` : Améliorations mineures pour la manipulation de chaînes de caractères et la méthode `sort`.

ECMAScript 11 (ES11) - 2020

BigInt : Introduit un type pour représenter des entiers très grands.

Promise.allSettled : Une nouvelle méthode de promesse qui renvoie un tableau de résultats après que toutes les promesses données se soient résolues ou rejetées.

Dynamique import() : Importations de modules sur demande pour améliorer la performance du chargement de modules.

Introduction à Vite

Vite est un outil de build moderne conçu pour les projets JavaScript, TypeScript, et les frameworks tels que Vue, React, et Svelte. Il offre un démarrage rapide de serveur de développement et des rechargements à chaud ultra-rapides en tirant parti de l'importation de modules ES natifs dans les navigateurs et en pré-bundling des dépendances avec esbuild.

```
# Installation de Vite pour un nouveau projet React
npm create vite@latest mon-projet-react -- --template react

# Démarrage du projet
cd mon-projet-react
npm install
npm run dev
```

Configuration de Vite

```
// vite.config.js
import reactRefresh from '@vitejs/plugin-react-refresh';

export default {
  plugins: [reactRefresh()],
  resolve: {
    alias: {
      '@': '/src',
    },
  },
  build: {
    outDir: 'build',
  },
};
```

Vite est hautement configurable via son fichier `vite.config.js`. Vous pouvez ajuster la résolution des modules, les plugins, les options CSS, et bien plus, permettant une personnalisation profonde pour répondre aux besoins spécifiques de votre projet.

Optimisations et Déploiement avec Vite

Vite offre des fonctionnalités d'optimisation prêtes à l'emploi pour le déploiement de production, telles que la minification, le découpage de code (code splitting), et le préchargement des modules. Ces optimisations améliorent la performance de chargement de l'application en production.

```
# Commande pour construire un projet pour la production
npm run build
```

```
# Analyse du bundle pour optimisation
npm run preview
```

02

Type**Script**



TypeScript, c'est quoi ?

TypeScript est un sur-ensemble de JavaScript développé par Microsoft, ayant pour but d'ajouter un système de typage statique. Cela nous permettra d'écrire un code plus sécurisé, plus facilement maintenable, et de détecter des erreurs avant même l'exécution du code.

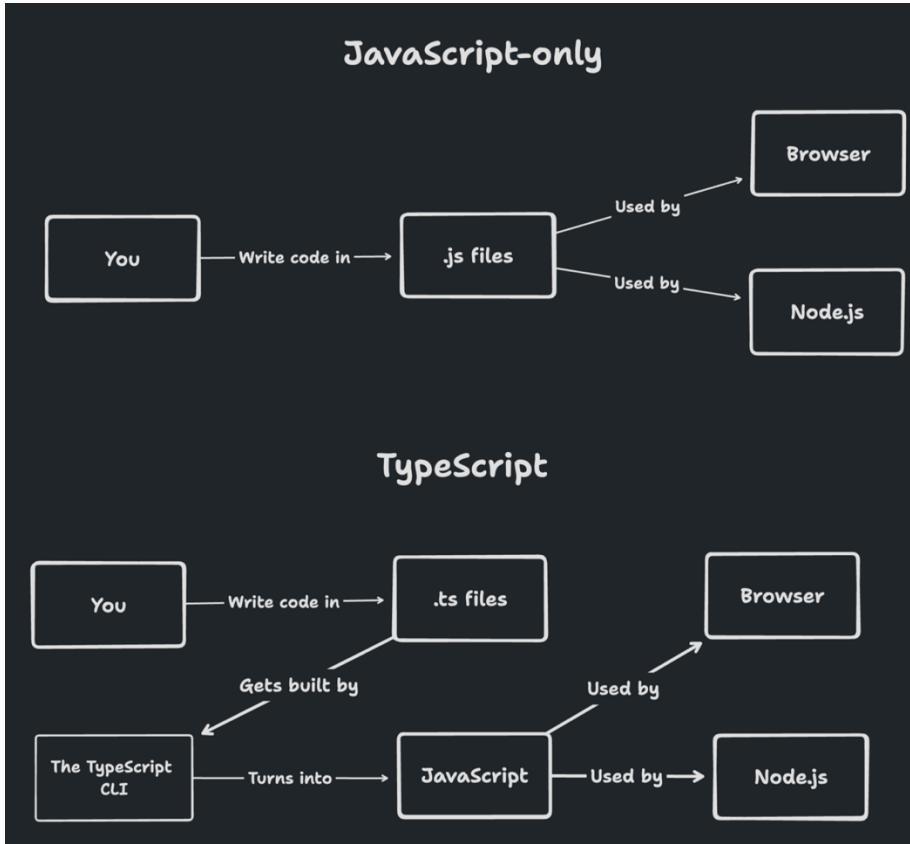
Le choix d'utiliser ou non TypeScript dépend de vos préférences en termes de sécurité et de flexibilité.

Les fichiers TypeScript auront une extension en .ts ou .tsx

Les avantages de **TypeScript**

1. Syntaxe proche du JavaScript
2. Permet de détecter des erreurs potentielles, même sans déclarer de type
3. Facilite la documentation du code
4. Permet à l'IDE de gérer les suggestion de code plus facilement
5. Meilleure maintenabilité
6. L'inférence de type

Le fonctionnement de **TypeScript**



La syntaxe TypeScript

Comme noté précédemment, nos fichiers auront une extension .ts ou .tsx.

Pour définir une variable avec un type, la syntaxe ressemblera à :

Déclaration (let ou const) variable : type = valeur

```
let person : string = "Jean";
let age : number = 30;
let isMajor : boolean = true;

person = age; //Impossible d'assigner le type number à une variable de type string
isMajor = isMajor + person; //Impossible d'additionner une variable de type boolean avec une variable de type string
```

Les types primitifs

TypeScript nous fournit 3 types primitifs, et 2 types utilitaires:

1. number
2. string
3. boolean
4. unknown (quand on ne sait pas quel type nous allons recevoir, on y reviendra après)
5. any (à éviter)

La syntaxe TypeScript

Comme noté précédemment, nos fichiers auront une extension .ts ou .tsx.

Pour définir une variable avec un type, la syntaxe ressemblera à :

Déclaration (let ou const) variable : type = valeur

```
let person : string = "Jean";
let age : number = 30;
let isMajor : boolean = true;
function sayHello(name : string) : string {
    return "Hello " + name;
}
person = age; //Impossible d'assigner le type number à une variable de type string
isMajor = isMajor + person; //Impossible d'additionner une variable de type boolean avec une variable de type string

sayHello(isMajor) // string attendu en paramètre, mais on passe un boolean
```

L'inférence de type

C'est sympa de pouvoir déclarer des types primitifs sur les variables lorsqu'on les déclare, mais ça va vite devenir très répétitif non ?

C'est là que l'inférence de type entre en scène: TS va être capable de « déduire » le type de certaines variables et signatures de fonction via leur valeurs d'origine. Pas besoin de tout typer explicitement !

```
let person = "Jean";
let age = 30;
let isMajor = true;
function sayHello(name : string)  {
    return "Hello " + name;
}
person = age; //Même erreur d'auparavant
isMajor = isMajor + person; //Idem

sayHello(isMajor) // Toujours pareil
```

L'inférence de type

Attention, TS n'est pas omniscient, et ne sera pas toujours capable d'inférer les types correctement.

Si il ne sait pas, il utilisera le type any, qui ne procède à aucune vérification !

Par exemple :

```
const myArray = []; //Type implicite: any[]  
myArray.push(1) //Pas d'erreur car non strict  
myArray.push('Hello') //Pas d'erreur car non strict, mais dangereux
```

Types non conformes à la réalité

TypeScript nous permet de détecter des erreurs dans le code, mais peut parfois autoriser des opérations qui généreraient des erreurs à l'exécution.

```
const names = ["Charles", "Pierrick"]
console.log(names[2].toLowerCase())
//TS ne signale pas d'erreur, mais la dernière ligne va générer une erreur à l'exécution
```

Type et Interface

TypeScript nous permet de créer nos propres types, grâce aux mots clés « type » et « interface ».

```
type Age = number;
interface Person {
    name: string;
    age: Age;
}
let driver: Person = {
    name: 'James May',
    age: 56
};
driver.age = 57; // OK
driver.age = '57'; // Error
```

Type ou Interface ?

Type et interface peuvent sembler interchangeables, car ils le sont majoritairement.

Normalement, type est plutôt utilisé pour définir les alias, et interface pour typer des objets complets. À notre niveau, nous pouvons nous contenter d'utiliser type au général, car ils nous permettent de faire des « union types » qui nous seront utiles pour la suite.

```
type item = {  
    name: string,  
    price: number,  
}
```

```
type weapon = item & {  
    damage: number,  
}
```

```
interface item {  
    name: string;  
    price: number;  
}
```

```
interface weapon extends item {  
    damage: number;  
}
```

Union types et Narrowing

Avec le mot clé `type`, nous pouvons créer des « union types », c'est-à-dire des variables qui peuvent accepter plusieurs types, par exemple:

```
type StringOrNumber = string | number;  
  
let myAge: StringOrNumber = 25;  
myAge = '25'; // OK
```

Pour savoir quel est le type réel d'une variable avec une union, on peut utiliser l'inférence de TS :

```
const doSomething = (value: StringOrNumber) => {  
    if (typeof value === 'string') {  
        //TypeScript infère que c'est une string  
        return value.toUpperCase();  
    }  
    //TypeScript infère que c'est un number  
    return value;  
}
```

Narrowing sur des objets ?

Pour faire du narrowing sur des types complexes, nous pouvons utiliser le mot clé « in » pour déterminer ou non l'existence d'une propriété propre à un des types.

```
type Fish = {  
    swim : () => void;  
}  
type Bird = {  
    fly : () => void;  
}  
  
function move(animal: Fish | Bird){  
    if("swim" in animal){  
        return animal.swim();  
    }  
    return animal.fly();  
}
```

Les génériques

Les génériques sont une fonctionnalité de TS qui nous permettent de créer des types génériques réutilisables et de réduire le code à écrire. Prenons l'exemple ci-dessous:

```
type StringCollection = {
    name: string;
    items : string[];
}

type NumberCollection = {
    name: string;
    items : number[];
}

type BooleanCollection = {
    name: string;
    items : boolean[];
}
```

Les génériques

On peut créer un type générique Collection, qui prendra en « paramètre » un type T qui sera assigné à la propriété items. Nos types précédents seront donc déclarées par le générique.

```
type Collection<T> = {  
    name: string;  
    items : T[];  
}  
type StringCollection = Collection<string>;  
type NumberCollection = Collection<number>;  
type BooleanCollection = Collection<boolean>;
```

Les génériques

Il existe des génériques utilitaires, comme `<Partial>`, `<Omit>`, `<Record>`... que nous utiliserons peut être lors de la formation.

Vous pouvez également les retrouver sur la documentation TypeScript, juste ici:
<https://www.typescriptlang.org/docs/handbook/utility-types.html>

Pour aller plus loin

Compte Twitter et Youtube et Matt Pocock

<https://twitter.com/mattpcockuk>

<https://www.youtube.com/@mattpcockuk>

03

Introduction à Vue



Introduction à Vue.js

- **Qu'est-ce que Vue.js ?**
- **Framework JavaScript progressif** : Conçu pour être adopté progressivement.
- **Créé par Evan You** : Lancé en 2014.
- **Objectif** : Simplifier la construction d'interfaces utilisateur (UI) avec une approche déclarative et réactive.
- **Caractéristiques principales :**
- **Réactivité** : Mise à jour automatique de la vue lors des changements de données.
- **Composants** : Code réutilisable et structuré.
- **Écosystème riche** : Vue Router, Vuex, Nuxt.js, etc.

Points Forts de Vue.js

- **Avantages :**
 - **Facilité d'apprentissage** : Documentation claire et bien organisée.
 - **Performances élevées** : Virtual DOM pour des mises à jour efficaces.
 - **Intégration facile** : Peut être intégré progressivement dans des projets existants.
 - **Communauté active** : Nombreux plugins, bibliothèques et outils.
- **Cas d'utilisation :**
 - Applications SPA (Single Page Applications)
 - Projets progressifs avec montée en complexité
 - Prototypes rapides et projets de petite taille

Comparaison avec React et Angular

- **Vue.js vs React :**
- **Architecture :**
 - **Vue.js** : MVVM (Model-View-View-Model)
 - **React** : V (Vue seulement, nécessite des bibliothèques additionnelles)
- **Facilité d'utilisation :**
 - **Vue.js** : Plus simple pour les débutants.
 - **React** : Nécessite souvent un apprentissage plus approfondi de l'écosystème.
- **Taille :**
 - **Vue.js** : Plus léger, ~30KB gzipped.
 - **React** : Un peu plus lourd, ~40KB gzipped.

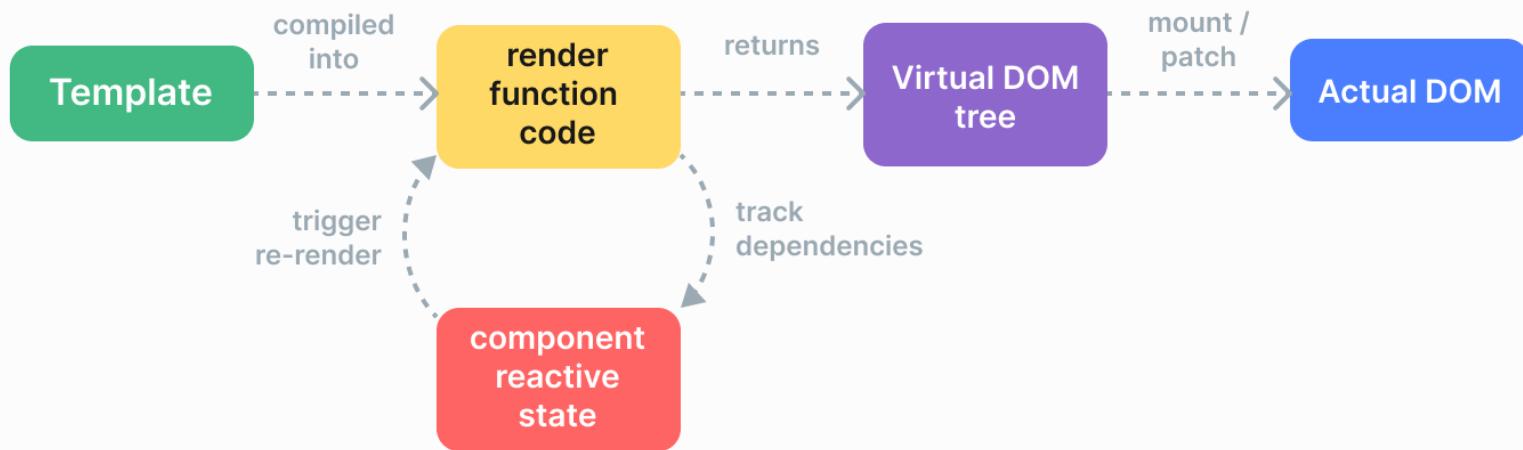
Comparaison avec React et Angular

- **Vue.js vs Angular :**
- **Complexité :**
 - **Vue.js** : Simplicité et légèreté.
 - **Angular** : Framework complet avec une courbe d'apprentissage plus abrupte.
- **Performances :**
 - **Vue.js** : Très performant avec une configuration minimale.
 - **Angular** : Performance élevée mais nécessite plus de configurations.
- **Utilisation :**
 - **Vue.js** : Adapté aux petites et moyennes applications.
 - **Angular** : Idéal pour les applications d'entreprise complexes.

Conclusion et Adoption de Vue.js

- **Conclusion :**
- **Polyvalence** : Vue.js est adapté à une large gamme de projets, des prototypes rapides aux applications complexes.
- **Écosystème en croissance** : De plus en plus d'outils et de bibliothèques pour soutenir le développement.
- **Support communautaire** : Documentation et assistance communautaire de haute qualité.
- **Adoption :**
- **Entreprises** : Alibaba, Xiaomi, GitLab utilisent Vue.js.
- **Popularité** : Croissance rapide et adoption large dans la communauté des développeurs.

Render Pipeline



Render Pipeline

Template

```
<h1>{{ pageTitle }}</h1>
```



Render Function

```
render: function (createElement) {  
  return createElement('h1', this.blogTitle)  
}
```

Render Function

```
render: function (createElement) {  
  return createElement('h1', this.blogTitle)  
}
```

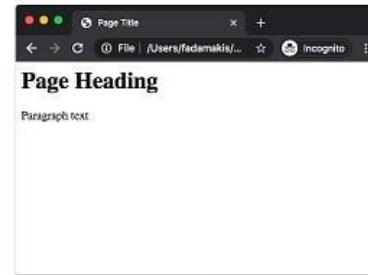


Virtual Dom Node

```
{  
  "tag": "h1",  
  "children": [  
    {  
      "text": "Page Heading"  
    }  
  ]  
}
```

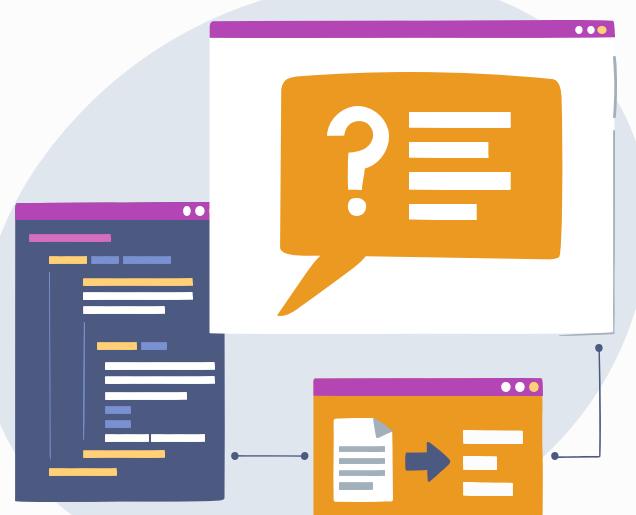


Rendered



04

Installation de Vue



Installation de Vue CLI

Vue CLI est un outil de ligne de commande qui facilite la création et la gestion de projets Vue. Il permet de démarrer rapidement avec une configuration par défaut ou personnalisée.

Structure de base d'un projet Vue

Un projet Vue typique comprend plusieurs fichiers et dossiers, notamment src, public, main.js, et App.vue. La structure permet une organisation claire et modulaire du code.

```
my-project/
├── node_modules/
├── public/
│   └── index.html
└── src/
    ├── assets/
    ├── components/
    ├── App.vue
    └── main.js
└── package.json
```

Vue Devtools

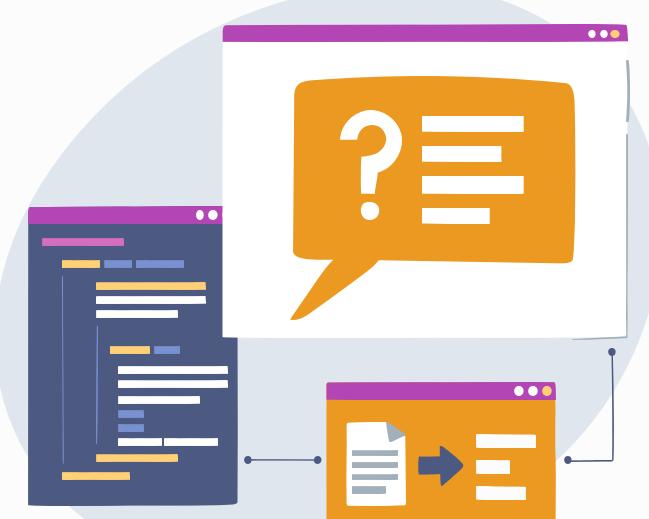
Vue Devtools est une extension de navigateur qui facilite le débogage et le développement des applications Vue. Elle offre des outils pour inspecter le composant, la réactivité, et bien plus encore.

```
# Instructions pour installer Vue Devtools:  
1. Installez l'extension pour votre navigateur.  
2. Ouvrez votre application Vue.  
3. Utilisez l'onglet Vue Devtools pour explorer vos composants.
```

05

Les

Composants



Introduction aux composants

Les composants sont des blocs de construction réutilisables dans Vue.js. Ils permettent de structurer et modulariser votre application, facilitant ainsi la maintenance et la réutilisation du code.

```
<template>
  <div>
    <Header />
    <Content />
    <Footer />
  </div>
</template>

<script setup>
import Header from './Header.vue';
import Content from './Content.vue';
import Footer from './Footer.vue';
</script>
```

Props dans les composants

Les props permettent de passer des données des composants parents aux composants enfants. Elles sont déclarées dans l'objet props du composant enfant.

```
<!-- ParentComponent.vue -->
<template>
  <div>
    <ChildComponent :message="parentMessage" />
  </div>
</template>

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

const parentMessage = ref('Message du parent');
</script>
```

```
<!-- ChildComponent.vue -->
<template>
  <div>
    <p>{{ message }}</p>
  </div>
</template>

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

const props = defineProps({
  message: {
    type: String,
    required: true
  }
});
</script>
```

Utiliser des modules ES6

Les modules ES6 permettent d'importer et d'exporter des fonctionnalités entre différents fichiers. Cela aide à garder le code organisé et maintenable.

```
<template>
  <div>
    <h1>{{ message }}</h1>
  </div>
</template>

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

const message = ref('Bonjour, Vue 3!');
</script>
```

Template Syntax

La syntaxe des templates dans Vue.js permet de lier des données à l'interface utilisateur de manière déclarative.

```
<template>
  <div>
    <p>{{ message }}</p>
  </div>
</template>

<script setup>
const message = 'Hello, World!';
</script>
```

Introduction à la syntaxe des templates

La syntaxe des templates dans Vue.js permet de créer des vues dynamiques en utilisant des expressions JavaScript directement dans le HTML.

```
<template>
  <div>
    <p>{{ 2 + 2 }}</p>
  </div>
</template>
```

Expressions dans les templates

Les expressions dans les templates Vue.js peuvent inclure des opérations arithmétiques, des appels de méthode, et bien plus.

```
<template>
  <div>
    <p>{{ message.toUpperCase() }}</p>
  </div>
</template>

<script setup>
const message = 'hello world';
</script>
```

Utilisation des Slots

Les slots sont une fonctionnalité puissante de Vue.js permettant d'insérer du contenu dans un composant enfant depuis un composant parent. Ils sont utiles pour créer des composants réutilisables et flexibles.

```
<!-- ChildComponent.vue -->
<template>
  <div>
    <header>
      <slot name="header"></slot>
    </header>
    <main>
      <slot></slot>
    </main>
    <footer>
      <slot name="footer"></slot>
    </footer>
  </div>
</template>

<script setup>
</script>
```

```
<!-- ParentComponent.vue -->
<template>
  <div>
    <ChildComponent>
      <template #header>
        <h1>Titre du Slot</h1>
      </template>
      <template #default>
        <p>Contenu principal du slot</p>
      </template>
      <template #footer>
        <p>Pied de page du slot</p>
      </template>
    </ChildComponent>
  </div>
</template>

<script setup>
import ChildComponent from './ChildComponent.vue';
</script>
```

Slot par Défaut

- Un slot par défaut est utilisé lorsque le parent ne fournit pas de contenu personnalisé.

```
<!-- Parent.vue -->
<template>
  <MyComponent />
</template>

<!-- MyComponent.vue -->
<template>
  <slot>Message par défaut</slot>
</template>
```

Slots Nommés

- Les slots nommés permettent d'insérer plusieurs contenus distincts dans un composant.

```
<!-- Parent.vue -->
<template>
  <MyComponent>
    <template #header>
      <h1>Header personnalisé</h1>
    </template>
    <template #footer>
      <p>Footer personnalisé</p>
    </template>
  </MyComponent>
</template>

<!-- MyComponent.vue -->
<template>
  <header>
    <slot name="header">Header par défaut</slot>
  </header>
  <footer>
    <slot name="footer">Footer par défaut</slot>
  </footer>
</template>
```

Slots Dynamiques

- Les slots dynamiques utilisent une clé dynamique pour déterminer quel contenu afficher.

```
<!-- Parent.vue -->
<template>
  <MyComponent>
    <template #[ "dynamic-slot"]>
      <p>Contenu dynamique</p>
    </template>
  </MyComponent>
</template>

<!-- MyComponent.vue -->
<template>
  <slot :name=" 'dynamic-slot'">Contenu par défaut</slot>
</template>
```

Passer des Données avec les Slots (Scoped Slots)

- Un **scoped slot** permet au composant enfant de transmettre des données au parent via un scope.

```
<!-- Parent.vue -->
<template>
  <MyComponent>
    <template #default="{ user }">
      <p>Nom de l'utilisateur : {{ user.name }}</p>
    </template>
  </MyComponent>
</template>

<!-- MyComponent.vue -->
<template>
  <slot :user="{ name: 'Alice' }" />
</template>
```

Slots avec des Fallbacks

- Un fallback est utilisé si aucun contenu n'est fourni pour un slot.

```
<!-- MyComponent.vue -->
<template>
  <slot name="header">
    <h2>Header par défaut</h2>
  </slot>
</template>
```

Réutilisation Avancée avec Slots

- Les slots permettent de créer des composants hautement réutilisables et personnalisables.

```
<!-- Parent.vue -->
<template>
  <DataTable :data="rows">
    <template #row="{ row }">
      <tr>
        <td>{{ row.name }}</td>
        <td>{{ row.age }}</td>
      </tr>
    </template>
  </DataTable>
</template>

<!-- DataTable.vue -->
<template>
  <table>
    <tbody>
      <template v-for="row in data" :key="row.id">
        <slot name="row" :row="row" />
      </template>
    </tbody>
  </table>
</template>
```

05.1

Les Composants Avancés



Introduction : `defineComponent` et `defineAsyncComponent`

Vue 3 introduit deux API puissantes pour définir et gérer des composants :

defineComponent : Simplifie la définition de composants en intégrant des types précis dans un environnement TypeScript.

defineAsyncComponent : Charge des composants de manière asynchrone, idéal pour optimiser le chargement des applications.

Ces outils permettent une meilleure intégration avec TypeScript et facilitent la gestion de gros projets Vue.js.

defineComponent : Définition et Utilité

Qu'est-ce que defineComponent ? Une fonction qui fournit des métadonnées pour indiquer à TypeScript comment interpréter un composant Vue. Cela améliore :

La vérification de type.

L'auto-complétion dans les IDE.

La sécurité des props, des événements et des émissaires.

Avantages de defineComponent avec TypeScript

Vérification des Props : Les types des props sont automatiquement vérifiés.

Auto-complétion des Événements : Permet de définir et d'écouter des événements personnalisés en toute sécurité.

Développement Modulaire : Facilite l'intégration des bibliothèques comme Vue Router ou Pinia avec des types explicites.

defineAsyncComponent : Définition et Utilité

Qu'est-ce que defineAsyncComponent ?

Une méthode pour charger des composants uniquement quand ils sont nécessaires. Idéal pour optimiser le lazy-loading dans des applications avec une navigation dynamique.

Avantages :

- Amélioration des Performances** : Réduit le temps de chargement initial.
- Modularité** : Permet de diviser le code en fichiers plus petits.
- Gestion des États (Chargement/Erreur)** : Fournit des options pour gérer les états intermédiaires.

Utilisation Basique de defineAsyncComponent

Un composant asynchrone peut être défini en important dynamiquement un module.

```
import { defineAsyncComponent } from 'vue';

const AsyncComponent = defineAsyncComponent(() =>
  import('./MyComponent.vue')
);

export default defineComponent({
  components: {
    AsyncComponent,
  },
});
```

Personnalisation du Comportement Asynchrone

defineAsyncComponent permet de personnaliser les états comme le chargement ou les erreurs.

```
import { defineAsyncComponent } from 'vue';

const AsyncComponent = defineAsyncComponent({
  loader: () => import('./MyComponent.vue'),
  loadingComponent: () => import('./LoadingSpinner.vue'),
  errorComponent: () => import('./ErrorFallback.vue'),
  delay: 200, // Délai avant d'afficher le spinner
  timeout: 3000, // Timeout pour le chargement
});

export default defineComponent({
  components: { AsyncComponent },
});
```

Gestion des Erreurs et des États dans defineAsyncComponent

Les composants asynchrones permettent une gestion fine des erreurs et des transitions.

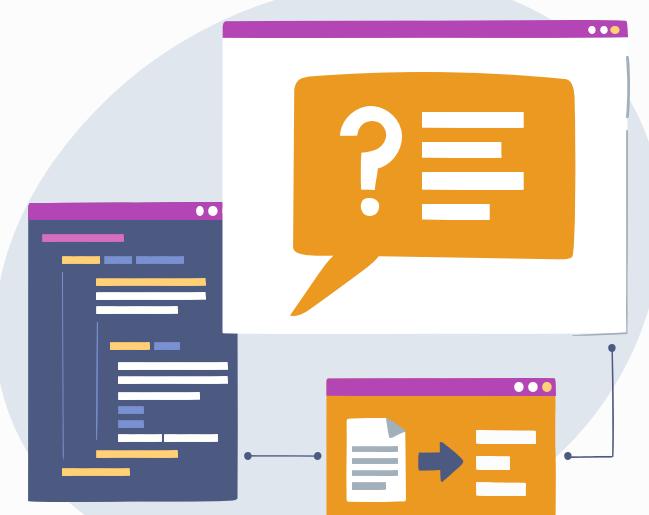
```
import { defineAsyncComponent } from 'vue';

const AsyncComponent = defineAsyncComponent({
  loader: () => import('./HeavyComponent.vue'),
  loadingComponent: () => import('./Loading.vue'),
  errorComponent: () => import('./Error.vue'),
  delay: 200,
  onError(error, retry, fail) {
    console.error('Erreur lors du chargement :', error);
    if (error.message.includes('Network Error')) {
      retry(); // Essaye de recharger
    } else {
      fail(); // Stoppe le chargement
    }
  },
});

export default defineComponent({
  components: { AsyncComponent },
});
```

05.2

Fallthrough Attributes



Qu'est-ce qu'un Fallthrough Attribute ?

Ce sont des attributs ou écouteurs d'événements transmis à un composant sans être explicitement déclarés comme props ou événements. Exemple : class, style, id, ou v-on listeners.

Comportement par défaut : Vue applique automatiquement ces attributs au nœud racine du composant.

```
<template>
  <MyButton class="large" />
</template>

<!-- MyButton.vue -->
<template>
  <button>Click Me</button>
</template>

<!-- DOM rendu -->
<button class="large">Click Me</button>
```

Fusion des Classes et Styles

Les attributs class et style du parent sont fusionnés avec ceux du composant enfant. Si le composant enfant possède ses propres class ou style, ils seront combinés.

```
<template>
  <MyButton class="large" style="color: red;" />
</template>

<!-- MyButton.vue -->
<template>
  <button class="btn" style="font-size: 14px;">Click Me</button>
</template>

<!-- DOM rendu -->
<button class="btn large" style="color: red; font-size: 14px;">Click Me</button>
```

Héritage des Écouteurs d'Événements

Les écouteurs d'événements comme @click sont automatiquement appliqués au nœud racine. Si le nœud racine possède déjà un écouteur pour le même événement, les deux seront déclenchés.

```
<template>
  <MyButton @click="onClickParent" />
</template>

<!-- MyButton.vue -->
<template>
  <button @click="onClickChild">Click Me</button>
</template>

<script>
export default {
  methods: {
    onClickChild() {
      console.log('Child Clicked');
    },
    onClickParent() {
      console.log('Parent Clicked');
    },
  },
};
</script>

<!-- Résultat console -->
// Parent Clicked
// Child Clicked
```

Attributs Hérités et Composants Imbriqués

Si un composant racine **rend un autre composant** comme élément principal, les Fallthrough Attributes sont transférés au composant enfant.

```
<!-- Parent.vue -->
<template>
  <MyButton class="primary" />
</template>

<!-- MyButton.vue -->
<template>
  <BaseButton />
</template>

<!-- BaseButton.vue -->
<template>
  <button class="btn">Click Me</button>
</template>

<!-- DOM rendu -->
<button class="btn primary">Click Me</button>
```

Désactiver l'Héritage des Attributs

Vous pouvez désactiver l'héritage des attributs avec `inheritAttrs: false`. Cela permet de prendre le contrôle total sur les attributs.

```
<script setup>
defineOptions({
  inheritAttrs: false
});
</script>

<template>
  <div class="wrapper">
    <button class="btn" v-bind="$attrs">Click Me</button>
  </div>
</template>
```

Accéder aux Attributs avec \$attrs

\$attrs regroupe tous les attributs hérités non déclarés comme props ou emits. Utilisez \$attrs pour appliquer les attributs sur des éléments spécifiques.

```
<template>
  <div class="wrapper">
    <button class="btn" v-bind="$attrs">Click Me</button>
  </div>
</template>
```

```
<script setup>
  import { useAttrs, onUpdated } from 'vue';

  const attrs = useAttrs();

  onUpdated(() => {
    console.log('Updated Attributes:', attrs);
  });
</script>
```

06

La réactivité



La réactivité dans Vue 3

Le système de réactivité de Vue.js est au cœur de son fonctionnement, permettant aux données de déclencher automatiquement des mises à jour de l'interface utilisateur.

```
<template>
  <div>
    <p>{{ state.message }}</p>
    <button @click="state.message = 'Bonjour !'">Change Message</button>
  </div>
</template>

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

const state = reactive({ message: 'Hello' });
</script>
```

Déclarer des variables réactives avec ref()

ref() est utilisé pour créer des variables réactives primitives dans Vue.js.

```
<template>
  <div>
    <p>{{ name }}</p>
    <input v-model="name" />
  </div>
</template>

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

const name = ref('John Doe');
</script>
```

Les objets réactifs avec reactive()

reactive() est utilisé pour créer des objets réactifs non primitifs dans Vue.js.

```
<template>
  <div>
    <p>{{ user.name }}</p>
    <input v-model="user.name" />
  </div>
</template>

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

const user = reactive({ name: 'John Doe' });
</script>
```

Comparaison

ref

- Meilleur pour les valeurs primitives (nombre, chaîne, booléen).
- Accès direct via .value.

reactive

- Meilleur pour les objets complexes.
- Propriétés accessibles directement.

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

// Utilisation de `ref`
const count = ref(0);

// Utilisation de `reactive`
const state = reactive({ count: 0 });
```

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 }
```

Introduction aux propriétés calculées

Les propriétés calculées sont des propriétés dérivées des données réactives, recalculées automatiquement lorsque les données source changent.

```
<template>
  <div>
    <p>{{ fullName }}</p>
  </div>
</template>

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

const firstName = ref('John');
const lastName = ref('Doe');

const fullName = computed(() => `${firstName.value} ${lastName.value}`);
</script>
```

06.1

La réactivité avancée



Introduction à la Réactivité Avancée

Vue 3 offre des outils avancés pour gérer la réactivité avec plus de contrôle et de flexibilité. Ces fonctionnalités sont utiles dans des cas spécifiques où les comportements standard ne suffisent pas ou nécessitent des optimisations.

shallowRef : Référence peu profonde

shallowRef crée une référence réactive où seule la propriété elle-même est réactive, mais les propriétés imbriquées ne le sont pas.

```
import { shallowRef } from 'vue';

const shallowValue = shallowRef({ nested: { value: 42 } });

// Seule `shallowValue.value` est réactive.
shallowValue.value = { nested: { value: 99 } }; // Réactif
shallowValue.value.nested.value = 100; // Pas réactif

console.log(shallowValue.value.nested.value); // 100
```

triggerRef : Forcer une mise à jour

triggerRef force une mise à jour manuelle d'une référence, utile avec shallowRef pour réagir aux changements dans des objets imbriqués.

```
import { shallowRef, triggerRef } from 'vue';

const data = shallowRef({ nested: { value: 42 } });

// Modifie une propriété interne (non réactif avec shallowRef)
data.value.nested.value = 99;

// Force une mise à jour réactive
triggerRef(data);
```

customRef : Référence personnalisée

Permet de créer une référence avec un comportement réactif personnalisé, par exemple pour appliquer un délai ou une validation.

```
import { customRef } from 'vue';

function debouncedRef(value, delay) {
  let timeout;
  return customRef((track, trigger) => ({
    get() {
      track(); // Suivre les dépendances
      return value;
    },
    set(newValue) {
      clearTimeout(timeout);
      timeout = setTimeout(() => {
        value = newValue;
        trigger(); // Notifier les changements
      }, delay);
    }
  }));
}

const myRef = debouncedRef('Bonjour', 300);
```

shallowReactive : Objet réactif peu profond

shallowReactive crée un objet réactif, mais uniquement pour les propriétés directes. Les objets imbriqués ne sont pas réactifs.

```
import { shallowReactive } from 'vue';

const state = shallowReactive({
  nested: { value: 42 },
});

state.newProp = 'Hello'; // Réactif
state.nested.value = 99; // Pas réactif
```

shallowReadonly : Objet en lecture seule peu profond

shallowReadonly crée un objet en lecture seule où seules les propriétés directes sont protégées contre les modifications.

```
import { shallowReadonly } from 'vue';

const state = shallowReadonly({
  nested: { value: 42 },
});

// Interdit
state.newProp = 'Hello'; // Erreur
state.nested.value = 99; // Pas d'erreur (pas protégé)
```

toRaw : Obtenir la version brute d'un objet réactif

toRaw retourne l'objet brut d'origine d'un objet réactif. Utile pour éviter la réactivité dans certaines situations.

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

const reactiveObject = reactive({ value: 42 });
const rawObject = toRaw(reactiveObject);

console.log(rawObject.value); // 42
```

markRaw : Exclure un objet de la réactivité

markRaw empêche un objet d'être rendu réactif, ce qui est utile pour des objets tiers (comme des instances de classes).

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

class CustomClass {
  constructor(value) {
    this.value = value;
  }
}

const instance = markRaw(new CustomClass(42));
const state = reactive({ instance });

state.instance.value = 99; // Pas réactif
```

effectScope : Isoler des effets réactifs

effectScope permet de regrouper plusieurs effets réactifs dans un conteneur isolé, qui peut être facilement nettoyé.

```
import { effectScope, reactive, watch } from 'vue';

const scope = effectScope();

scope.run(() => {
  const state = reactive({ count: 0 });
  watch(() => state.count, (newVal) => {
    console.log(`Count: ${newVal}`);
  });
});

// Nettoyer tous les effets du scope
scope.stop();
```

getCurrentScope : Accéder au scope actuel

getCurrentScope retourne le scope d'effet actuel, si disponible. Utile pour vérifier ou associer des nettoyages.

```
import { effectScope, getCurrentScope } from 'vue';

effectScope(() => {
  const scope = getCurrentScope();
  console.log(scope ? 'Dans un scope' : 'Pas dans un scope');
});
```

onScopeDispose : Nettoyage automatique dans un scope

onScopeDispose enregistre une fonction de nettoyage qui sera appelée automatiquement lorsque le scope sera détruit.

```
import { effectScope, onScopeDispose } from 'vue';

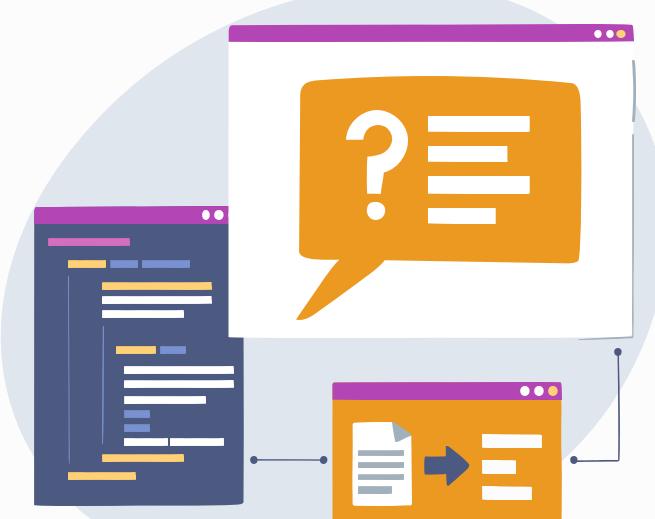
const scope = effectScope(() => {
  const interval = setInterval(() => console.log('Tick'), 1000);

  // Nettoyage
  onScopeDispose(() => clearInterval(interval));
});

scope.stop(); // Arrête le scope et le `setInterval`
```

06.2

Les Proxys dans Vue 3



Introduction aux Proxys dans Vue 3

Qu'est-ce qu'un Proxy ?

Un Proxy est une fonctionnalité native de JavaScript introduite avec ES6. Il permet de surveiller, d'intercepter et de personnaliser les opérations sur un objet ou une fonction. Dans Vue 3, les Proxys remplacent les objets définis avec `Object.defineProperty` pour le système de réactivité.

Points clés

Interception des Opérations : Accéder, modifier ou supprimer des propriétés.

Performance : Plus rapide et flexible que les getter/setter d'ES5.

Support des Structures Complexes : Gère nativement les tableaux et les objets imbriqués.

Comment Vue utilise les Proxys

Vue utilise les Proxys pour suivre les dépendances réactives et notifier les composants en cas de changement. Chaque objet réactif est enveloppé dans un Proxy qui :

Intercepte la lecture (get) pour suivre les dépendances.

Intercepte l'écriture (set) pour notifier les mises à jour.

Gère les structures imbriquées dynamiquement.

```
const target = { value: 42 };
const proxy = new Proxy(target, {
  get(target, key) {
    console.log(`Accès à la propriété "${key}"`);
    return target[key];
  },
  set(target, key, value) {
    console.log(`Modification de "${key}" : ${value}`);
    target[key] = value;
    return true;
  },
});

proxy.value; // Log: Accès à la propriété "value"
proxy.value = 100; // Log: Modification de "value" : 100
```

Proxys et Réactivité dans Vue

Vue enveloppe les objets dans un Proxy pour les rendre réactifs. Cela est réalisé avec les APIs `reactive()` et `readonly()`.

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

const state = reactive({
  count: 0,
  nested: {
    value: 42,
  },
});

state.count++; // Réactif
state.nested.value = 99; // Réactif
```

- Les Proxys permettent de détecter les changements en profondeur.
- Les tableaux et objets imbriqués sont automatiquement réactifs.

Manipulation des Proxys avec Vue

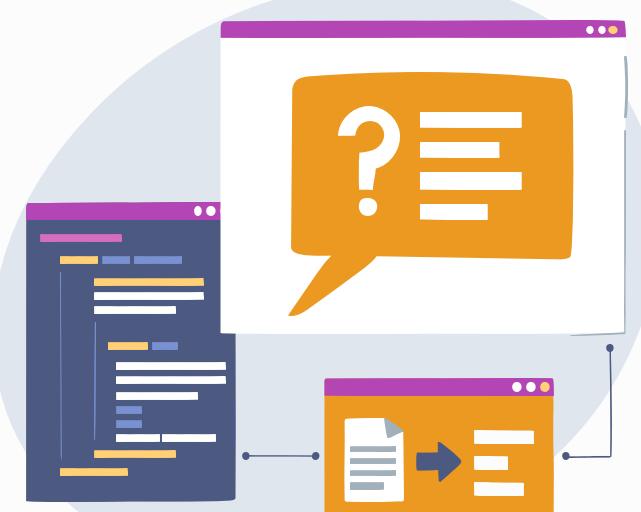
Vous pouvez utiliser des Proxys dans des cas avancés, comme exclure certaines propriétés de la réactivité ou transformer dynamiquement les valeurs.

```
const target = { sensitive: 'secret', public: 'data' };
const proxy = new Proxy(target, {
  get(target, key) {
    if (key === 'sensitive') {
      return '***';
    }
    return target[key];
  },
});

console.log(proxy.sensitive); // "***"
console.log(proxy.public); // "data"
```

07

Les évènements



Écoute des événements

Dans Vue.js, vous pouvez écouter les événements DOM en utilisant v-on ou la directive raccourcie @. Cela permet d'exécuter des méthodes spécifiques lorsque des événements se produisent.

```
<template>
  <div>
    <button @click="handleClick">Cliquez-moi</button>
  </div>
</template>

<script setup>
const handleClick = () => {
  console.log('Bouton cliqué');
};
</script>
```

Gestion des événements natifs

```
<!-- ParentComponent.vue -->
<template>
  <div>
    <ChildComponent @click.native="handleClick" />
  </div>
</template>

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

const handleClick = () => {
  console.log('Événement natif cliqué');
};
</script>

<!-- ChildComponent.vue -->
<template>
  <button>Cliquez-moi (enfant)</button>
</template>
```

Pour gérer les événements natifs dans des composants enfants, vous pouvez utiliser .native avec v-on ou @. Cela permet aux parents d'écouter des événements natifs sur les composants enfants.

Méthodes dans les templates

Vous pouvez appeler des méthodes directement depuis les templates pour gérer les événements. Cela permet une manipulation simple et efficace des interactions utilisateur.

```
<template>
  <div>
    <button @click="showMessage('Bonjour !')>Cliquez-moi</button>
  </div>
</template>

<script setup>
const showMessage = (msg) => {
  console.log(msg);
};
</script>
```

Modificateurs d'événements

Les modificateurs d'événements ajoutent des fonctionnalités supplémentaires aux écouteurs d'événements. Par exemple, .stop arrête la propagation de l'événement, et .prevent empêche le comportement par défaut.

```
<template>
  <div>
    <button @click.stop="handleClick">Cliquez-moi sans propagation</button>
    <form @submit.prevent="handleSubmit">
      <button type="submit">Soumettre</button>
    </form>
  </div>
</template>

<script setup>
const handleClick = () => {
  console.log('Propagation arrêtée');
};

const handleSubmit = () => {
  console.log('Soumission de formulaire empêchée');
};
</script>
```

Exemples pratiques d'écoute d'événements

```
<template>
<div>
  <input @input="handleInput" placeholder="Tapez ici" />
  <p>{{ inputText }}</p>
</div>
</template>

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

const inputText = ref('');

const handleInput = (event) => {
  inputText.value = event.target.value;
};
</script>
```

Les écouteurs d'événements peuvent être utilisés pour diverses interactions utilisateur, comme les clics de boutons, les mouvements de la souris, et les soumissions de formulaires.

08

Les formulaires



Binding de base avec v-model

v-model crée une liaison bidirectionnelle entre les données et les éléments de formulaire, synchronisant automatiquement les valeurs.

```
<template>
  <div>
    <input v-model="text" placeholder="Tapez quelque chose" />
    <p>{{ text }}</p>
  </div>
</template>

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

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

Formulaires avec input, textarea, select

v-model peut être utilisé avec divers éléments de formulaire, y compris les input, textarea et select.

```
<template>
  <div>
    <input v-model="name" placeholder="Nom" />
    <textarea v-model="message" placeholder="Message"></textarea>
    <select v-model="selected">
      <option>A</option>
      <option>B</option>
      <option>C</option>
    </select>
    <p>Nom: {{ name }}</p>
    <p>Message: {{ message }}</p>
    <p>Sélectionné: {{ selected }}</p>
  </div>
</template>

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

const name = ref('');
const message = ref('');
const selected = ref('A');

</script>
```

Modificateurs de v-model

Les modificateurs de v-model ajoutent des fonctionnalités supplémentaires, comme .lazy pour mettre à jour après le changement, .number pour convertir en nombre, et .trim pour enlever les espaces.

```
<template>
  <div>
    <input v-model.lazy="text" placeholder="Tapez et sortez" />
    <input v-model.number="age" type="number" placeholder="Age" />
    <input v-model.trim="name" placeholder="Nom" />
    <p>Texte: {{ text }}</p>
    <p>Age: {{ age }}</p>
    <p>Nom: {{ name }}</p>
  </div>
</template>

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

const text = ref('');
const age = ref(0);
const name = ref('');


```

Gestion des formulaires multiples

Vous pouvez gérer plusieurs formulaires et leurs données en utilisant v-model pour chaque champ et en regroupant les données de formulaire dans des objets réactifs.

```
<template>
  <div>
    <form @submit.prevent="submitForm1">
      <input v-model="form1.name" placeholder="Nom du formulaire 1" />
      <button type="submit">Soumettre Formulaire 1</button>
    </form>
    <form @submit.prevent="submitForm2">
      <input v-model="form2.email" placeholder="Email du formulaire 2" />
      <button type="submit">Soumettre Formulaire 2</button>
    </form>
  </div>
</template>

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

const form1 = ref({
  name: ''
});

const form2 = ref({
  email: ''
});

const submitForm1 = () => {
  console.log('Formulaire 1 soumis:', form1.value);
};

const submitForm2 = () => {
  console.log('Formulaire 2 soumis:', form2.value);
};
</script>
```

Validation de formulaires

La validation des formulaires peut être effectuée en ajoutant des règles de validation personnalisées et en fournissant des messages d'erreur en fonction de l'état du formulaire.

```
<template>
  <div>
    <form @submit.prevent="handleSubmit">
      <input v-model="email" placeholder="Email" @input="validateEmail" />
      <span v-if="emailError">{{ emailError }}</span>
      <button type="submit" :disabled="emailError">Soumettre</button>
    </form>
  </div>
</template>

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

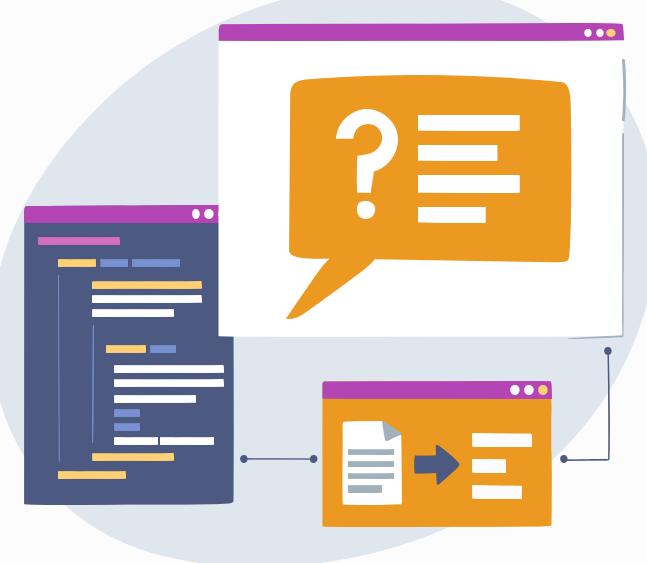
const email = ref('');
const emailError = ref('');

const validateEmail = () => {
  emailError.value = email.value.includes('@') ? '' : 'Email invalide';
};

const handleSubmit = () => {
  if (!emailError.value) {
    console.log('Formulaire soumis avec:', email.value);
  }
};
</script>
```

08.1

Utilisation d'un package de formulaire : Formkit



Introduction à FormKit

FormKit est une bibliothèque puissante pour la gestion des formulaires dans Vue 3. Elle offre une API intuitive, des champs dynamiques, une validation en temps réel, et une prise en charge étendue des formulaires complexes.

```
<script setup>
import { plugin as FormKitPlugin } from '@formkit/vue';
import '@formkit/themes/genesis';

app.use(FormKitPlugin);
</script>

<template>
  <FormKit type="text" name="email" label="Email" />
</template>
```

Installation et Configuration

- Installez FormKit avec npm ou yarn et configurez-le dans votre projet Vue 3.

```
npm install @formkit/vue
```

```
// main.js
import { createApp } from 'vue';
import App from './App.vue';
import { plugin as FormKitPlugin } from '@formkit/vue';

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

Création d'un Champ de Texte

- FormKit permet de créer facilement des champs de texte avec des labels et des placeholders.

```
<template>
  <FormKit type="text" name="username" label="Nom d'utilisateur" placeholder="Entrez
</template>
```

Champs Dynamiques

- Vous pouvez générer des champs dynamiques en utilisant des boucles dans Vue 3.

```
<template>
  <div v-for="(field, index) in fields" :key="index">
    <FormKit :type="field.type" :name="field.name" :label="field.label" />
  </div>
</template>

<script setup>
const fields = [
  { type: 'text', name: 'firstName', label: 'Prénom' },
  { type: 'text', name: 'lastName', label: 'Nom' },
];
</script>
```

Validation Basique

- Ajoutez des règles de validation directement dans vos champs.

```
<template>
  <FormKit
    type="text"
    name="email"
    label="Email"
    validation="required|email"
    validation-label="Adresse e-mail"
  />
</template>
```

Validation Personnalisée

- Créez des règles de validation personnalisées pour des besoins spécifiques.

```
// main.js
import { createValidationPlugin } from '@formkit/validation';

const customRules = {
  startsWithA: ({ value }) => value.startsWith('A') || 'Doit commencer par un A',
};

app.use(FormKitPlugin, {
  plugins: [createValidationPlugin(customRules)],
});
```

```
<template>
<FormKit
  type="text"
  name="customField"
  label="Commence par A"
  validation="startsWithA"
/>
</template>
```

Groupes de Champs

- Regroupez plusieurs champs dans une structure logique.

```
<template>
  <FormKit type="group" name="adresse">
    <FormKit type="text" name="rue" label="Rue" />
    <FormKit type="text" name="ville" label="Ville" />
  </FormKit>
</template>
```

Utilisation des Slots

- Personnalisez vos champs en utilisant des slots.

```
<template>
  <FormKit type="text" name="custom">
    <template #label>Label personnalisé</template>
  </FormKit>
</template>
```

Champs Conditionnels

- Affichez ou masquez des champs en fonction de conditions.

```
<template>
  <FormKit type="checkbox" name="subscribe" label="S'abonner à la newsletter" v-model="subscribe" />
  <FormKit v-if="subscribe" type="email" name="email" label="Votre email" />
</template>

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

const subscribe = ref(false);
</script>
```

Composants Personnalisés

- Créez vos propres composants en utilisant les hooks FormKit.

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

const CustomInput = defineComponent({
  props: ['label', 'name'],
  template: `<label>{{ label }}<input :name="name" /></label>`,
});

export default CustomInput;
</script>
```

Soumission du Formulaire

- Récupérez les données du formulaire avec une gestion des événements simple.

```
<template>
  <form @submit.prevent="handleSubmit">
    <FormKit type="text" name="username" label="Nom d'utilisateur" />
    <button type="submit">Envoyer</button>
  </form>
</template>

<script setup>
const handleSubmit = () => {
  console.log('Formulaire soumis !');
};
</script>
```

Gestion des Messages d'Erreur

- Affichez des messages d'erreur personnalisés pour les champs invalides.

```
<template>
  <FormKit
    type="text"
    name="email"
    label="Email"
    validation="required|email"
    validation-label="Adresse e-mail"
  />
</template>
```

Champs Répétables

- Ajoutez des champs dynamiquement en utilisant les listes répétables.

```
<template>
  <FormKit type="list" name="tâches" label="Tâches">
    <FormKit type="text" name="tâche" label="Nom de la tâche" />
  </FormKit>
</template>
```

Gestion des États

- Les champs FormKit peuvent avoir des états tels que "disabled" ou "readonly".

```
<template>
  <FormKit type="text" name="email" label="Email" :disabled="true" />
</template>
```

Styles Personnalisés

- Personnalisez les styles avec vos propres classes CSS.

```
<template>
  <FormKit type="text" name="username" label="Nom d'utilisateur" input-class="mon-style"
</template>
```

Thèmes Globaux

- Appliquez un thème à tous les composants FormKit.

```
import '@formkit/themes/genesis';
```

Champs Multilingues

- Configurez FormKit pour supporter plusieurs langues.

```
app.use(FormKitPlugin, {  
    config: { locales: { fr, en }, locale: 'fr' },  
});
```

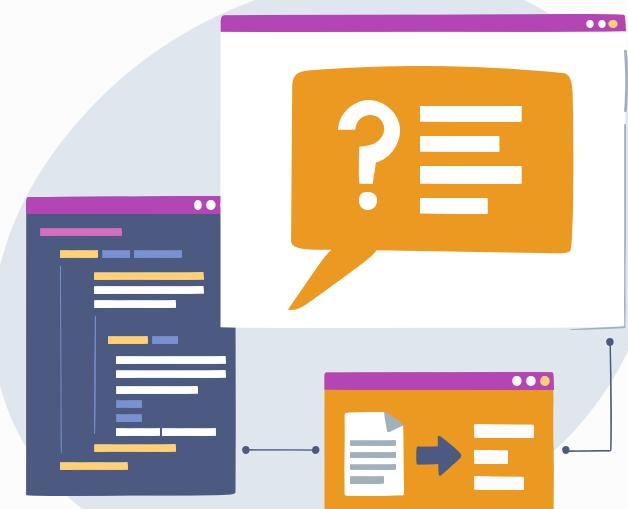
Validation au Niveau du Groupe

- Appliquez des règles de validation à un groupe entier.

```
<template>
  <FormKit type="group" name="coordonnees" validation="required">
    <FormKit type="text" name="email" label="Email" />
    <FormKit type="text" name="telephone" label="Téléphone" />
  </FormKit>
</template>
```

09

Les cycles de vie



Introduction aux cycles de vie

Les hooks de cycle de vie dans Vue.js permettent d'exécuter du code à différents moments du cycle de vie d'un composant. Cela inclut des moments comme la création, le montage, la mise à jour et la destruction du composant.

beforeCreate

Quand ? Juste après l'instanciation du composant.

Usage : Initialisation de très bas niveau (avant que les données réactives soient configurées).

Exemple concret : Si vous voulez exécuter un log pour tracer les instanciations.

```
export default {
  beforeCreate() {
    console.log("Composant en cours d'instanciation !");
  }
}
```

created()

Le hook created() est exécuté après l'initialisation de l'instance du composant, mais avant que celui-ci soit monté dans le DOM. Il est utilisé pour configurer les données réactives, les appels API, etc.

```
<template>
  <div>
    <p>{{ message }}</p>
  </div>
</template>

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

const message = ref('Initialisation...');

onBeforeMount(() => {
  message.value = 'Composant créé!';
  console.log('Composant créé');
});
</script>
```

beforeMount

Quand ? Avant que le composant soit inséré dans le DOM.

Usage : Rarement utilisé. Permet de modifier la configuration avant que Vue ne commence à travailler avec le DOM.

```
export default {
  beforeMount() {
    console.log("Avant le montage du DOM !");
  }
}
```

mounted()

Le hook mounted() est appelé après que le composant a été monté dans le DOM. Il est souvent utilisé pour manipuler le DOM ou effectuer des opérations nécessitant que le composant soit affiché.

```
<template>
  <div ref="divElement">
    <p>{{ message }}</p>
  </div>
</template>

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

const message = ref('Montage en cours...');
const divElement = ref(null);

onMounted(() => {
  message.value = 'Composant monté!';
  console.log('Composant monté');
  divElement.value.style.color = 'blue';
});
</script>
```

beforeUpdate

Quand ? Avant que les modifications du DOM liées aux données réactives soient appliquées.

Usage : Permet d'inspecter les données actuelles avant la mise à jour.

Exemple concret : Suivre des changements avant leur application au DOM.

```
export default {
  data() {
    return { counter: 0 };
  },
  beforeUpdate() {
    console.log("Avant mise à jour : ", this.counter);
  },
  methods: {
    increment() {
      this.counter++;
    }
  }
}
```

updated

Le hook `updated()` est appelé après qu'une mise à jour a été effectuée sur le composant. Il est utilisé pour répondre aux changements de données réactives ou de props.

```
<template>
  <div>
    <p>{{ message }}</p>
    <button @click="updateMessage">Mettre à jour le message</button>
  </div>
</template>

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

const message = ref('Message initial');

const updateMessage = () => {
  message.value = 'Message mis à jour!';
};

onUpdated(() => {
  console.log('Composant mis à jour avec le message:', message.value);
});
```

beforeUnmount

Quand ? Juste avant que le composant soit retiré du DOM.

Usage : Nettoyer les ressources comme les timers ou les écouteurs.

Exemple concret : Nettoyer un setInterval.

```
export default {
  beforeUnmount() {
    clearInterval(this.interval);
  },
  mounted() {
    this.interval = setInterval(() => {
      console.log("Interval actif");
    }, 1000);
  }
}
```

unmounted

Quand ? Après que le composant a été retiré du DOM.

Usage : Dernière étape pour effectuer du nettoyage ou des logs.

Exemple concret : Détruire des objets liés à une bibliothèque.

```
export default {
  unmounted() {
    console.log("Composant détruit !");
  }
}
```

09.1

Les cycles de vie avancés



onErrorCaptured : Capture des Erreurs

Capture les erreurs dans l'arbre descendant des composants.

Utile pour le reporting ou les fallback UI.

```
import { defineComponent, onErrorCaptured } from 'vue';

export default defineComponent({
  setup() {
    onErrorCaptured((err, instance, info) => {
      console.error('Erreur capturée :', err, info);
      return false; // Empêche la propagation
    });
  },
});
```

onRenderTracked : Suivi du Rendu

Appelé lorsque des dépendances réactives sont suivies lors du rendu.

```
import { defineComponent, onRenderTracked } from 'vue';

export default defineComponent({
  setup() {
    onRenderTracked((event) => {
      console.log('Rendu suivi :', event);
    });
  },
});
```

onRenderTriggered : Déclenchement du Rendu

Appelé lorsqu'un rendu est déclenché par un changement de dépendance.

```
import { defineComponent, onRenderTriggered } from 'vue';

export default defineComponent({
  setup() {
    onRenderTriggered((event) => {
      console.log('Rendu déclenché :', event);
    });
  },
});
```

onActivated et onDeactivated

onActivated : Appelé lorsqu'un composant est activé (par exemple, dans des routes ou des onglets dynamiques).

onDeactivated : Appelé lorsqu'un composant est désactivé,

```
import { defineComponent, onActivated, onDeactivated } from 'vue';

export default defineComponent({
  setup() {
    onActivated(() => {
      console.log('Composant activé.');
    });

    onDeactivated(() => {
      console.log('Composant désactivé.');
    });
  },
});
```

onServerPrefetch : Préchargement pour SSR

Spécifique au rendu côté serveur (SSR).

Utilisé pour précharger les données avant le rendu.

```
import { defineComponent, onServerPrefetch } from 'vue';

export default defineComponent({
  setup() {
    onServerPrefetch(async () => {
      const data = await fetchData();
      console.log('Données préchargées :', data);
    });
  },
});
```

10

Rendus de listes



List Rendering

Le rendu de liste permet d'itérer sur des collections de données et de générer des éléments pour chaque élément de la collection.

```
<template>
  <ul>
    <li v-for="item in items" :key="item.id">{{ item.text }}</li>
  </ul>
</template>

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

const items = ref([
  { id: 1, text: 'Item 1' },
  { id: 2, text: 'Item 2' },
  { id: 3, text: 'Item 3' }
]);
</script>
```

Rendu de listes avec v-for

v-for permet de rendre une liste d'éléments en itérant sur une collection.

```
<template>
  <div>
    <p v-for="n in 10" :key="n">Numéro {{ n }}</p>
  </div>
</template>
```

Les clés uniques avec v-bind

Utiliser v-bind:key pour fournir des clés uniques est essentiel pour que Vue suive les éléments correctement.

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

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

const items = ref([
  { id: 1, name: 'Article 1' },
  { id: 2, name: 'Article 2' }
]);
</script>
```

Index dans v-for

L'index de l'itération dans un v-for peut être utilisé pour identifier la position de l'élément dans la liste.

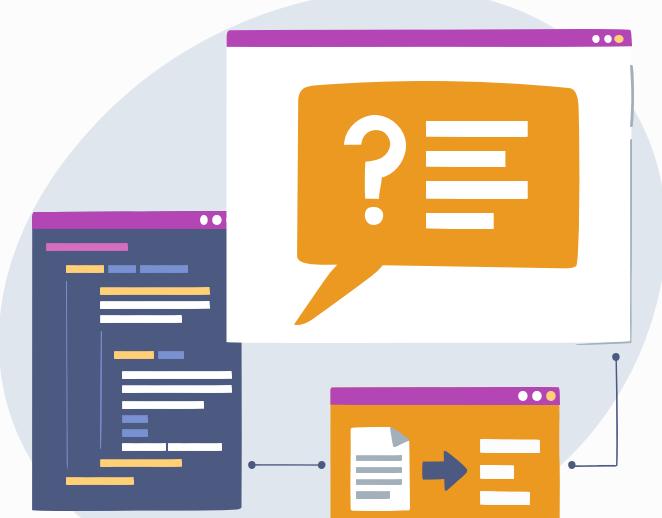
```
<template>
  <ul>
    <li v-for="(item, index) in items" :key="item.id">
      {{ index + 1 }}. {{ item.name }}
    </li>
  </ul>
</template>

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

const items = ref([
  { id: 1, name: 'Element 1' },
  { id: 2, name: 'Element 2' }
]);
</script>
```

10

Rendu Conditionnel



Rendu conditionnel avec v-if

v-if permet de conditionner le rendu d'un élément.

```
<template>
  <div v-if="show">
    Ceci est visible
  </div>
</template>

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

const show = ref(true);
</script>
```

v-else et v-else-if

v-else et v-else-if permettent de gérer plusieurs conditions dans le rendu d'un élément.

```
<template>
  <div v-if="type === 'A'">
    Type A
  </div>
  <div v-else-if="type === 'B'">
    Type B
  </div>
  <div v-else>
    Autre type
  </div>
</template>

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

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

Utiliser v-show

v-show permet d'afficher ou de masquer un élément via CSS sans détruire et recréer l'élément dans le DOM.

```
<template>
  <div v-show="isVisible">
    Ceci est visible avec v-show
  </div>
</template>

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

const isVisible = ref(true);
</script>
```

Différences entre v-if et v-show

v-if ajoute ou retire un élément du DOM, tandis que v-show modifie la propriété display de l'élément pour le masquer ou l'afficher.

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

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

const show = ref(true);
</script>
```

Binding de classes

v-bind:class permet de lier dynamiquement des classes CSS à un élément.

```
<template>
  <div :class="classObject">
    Classe Dynamique
  </div>
</template>

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

const classObject = ref({
  active: true,
  'text-danger': false
});
</script>
```

Classes conditionnelles

Vous pouvez utiliser des expressions pour appliquer des classes de manière conditionnelle.

```
<template>
  <div :class="{ active: isActive, 'text-danger': hasError }">
    Classe Conditionnelle
  </div>
</template>

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

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

Binding de styles

v-bind:style permet de lier dynamiquement des styles en ligne à un élément.

```
<template>
  <div :style="styleObject">
    Style Dynamique
  </div>
</template>

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

const styleObject = ref({
  color: 'blue',
  fontSize: '14px'
});
</script>
```

Styles dynamiques

Les styles peuvent être définis de manière dynamique en fonction des données réactives.

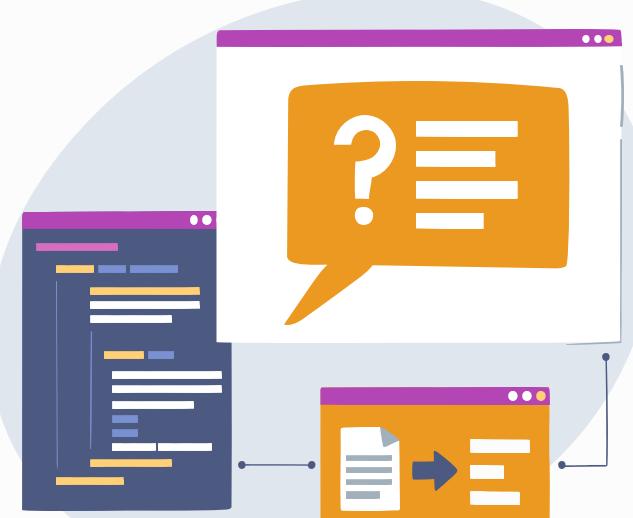
```
<template>
  <div :style="{ color: isActive ? 'green' : 'red' }">
    Style Dynamique Conditionnel
  </div>
</template>

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

const isActive = ref(true);
</script>
```

11

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);
  }
});
```

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>
```

11.1

Vue Router avancé



Personnaliser l'affichage avec RouterView et son slot

RouterView est un composant clé de Vue Router qui affiche le composant correspondant à la route active. Avec la Composition API, vous pouvez exploiter son slot pour personnaliser ou étendre le rendu. Cela permet, par exemple, d'ajouter des animations ou des wrappers spécifiques autour de vos composants de page.

```
<template>
  <div>
    <RouterView v-slot="{ Component }">
      <transition name="fade">
        <component :is="Component" />
      </transition>
    </RouterView>
  </div>
</template>

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

export default defineComponent({
  name: 'App',
});
</script>

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

Scroll Behavior

Avec Vue Router, vous pouvez contrôler le comportement de défilement (scrolling) lors des transitions entre les routes grâce à l'option scrollBehavior. Cela permet de personnaliser la position de défilement, par exemple pour toujours revenir en haut de la page ou restaurer la position précédente.

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

const routes = [
  { path: '/', component: () => import('./Home.vue') },
  { path: '/about', component: () => import('./About.vue') },
];

const router = createRouter({
  history: createWebHistory(),
  routes,
  scrollBehavior(to, from, savedPosition) {
    if (savedPosition) {
      return savedPosition; // Restaure la position précédente
    } else {
      return { top: 0 }; // Défile tout en haut
    }
  },
});

export default router;
```

Lazy Loading des routes

Le Lazy Loading (chargement à la demande) optimise les performances en chargeant les composants des routes uniquement lorsque cela est nécessaire. Ceci réduit le poids initial de l'application et améliore le temps de chargement.

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

const routes = [
  {
    path: '/',
    component: () => import('./views/Home.vue'), // Chargement différé
  },
  {
    path: '/about',
    component: () => import('./views/About.vue'), // Chargement différé
  },
];

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

export default router;
```

12

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>
```

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>
```

Watcher avec Options

Les watchers supportent des options comme `immediate` (exécuter dès l'initialisation) et `deep` (surveiller les changements dans des objets imbriqués).

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

const utilisateur = reactive({
  nom: 'Jean',
  details: { age: 30 }
});

watch(
  () => utilisateur.details,
  (nouveau, ancien) => {
    console.log('Détails modifiés', ancien, nouveau);
  },
  { deep: true }
);
</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>
```

Watchers avec Destruction Automatique

Les watchers créés dans setup sont automatiquement détruits lorsque le composant est démonté, évitant ainsi les fuites de mémoire.

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

const compteur = ref(0);

watch(compteur, () => {
  console.log('Compteur changé');
});
</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>
```

Différence entre watch et watchEffect

watch surveille explicitement des variables.

watchEffect détecte automatiquement toutes les dépendances réactives utilisées.

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

const compteur = ref(0);

// Watch explicite
watch(compteur, (val) => console.log('watch:', val));

// WatchEffect implicite
watchEffect(() => console.log('watchEffect:', compteur.value));
</script>
```

Watchers sur des Collections

Pour surveiller des tableaux ou des objets imbriqués, utilisez l'option `deep: true`.

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

const liste = ref(['item1', 'item2']);

watch(
  liste,
  () => {
    console.log('Liste modifiée:', liste.value);
  },
  { deep: true }
);
</script>
```

Watcher avec Fonction de Nettoyage

- Un watcher peut inclure une fonction de nettoyage pour annuler ou réinitialiser des effets secondaires avant la prochaine exécution.

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

const compteur = ref(0);

watch(compteur, (nouveau, ancien, onCleanup) => {
    const timer = setTimeout(() => console.log('Compteur:', nouveau), 1000);

    onCleanup(() => {
        clearTimeout(timer);
        console.log('Timer réinitialisé');
    });
});

</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>
```

Observer Plusieurs Sources

- Les watchers peuvent surveiller plusieurs sources réactives en utilisant un tableau.

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

const x = ref(0);
const y = ref(0);

watch([x, y], ([nouveauX, nouveauY]) => {
  console.log(`Coordonnées : (${nouveauX}, ${nouveauY})`);
});
</script>
```

Gestion des Erreurs dans un Watch

- Les erreurs dans un watcher sont capturées et affichées dans la console. Vous pouvez aussi les gérer explicitement.

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

const compteur = ref(0);

watch(
  compteur,
  () => {
    throw new Error('Erreur dans le watcher');
  },
  { deep: true }
);
</script>
```

Désactiver un Watcher Manuellement

- Les watchers retournent une fonction de destruction que vous pouvez appeler pour désactiver manuellement le watcher.

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

const compteur = ref(0);

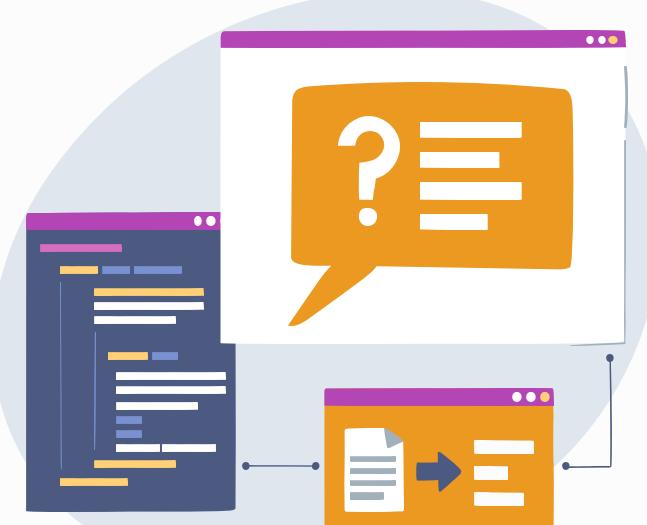
const stopWatcher = watch(compteur, (val) => {
    console.log('Compteur:', val);
});

// Désactiver le watcher après 5 secondes
setTimeout(() => stopWatcher(), 5000);
</script>
```

13

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>
```

14

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.

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

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

</script>

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

```
<!-- 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>
```

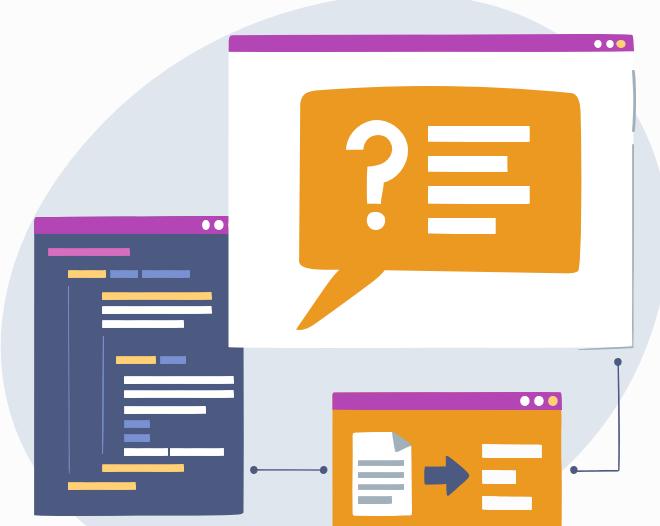
```
<template>
  <div>
    <ChildComponent v-model="parentValue" />
    <p>Valeur dans le composant parent: {{ parentValue }}</p>
  </div>
</template>

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

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

15

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' }
  });

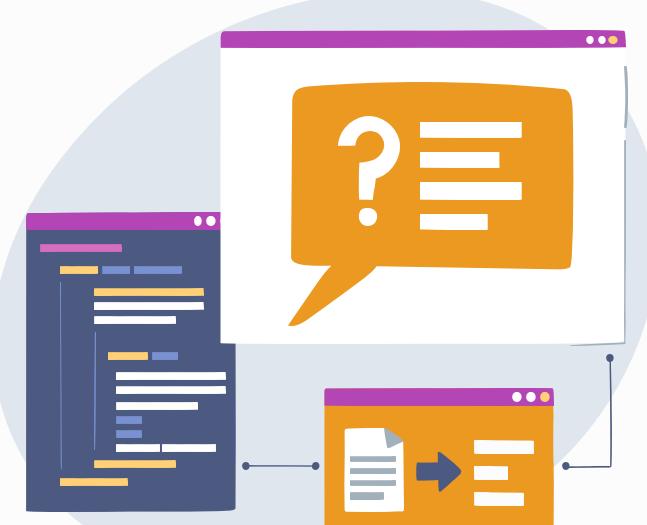
  const setUser = (name) => {
    state.value.user.name = name;
  };

  return { state, setUser };
}
```

16

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');
```

17

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.

```
<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>
```

```
<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({});
```

```
// 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();
  }
};
```

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,
};
```

18

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>
```

18.1

Pinia avancé



Introduction aux Plugins Pinia

Les plugins Pinia permettent d'étendre les fonctionnalités de Pinia en ajoutant des comportements globaux ou spécifiques aux stores. Ils interviennent lors de la création d'un store et peuvent modifier son comportement ou ajouter des propriétés personnalisées.

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

const pinia = createPinia();

// Exemple de plugin simple
pinia.use(({ store }) => {
  store.$greet = () => console.log(`Hello depuis le store ${store.$id}`);
});

export default pinia;
```

Ajouter une Propriété Globale

- Un plugin peut ajouter des propriétés ou méthodes personnalisées accessibles dans tous les stores.

```
function globalPropertyPlugin({ store }) {
  store.$timestamp = () => Date.now();
}

const pinia = createPinia();
pinia.use(globalPropertyPlugin);

export default pinia;

// Utilisation dans un composant
store.$timestamp(); // Renvoie le timestamp actuel
```

Persistance des Données avec localStorage

Les plugins permettent de sauvegarder automatiquement les données du store dans localStorage pour conserver l'état entre les rechargements de page.

```
function persistPlugin({ store }) {
  const savedState = localStorage.getItem(store.$id);
  if (savedState) {
    store.$patch(JSON.parse(savedState));
  }

  store.$subscribe((mutation, state) => {
    localStorage.setItem(store.$id, JSON.stringify(state));
  });
}

const pinia = createPinia();
pinia.use(persistPlugin);
```

Logger des Actions

Un plugin peut capturer et enregistrer toutes les actions déclenchées dans un store pour un suivi ou un débogage facile.

```
function loggerPlugin({ store }) {
  store.$onAction(({ name, args }) => {
    console.log(`Action déclenchée: ${name}`, args);
  });
}

const pinia = createPinia();
pinia.use(loggerPlugin);
```

Utilisation d'API Externe

Un plugin peut ajouter des méthodes permettant de simplifier l'intégration avec des services externes, comme des APIs.

```
function apiPlugin({ store }) {
  store.$fetchData = async (url) => {
    const response = await fetch(url);
    return response.json();
  };
}

const pinia = createPinia();
pinia.use(apiPlugin);

// Exemple d'utilisation dans un composant
store.$fetchData('https://api.example.com/data')
  .then(data => console.log(data));
```

Gestion des Erreurs

- Les plugins peuvent centraliser la gestion des erreurs des stores, notamment dans les actions.

```
function errorHandlerPlugin({ store }) {
  store.$onAction(({ name, args, after, onError }) => {
    onError((error) => {
      console.error(`Erreur dans l'action ${name}:`, error);
    });
  });
}

const pinia = createPinia();
pinia.use(errorHandlerPlugin);
```

Initialisation Dynamique

Un plugin peut être utilisé pour initialiser un store de manière dynamique en fonction de données externes.

```
function dynamicInitPlugin({ store }) {
  store.$initialize = async (url) => {
    const data = await fetch(url).then(res => res.json());
    store.$patch(data);
  };
}

const pinia = createPinia();
pinia.use(dynamicInitPlugin);

// Utilisation dans un store
store.$initialize('/api/storeData');
```

Ajout de Méthodes Utilitaires

Ajoutez des méthodes utilitaires aux stores pour des cas spécifiques, comme le nettoyage ou le reset de l'état.

```
function utilityPlugin({ store }) {
  store.$reset = () => {
    store.$patch(store.$state.$original);
  };
}

const pinia = createPinia();
pinia.use(utilityPlugin);

// Utilisation dans un composant
store.$reset();
```

Gestion des Permissions

Un plugin peut intégrer des règles de permissions pour sécuriser l'accès à certaines actions ou données.

```
function permissionPlugin({ store }) {
  store.$can = (permission) => {
    // Exemple simple : vérifier des permissions stockées
    return store.permissions.includes(permission);
  };
}

const pinia = createPinia();
pinia.use(permissionPlugin);

// Utilisation
if (store.$can('edit')) {
  console.log('Permission accordée');
}
```

Plugin Combiné

Combinez plusieurs comportements dans un seul plugin pour des fonctionnalités avancées, comme la persistance et le logging.

```
function combinedPlugin({ store }) {
  // Persistance
  const savedState = localStorage.getItem(store.$id);
  if (savedState) {
    store.$patch(JSON.parse(savedState));
  }

  store.$subscribe((mutation, state) => {
    localStorage.setItem(store.$id, JSON.stringify(state));
  });

  // Logging
  store.$onAction(({ name, args }) => {
    console.log(`Action: ${name}`, args);
  });
}

const pinia = createPinia();
pinia.use(combinedPlugin);
```

18.2

Pinia Colada



Introduction à Pinia Colada

Pinia Colada est une extension pour Pinia, conçue pour gérer les états asynchrones dans les applications Vue.js. Elle simplifie les requêtes de données, introduit la mise en cache, la déduplication et l'invalidation, tout en éliminant les tâches répétitives liées à la gestion d'états complexes.

```
import { createApp } from 'vue'
import { createPinia } from 'pinia'
import { PiniaColada } from '@pinia/colada'

const app = createApp(App)
app.use(createPinia())
app.use(PiniaColada)

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

Requêtes avec useQuery

useQuery permet de déclarer des requêtes de données et de gérer automatiquement leur état (en cours, réussi ou en erreur). Les données sont mises en cache et partagées à travers l'application.

```
<script setup>
import { useQuery } from '@pinia/colada'
import { getAllProducts } from '@/api/products'

const { state, asyncStatus } = useQuery({
  key: ['products'],
  query: getAllProducts,
})
</script>

<template>
  <div v-if="asyncStatus === 'loading'">Chargement...</div>
  <ul v-else-if="state.data">
    <li v-for="product in state.data" :key="product.id">{{ product.name }}</li>
  </ul>
  <div v-else>Erreur : {{ state.error }}</div>
</template>
```

Cache des Requêtes

- Le cache des requêtes permet de stocker les résultats et de réutiliser les données sans effectuer à nouveau la requête. Cela améliore les performances et réduit les appels réseau inutiles.

```
<script setup>
import { useQuery } from '@pinia/colada'

const { state, refresh } = useQuery({
  key: ['todos'],
  query: () => fetch('/api/todos').then(res => res.json()),
})
</script>

<template>
  <button @click="refresh">Actualiser les données</button>
  <ul v-if="state.data">
    <li v-for="todo in state.data" :key="todo.id">{{ todo.text }}</li>
  </ul>
</template>
```

Mutations avec useMutation

Les mutations permettent de modifier les données sur le serveur et de notifier les requêtes liées pour qu'elles se mettent à jour automatiquement.

```
<script setup>
import { useMutation } from '@pinia/colada'
import { patchContact } from '@/api/contacts'

const { mutate, asyncStatus } = useMutation({
  mutation: patchContact,
})
</script>

<template>
  <button @click="mutate(contact)" :disabled="asyncStatus === 'loading'">
    Mettre à jour
  </button>
</template>
```

Invalidation des Requêtes

- L'invalidation des requêtes force le rafraîchissement des données mises en cache après une mutation.

```
<script setup>
import { useMutation, useQueryCache } from '@pinia/colada'
import { patchContact } from '@/api/contacts'

const queryCache = useQueryCache()

const { mutate } = useMutation({
  mutation: patchContact,
  onSettled: () => {
    queryCache.invalidateQueries({ key: ['contacts'] })
  },
})
</script>
```

Optimistic Updates

- Les mises à jour optimistes mettent à jour l'interface utilisateur avant que la mutation ne soit confirmée, offrant une expérience plus fluide.

```
<script setup>
import { useMutation, useQueryCache } from '@pinia/colada'
import { patchContact } from '@/api/contacts'

const queryCache = useQueryCache()

const { mutate } = useMutation({
  mutation: patchContact,
  onMutate: (updatedContact) => {
    const previousData = queryCache.getQueryData(['contacts', updatedContact.id])
    queryCache.setQueryData(['contacts', updatedContact.id], updatedContact)
    return { previousData }
  },
  onError: (err, updatedContact, context) => {
    queryCache.setQueryData(['contacts', updatedContact.id], context.previousData)
  },
})
</script>
```

Utilisation des Clés de Requête

- Les clés de requête permettent d'identifier les requêtes dans le cache et de les organiser en hiérarchies pour une invalidation efficace.

```
<script setup>
import { useQuery } from '@pinia/colada'

const { state } = useQuery({
  key: ['contacts', { page: 1 }],
  query: () => fetch('/api/contacts?page=1').then(res => res.json()),
})
</script>
```

Pagination avec Pinia Colada

- Pinia Colada prend en charge les requêtes paginées en utilisant des clés réactives basées sur les paramètres de la route ou les états locaux.

```
<script setup>
import { useRoute } from 'vue-router'
import { useQuery } from '@pinia,colada'

const route = useRoute()

const { state } = useQuery({
  key: () => ['contacts', Number(route.query.page) || 1],
  query: () => fetch(`/api/contacts?page=${Number(route.query.page)}`).then(res => res.json)
})
</script>
```

Gestion des Erreurs

Pinia Colada fournit des mécanismes pour gérer les erreurs dans les requêtes et mutations, avec des états comme `error` et `asyncStatus`.

```
<script setup>
import { useQuery } from '@pinia/colada'

const { state } = useQuery({
  key: ['products'],
  query: () => fetch('/api/products').then(res => {
    if (!res.ok) throw new Error('Erreur de chargement')
    return res.json()
  }),
})
</script>

<template>
  <div v-if="state.error">Erreur : {{ state.error.message }}</div>
</template>
```

Pinia Colada et Nuxt

Pinia Colada s'intègre parfaitement avec Nuxt pour gérer l'état asynchrone côté serveur, en utilisant useState ou des magasins Pinia.

```
<script setup>
  import { defineQuery } from '@pinia/colada'
  import { useState } from '#app'

  const search = useState('search', () => '')

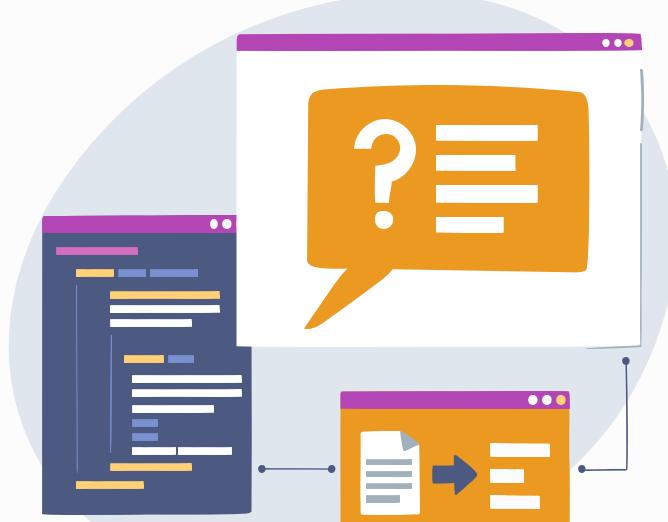
  const useFilteredTodos = defineQuery(() => ({
    key: ['todos', search.value],
    query: () => fetch(`/api/todos?search=${search.value}`).then(res => res.json()),
  }))
</script>
```

Plugin Combiné

Combinez plusieurs comportements dans un seul plugin pour des fonctionnalités avancées, comme la persistance et le logging.

19

Installation de Tanstack Query



Pourquoi Utiliser Tanstack Query ?

Tanstack Query (anciennement React Query) est une bibliothèque puissante pour la gestion des requêtes de données dans vos applications Vue.js. Elle simplifie le traitement des requêtes asynchrones, la mise en cache des résultats, et la gestion des états de chargement et d'erreur, vous permettant de vous concentrer sur l'expérience utilisateur.

Installation et Configuration de Tanstack Query

Pour utiliser Tanstack Query dans votre projet Vue 3, commencez par l'installer via npm ou yarn.

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

```
import { VueQueryPlugin } from '@tanstack/vue-query'  
  
app.use(VueQueryPlugin)
```

Les Bases des Hooks de Tanstack Query

Les hooks de Tanstack Query comme `useQuery` et `useMutation` facilitent la gestion des requêtes et des mutations de données dans vos composants Vue 3, tout en intégrant la gestion des états de chargement et d'erreur.

```
<script setup>
import { useQuery } from '@tanstack/vue-query'

const { isPending, isError, data, error } = useQuery({
  queryKey: ['users'],
  queryFn: fetchUsers,
})
</script>

<template>
  <div v-if="isPending">Loading...</div>
  <div v-else-if="isError">Error: {{ error.message }}</div>
  <ul v-else>
    <li v-for="user in data" :key="user.id">{{ user.name }}</li>
  </ul>
</template>
```

Utilisation de useQuery pour les Requêtes de Données

useQuery est un hook puissant pour effectuer des requêtes de données. Il prend en charge la mise en cache, le refetching, et la gestion des états de chargement et d'erreur.

```
<script setup>
import { useQuery } from '@tanstack/vue-query'

const { isPending, isError, data, error } = useQuery({
  queryKey: ['posts'],
  queryFn: fetchPosts,
})
</script>

<template>
  <div v-if="isPending">Loading...</div>
  <div v-else-if="isError">Error: {{ error.message }}</div>
  <ul v-else>
    <li v-for="post in data" :key="post.id">{{ post.title }}</li>
  </ul>
</template>
```

Gestion des Etats de Chargement et d'Erreur

Tanstack Query gère automatiquement les états de chargement et d'erreur pour vous, vous permettant de fournir une meilleure expérience utilisateur sans code supplémentaire complexe.

```
<script setup>
import { useQuery } from '@tanstack/vue-query'

const { isPending, isError, data, error } = useQuery({
  queryKey: ['comments'],
  queryFn: fetchComments,
})
</script>

<template>
  <div v-if="isPending">Loading...</div>
  <div v-else-if="isError">Error: {{ error.message }}</div>
  <ul v-else>
    <li v-for="comment in data" :key="comment.id">{{ comment.body }}</li>
  </ul>
</template>
```

Paramétrage des Requêtes avec useQuery

useQuery permet de personnaliser les requêtes via des options telles que queryKey, queryFn, et config pour des comportements comme le refetching et la mise en cache.

```
<script setup>
import { useQuery } from '@tanstack/vue-query'

const fetcher = (key, id) => fetch(`/api/data/${id}`).then(res => res.json())

const { isPending, isError, data, error } = useQuery({
  queryKey: ['data', 1],
  queryFn: () => fetcher(1),
})
</script>

<template>
  <div v-if="isPending">Loading...</div>
  <div v-else-if="isError">Error: {{ error.message }}</div>
  <div v-else>{{ data }}</div>
</template>
```

Référencement de Données et Cache dans Tanstack Query

Tanstack Query maintient un cache des données pour améliorer les performances des applications. Les données référencées peuvent être utilisées dans différents composants sans recharger les données.

```
<script setup>
import { useQuery } from '@tanstack/vue-query'

const fetchData = () => fetch('/api/data').then(res => res.json())

const { data } = useQuery({
  queryKey: ['data'],
  queryFn: fetchData,
})
</script>

<template>
  <div>{{ data }}</div>
</template>
```

Introduction à useMutation pour les Opérations d'Écriture

useMutation est utilisé pour les opérations d'écriture comme les créations, mises à jour ou suppressions de données. Il gère les états de succès et d'erreur pour ces opérations.

```
<script setup>
import { useMutation } from '@tanstack/vue-query'
import axios from 'axios'

const { error, mutate, reset } = useMutation({
  mutationFn: (newTodo) => axios.post('/todos', newTodo),
})

function addTodo() {
  mutate({ id: new Date(), title: 'Do Laundry' })
}
</script>

<template>
  <span v-if="error">
    An error occurred: {{ error.message }}
    <button @click="reset">Reset error</button>
  </span>
  <button @click="addTodo">Create Todo</button>
</template>
```

Gestion Optimiste des Mutations

La gestion optimiste permet de mettre à jour l'interface utilisateur immédiatement après une mutation, avant que la réponse serveur ne soit reçue, offrant une expérience utilisateur plus réactive.

```
<script setup>
import { useMutation, useQueryClient } from '@tanstack/vue-query'
import axios from 'axios'

const queryClient = useQueryClient()

const { mutate } = useMutation({
  mutationFn: (newTodo) => axios.post('/todos', newTodo),
  onMutate: async (newTodo) => {
    await queryClient.cancelQueries(['todos'])

    const previousTodos = queryClient.getQueryData(['todos'])
    queryClient.setQueryData(['todos'], (old) => [...old, newTodo])

    return { previousTodos }
  },
  onError: (err, newTodo, context) => {
    queryClient.setQueryData(['todos'], context.previousTodos)
  },
  onSettled: () => {
    queryClient.invalidateQueries(['todos'])
  },
})

function addTodo() {
  mutate({ id: new Date(), title: 'Do Laundry' })
}
</script>
```

Invalidation et Réactivation de Requêtes

Après une mutation, il est souvent nécessaire d'invalider les requêtes pour obtenir des données fraîches. Tanstack Query facilite cette tâche avec `invalidateQueries` et `refetchQueries`.

```
<script setup>
import { useMutation, useQueryClient } from '@tanstack/vue-query'
import axios from 'axios'

const queryClient = useQueryClient()

const { mutate } = useMutation({
  mutationFn: (newTodo) => axios.post('/todos', newTodo),
  onSuccess: () => {
    queryClient.invalidateQueries(['todos'])
  }
})

function addTodo() {
  mutate({ id: new Date(), title: 'Do Laundry' })
}
</script>

<template>
  <button @click="addTodo">Create Todo</button>
</template>
```

Pagination et Requêtes Infinies avec Tanstack Query

Tanstack Query prend en charge la pagination et les requêtes infinies pour charger des données en lots. Cela améliore la performance et l'expérience utilisateur lors de la navigation dans de grandes collections de données.

```
<script setup lang="ts">
import { ref } from 'vue'
import { useQuery } from '@tanstack/vue-query'

const fetcher = (page) =>
  fetch(`https://jsonplaceholder.typicode.com/posts?_page=${page}&_limit=10`)
    .then(res => res.json())

const page = ref(1)
const { isPending, isError, data, error } = useQuery({
  queryKey: ['projects', page],
  queryFn: () => fetcher(page.value),
  keepPreviousData: true,
})

const prevPage = () => {
  page.value = Math.max(page.value - 1, 1)
}
const nextPage = () => {
  page.value += 1
}
</script>
```

20

Les Plugins



Introduction aux Plugins Vue

Les plugins Vue.js permettent d'étendre les fonctionnalités de votre application. Ils peuvent ajouter des méthodes globales, des directives, des mixins et bien plus encore. Utiliser des plugins facilite la modularité et la réutilisation du code.

Création de votre Premier Plugin

Créer un plugin Vue est simple. Commencez par définir un fichier JavaScript exportant une fonction d'installation qui ajoute la fonctionnalité souhaitée à l'application Vue.

```
// plugins/monPlugin.js
export default {
  install(app) {
    console.log('Plugin installé');
  }
}
```

Structure de base d'un plugin Vue

Un plugin Vue de base est un objet avec une méthode install. Cette méthode prend l'instance de l'application en argument et optionnellement des options de configuration.

```
// plugins/basePlugin.js
export default {
  install: (app, options) => {
    console.log('Base Plugin installé avec les options:', options)
  }
}

// main.js
import { createApp } from 'vue'
import App from './App.vue'
import basePlugin from './plugins/basePlugin'

const app = createApp(App)
app.use(basePlugin, { option1: 'valeur1' })
app.mount('#app')
```

Ajout de directives personnalisées

Les directives personnalisées permettent d'étendre les fonctionnalités des éléments du DOM.

```
// plugins/directivePlugin.js
export default {
  install: (app, options) => {
    app.directive('color', {
      beforeMount(el, binding) {
        el.style.color = binding.value
      }
    })
  }
}

// main.js
import { createApp } from 'vue'
import App from './App.vue'
import directivePlugin from './plugins/directivePlugin'

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

Création de composants globaux dans un plugin

Les plugins peuvent également enregistrer des composants globaux accessibles dans toute l'application.

```
// plugins/globalComponents.js
import MyComponent from './components/MyComponent.vue'

export default {
  install: (app, options) => {
    app.component('MyComponent', MyComponent)
  }
}

// main.js
import { createApp } from 'vue'
import App from './App.vue'
import globalComponents from './plugins/globalComponents'

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

Utilisation de provide/inject dans les plugins

Les plugins peuvent utiliser provide et inject pour partager des données ou des méthodes entre les composants.

```
// plugins/providePlugin.js
export default {
  install: (app, options) => {
    app.provide('pluginData', options.data)
  }
}

// main.js
import { createApp } from 'vue'
import App from './App.vue'
import providePlugin from './plugins/providePlugin'

const app = createApp(App)
app.use(providePlugin, { data: 'someData' })
app.mount('#app')
```

21

Mise en place d'un layout



Introduction à la mise en place d'un Layout

Un layout dans une application Vue.js permet de structurer la disposition de l'interface utilisateur, en définissant des sections communes comme l'en-tête, le pied de page et la barre de navigation. Utiliser des layouts permet de réutiliser ces sections à travers différentes pages de l'application.

```
<template>
  <div>
    <Header />
    <main>
      <router-view />
    </main>
    <Footer />
  </div>
</template>

<script setup>
import Header from './components/Header.vue'
import Footer from './components/Footer.vue'
</script>

<style scoped>
/* Styles du layout */
</style>
```

Utilisation de la balise <component :is>

La balise <component :is> est utilisée pour rendre dynamiquement un composant selon une condition. Cela est particulièrement utile pour gérer les layouts variés basés sur les routes.

```
<template>
  <component :is="layout">
    <router-view />
  </component>
</template>

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

const route = useRoute()
const layout = computed(() => route.meta.layout || 'default-layout')
</script>

<style scoped>
/* Styles du layout dynamique */
</style>
```

Configuration de Vue Router

Vue Router est utilisé pour définir les routes de l'application et les layouts associés à chaque route via les métadonnées (meta).

```
// router/index.js
import { createRouter, createWebHistory } from 'vue-router'
import DefaultLayout from '../layouts/DefaultLayout.vue'
import AdminLayout from '../layouts/AdminLayout.vue'
import Home from '../views/Home.vue'
import Admin from '../views/Admin.vue'

const routes = [
  {
    path: '/',
    component: Home,
    meta: { layout: DefaultLayout }
  },
  {
    path: '/admin',
    component: Admin,
    meta: { layout: AdminLayout }
  }
]

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

export default router
```

Création de Layouts personnalisés

Définir des layouts personnalisés en tant que composants Vue permet de réutiliser la structure de base sur différentes pages.

```
<!-- layouts/DefaultLayout.vue -->
<template>
  <div>
    <Header />
    <main>
      <slot />
    </main>
    <Footer />
  </div>
</template>

<script setup>
import Header from '../components/Header.vue'
import Footer from '../components/Footer.vue'
</script>

<style scoped>
/* Styles du layout par défaut */
</style>
```

Intégration des Layouts dans l'Application Vue

Intégrer les layouts dans l'application en utilisant Vue Router et la balise `<component :is>` pour rendre dynamiquement le layout approprié.

```
<!-- App.vue -->
<template>
  <component :is="layout">
    <router-view />
  </component>
</template>

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

const route = useRoute()
const layout = computed(() => route.meta.layout || 'default-layout')
</script>

<style scoped>
/* Styles de l'application */
</style>
```

Exemple complet avec Vue Router et Layouts

Un exemple complet intégrant Vue Router, layouts dynamiques et métadonnées pour définir des layouts spécifiques à chaque route.

```
// main.js
import { createApp } from 'vue'
import App from './App.vue'
import router from './router'
import DefaultLayout from './layouts/DefaultLayout.vue'
import AdminLayout from './layouts/AdminLayout.vue'

const app = createApp(App)

app.component('default-layout', DefaultLayout)
app.component('admin-layout', AdminLayout)

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

22

Mise en place de tests dans Vue



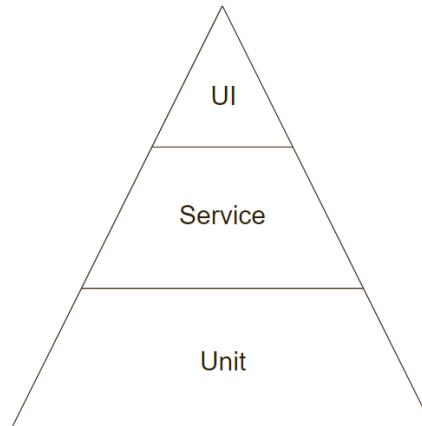
Introduction au test moderne de vue

Les tests automatisés permettent de s'assurer que les fonctionnalités précédemment fonctionnelles continuent de l'être sans intervention manuelle, augmentant ainsi la confiance dans les modifications du code.

Les tests automatisés apportent confiance lors des refactorisations, servent de documentation toujours à jour et préviennent les bugs et régressions.

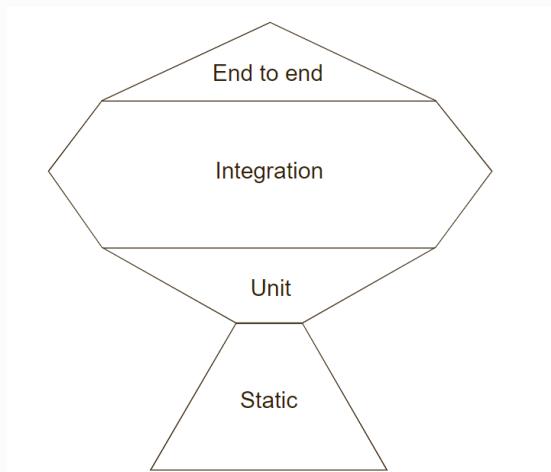
Présentation de la pyramide de tests

Explication de la pyramide des tests qui conseille une grande quantité de tests unitaires par rapport aux tests d'UI qui sont plus coûteux.



Approches alternatives de test : Le testing trophy de Kent C. Dodds

“It says that integration tests give you the biggest return on investment, so you should write more integration tests than any other kinds of tests.”



Quels outils ?

L'analyse statique permet de détecter les erreurs de syntaxe, les mauvaises pratiques et l'utilisation incorrecte des API : Formateurs de code, comme Prettier ; Les linters, comme ESLint ; Les vérificateurs de type, comme TypeScript et Flow.

Les tests unitaires vérifient que les algorithmes délicats fonctionnent correctement.
Outils : Jest / Vitest.

Quels outils ?

Les tests d'intégration vous donnent l'assurance que toutes les fonctionnalités de votre application fonctionnent comme prévu. Outils : Vitest & vue-testing-library.

Les tests de bout en bout permettent de s'assurer que votre application fonctionne comme un tout : le frontend, le backend, la base de données et tout le reste. Outils : Cypress ; Playwright

Vitest



Présentation de Vitest

- **Description:** Vitest est un framework moderne pour le test JavaScript, inspiré de Jest, rapide et conçu pour être intégré avec Vite.
- **Points forts:** Support des ESM, test en parallèle, rapidité.

```
import { defineConfig } from 'vite'
import react from '@vitejs/plugin-react'

export default defineConfig({
  plugins: [react()],
  test: {
    globals: true,
    environment: 'jsdom',
    setupFiles: './src/setupTests.js',
  },
})
```

```
npm install vite vitest @vitejs/plugin-react --save-dev
```

```
import '@testing-library/jest-dom';

{
  "scripts": {
    "test": "vitest"
  }
}
```

Premier test : qu'est-ce qu'un test ?

Un test vérifie le comportement attendu d'une fonction ou d'un bout de code.

```
import { expect, test } from 'vitest';

test('addition works', () => {
  expect(1 + 1).toBe(2); // Verifies that the result of 1 + 1 is indeed 2
});
```

Qu'est-ce qu'une assertion ?

Une assertion vérifie qu'une condition est vraie. Vitest utilise expect() pour définir les assertions.

```
expect(1 + 1).toBe(2); // This assertion checks if 1 + 1 equals 2
```

Liste des assertions dans Vitest

Liste des principales assertions utilisées dans Vitest.

```
expect(value).toBe(expected); // Asserts that value === expected
expect(value).toEqual(expected); // Asserts deep equality (for objects or arrays)
expect(value).toBeTruthy(); // Asserts that the value is truthy
expect(value).toBeFalsy(); // Asserts that the value is falsy
expect(value).toContain(item); // Asserts that the array or string contains the item
expect(value).toBeGreaterThan(number); // Asserts that the value is greater than a number
expect(value).toThrow(); // Asserts that a function throws an error
```

Détail sur toBe vs toEqual

Différence entre toBe et toEqual. toBe vérifie l'égalité stricte (==), tandis que toEqual compare la structure des objets et tableaux.

```
expect({ a: 1 }).toEqual({ a: 1 }); // Passes, because the objects are deeply equal
expect({ a: 1 }).toBe({ a: 1 });    // Fails, because they are different object refere
```

Test d'une fonction utilitaire : multiply

Exemple de test sur une fonction utilitaire.

```
export function multiply(a, b) {
    return a * b;
}
```

```
import { multiply } from '../services/multiply';
import { expect, test } from 'vitest';

test('multiply function works correctly', () => {
    expect(multiply(2, 3)).toBe(6); // Multiplication of 2 and 3 should return 6
    expect(multiply(2, 0)).toBe(0); // Multiplying by zero should return 0
});
```

Groupement de tests avec describe

Utilisation de describe pour organiser des tests par thème ou par fonction.

```
import { describe, expect, test } from 'vitest';

describe('multiplication tests', () => {
    test('multiply positive numbers', () => {
        expect(multiply(2, 3)).toBe(6); // 2 * 3 equals 6
    });

    test('multiply by zero', () => {
        expect(multiply(2, 0)).toBe(0); // Multiplying any number by 0 returns 0
    });

    test('multiply by negative number', () => {
        expect(multiply(2, -2)).toBe(-4); // 2 * -2 equals -4
    });
});
```

Tester des fonctions asynchrones

Comment tester des fonctions asynchrones dans Vitest.

```
export async function fetchData() {
    return new Promise((resolve) => {
        setTimeout(() => resolve('data'), 1000);
    });
}

import { fetchData } from '../services/dataService';
import { expect, test } from 'vitest';

test('fetchData returns correct data', async () => {
    const data = await fetchData();
    expect(data).toBe('data'); // Asserts that fetchData resolves to 'data'
});
```

Utilisation de beforeEach et afterEach

beforeEach permet d'exécuter du code avant chaque test, et afterEach après chaque test.
Très utile pour initialiser ou nettoyer des variables.

```
let counter;

beforeEach(() => {
  counter = 0; // Initialize counter before each test
});

test('increments counter', () => {
  counter++;
  expect(counter).toBe(1); // The counter should be 1 after incrementing
});

afterEach(() => {
  // Optionally clean up after each test
});
```

Mocking des services HTTP avec Vitest

Utilisation de vi.mock() pour simuler des appels HTTP dans les tests, sans avoir à réellement appeler des services externes.

```
import axios from 'axios';
import { vi } from 'vitest';

vi.mock('axios'); // Mock axios to prevent real HTTP requests
const mockedAxios = vi.mocked(axios);

mockedAxios.get.mockResolvedValue({ data: { userId: 1 } }); // Mock the get request

test('fetches user data', async () => {
  const response = await axios.get('/user/1');
  expect(response.data.userId).toBe(1); // Expect the mocked data to be returned
});
```

Utilisation des stubs pour simuler des comportements

Les stubs permettent de remplacer temporairement une fonction pour tester différentes situations.

```
import { vi } from 'vitest';

const logStub = vi.fn(); // Stub a function

function greet() {
  logStub('Hello'); // Use the stub instead of the real implementation
}

test('log is called with correct argument', () => {
  greet();
  expect(logStub).toHaveBeenCalledWith('Hello'); // Verify that the stub was called
});
```

Mocking de fonctions asynchrones

Comment créer des mocks de fonctions asynchrones avec Vitest.

```
import { vi } from 'vitest';

const asyncFunction = vi.fn().mockResolvedValue('mocked data'); // Mock an async function

test('async function returns mocked data', async () => {
    const result = await asyncFunction();
    expect(result).toBe('mocked data'); // Expect the mocked result
});
```

Tester des erreurs avec toThrow()

Utilisation de l'assertion toThrow() pour tester si une fonction lance une erreur

```
function throwError() {
  throw new Error('This is an error!');
}

test('function throws an error', () => {
  expect(() => throwError()).toThrow('This is an error!'); // Expect an error to be thrown
});
```

Gestion des tests avec des valeurs paramétrées

Utilisation de test.each() pour exécuter le même test avec différentes valeurs.

```
test.each([[1, 1, 2], [2, 2, 4], [3, 3, 6]])(  
  'adds %i and %i to equal %i',  
  (a, b, expected) => {  
    expect(a + b).toBe(expected); // Test addition with different values  
  }  
);
```

Utilisation de test.only pour exécuter un test unique

test.only permet de n'exécuter qu'un seul test, utile pour le débogage.

```
test.only('runs this test only', () => {
  expect(1 + 1).toBe(2); // Only this test will run
});
```

Combiner les assertions pour tester plusieurs aspects

Exemple combinant plusieurs assertions pour vérifier différentes parties du code.

```
test('checks multiple aspects', () => {
  const user = { name: 'Bob', age: 30 };
  expect(user.name).toBe('Bob'); // Verify the name
  expect(user.age).toBeGreaterThan(20); // Verify the age is greater than 20
});
```

Mocking de modules entiers

Utilisation de vi.mock() pour simuler des modules entiers, pas seulement des fonctions spécifiques.

```
vi.mock('../utils/math', () => ({
  multiply: vi.fn(() => 10) // Mock the entire module with custom implementations
}));

import { multiply } from '../utils/math';

test('multiply function is mocked', () => {
  expect(multiply(2, 3)).toBe(10); // The mocked function returns 10 regardless of i
});
```

Écrire des tests pour des fonctions purement utilitaires

Les fonctions utilitaires peuvent être testées isolément car elles n'ont pas de dépendances extérieures.

```
function add(a, b) {
  return a + b;
}

test('add function works', () => {
  expect(add(2, 3)).toBe(5); // Test the utility function directly
});
```

Utilisation de spyOn pour observer les appels de fonction

vi.spyOn() permet de surveiller les appels à une fonction sans la remplacer.

```
const obj = {
  log: (message) => console.log(message),
};

test('spy on log function', () => {
  const spy = vi.spyOn(obj, 'log'); // Spy on the log function

  obj.log('Hello, world!');
  expect(spy).toHaveBeenCalledWith('Hello, world!'); // Check that Log was called
});
```

Tester les effets secondaires

Tester des fonctions qui modifient un état externe ou un environnement, par exemple en modifiant des variables globales ou le stockage local.

```
test('modifies global variable', () => {
  global.counter = 0; // Set a global variable
  function incrementCounter() {
    global.counter++;
  }

  incrementCounter();
  expect(global.counter).toBe(1); // Check if the global variable was modified
});
```

Tests avec des timers simulés

Utilisation de vi.useFakeTimers() pour simuler des délais ou des timers dans les tests.

```
test('delays execution', () => {
  vi.useFakeTimers(); // Use fake timers for the test

  let value = false;
  setTimeout(() => {
    value = true;
  }, 1000);

  vi.runAllTimers(); // Fast-forward all timers

  expect(value).toBe(true); // The value should now be true after the timer has run
});
```

Contrôler le temps avec vi.advanceTimersByTime

Avancer manuellement le temps pour simuler des retards dans l'exécution du code.

```
test('advances timers by specific time', () => {
  vi.useFakeTimers();

  let value = false;
  setTimeout(() => {
    value = true;
  }, 1000);

  vi.advanceTimersByTime(500); // Move time forward by 500ms
  expect(value).toBe(false); // The timer hasn't completed yet

  vi.advanceTimersByTime(500); // Move time forward by another 500ms
  expect(value).toBe(true); // Now the timer should have completed
});
```

Mocking des modules externes

Utilisation de vi.mock() pour simuler des modules ou des bibliothèques externes dans les tests.

```
vi.mock('axios', () => ({
  get: vi.fn(() => Promise.resolve({ data: { id: 1 } }))
}));

import axios from 'axios';

test('fetches data with mocked axios', async () => {
  const response = await axios.get('/api/user');
  expect(response.data.id).toBe(1); // Test the mocked response
});
```

Tests de performance avec vi.measure

Utilisation de vi.measure() pour mesurer la performance d'une fonction ou d'un morceau de code.

```
test('measures performance of function', () => {
  const timeTaken = vi.measure(() => {
    let sum = 0;
    for (let i = 0; i < 1000000; i++) {
      sum += i;
    }
    return sum;
  });

  expect(timeTaken.duration).toBeLessThan(100); // Expect the function to execute
});
```

Mocking des dates avec vi.setSystemTime

Simuler des dates et des heures spécifiques pour tester des fonctions dépendantes du temps.

```
test('mocks system date', () => {
  const mockDate = new Date(2020, 1, 1);
  vi.setSystemTime(mockDate); // Set the system time to a specific date

  expect(new Date().getFullYear()).toBe(2020); // The current year should now be 2020
});
```

Mocking des événements avec des callbacks

Utilisation de mocks pour simuler des événements ou des callbacks dans les tests.

```
function onClick(callback) {
  callback('Button clicked');
}

test('calls the callback on click', () => {
  const mockCallback = vi.fn(); // Mock the callback function

  onClick(mockCallback);
  expect(mockCallback).toHaveBeenCalledWith('Button clicked'); // The callback should
});
```

Tests d'interaction entre plusieurs services

Tester l'intégration entre plusieurs services ou modules dans une application.

```
import { getUser, saveUser } from '../services/userService';

test('gets and saves user data', () => {
  const user = getUser(1);
  saveUser(user);

  expect(user.id).toBe(1); // Verify that the correct user was fetched and saved
});
```

Tests de services dépendants de l'état global

Comment tester des services qui dépendent de l'état global ou du contexte de l'application.

```
let globalState = {
  loggedIn: false,
};

function login() {
  globalState.loggedIn = true;
}

test('logs in user and updates global state', () => {
  login();
  expect(globalState.loggedIn).toBe(true); // Check if the global state was updated
});
```

Mocking d'API externes dans les services

Simuler les appels API externes dans les tests de services.

```
vi.mock('../api/externalApi', () => ({
  fetchData: vi.fn(() => Promise.resolve({ data: 'mocked data' }))
}));

import { fetchData } from '../api/externalApi';

test('fetches mocked data', async () => {
  const result = await fetchData();
  expect(result.data).toBe('mocked data'); // Test the mocked API response
});
```

Tester les retours d'erreur des services

Vérifier la gestion des erreurs dans les services en simulant des échecs.

```
function getUser(id) {
  if (id <= 0) {
    throw new Error('Invalid user ID');
  }
  return { id, name: 'John Doe' };
}

test('throws error for invalid user ID', () => {
  expect(() => getUser(-1)).toThrow('Invalid user ID'); // Expect an error to be thrown
});
```

Utilisation de toMatchObject pour vérifier des objets partiels

toMatchObject() permet de vérifier partiellement un objet sans nécessiter une égalité complète.

```
const user = { id: 1, name: 'John', age: 30 };

test('matches partial object', () => {
  expect(user).toMatchObject({ id: 1, name: 'John' }); // Only checks for matching
});
```

Exécuter des Tests avec Vitest

Exécution des tests avec Vitest via la ligne de commande et en mode watch.

```
npm run test  # Exécute tous les tests  
npm run test -- --watch # Mode watch pour réexécuter automatiquement les tests
```

Exécution des Tests avec Couverture

Comment générer des rapports de couverture de tests avec Vitest pour visualiser les zones non testées du code.

```
vitest run --coverage
```

Optimisation des Tests avec Vitest

Optimisation des performances des tests en activant les tests en parallèle et le mode watch.

```
test: {  
  environment: 'jsdom',  
  maxThreads: 4,  
  minThreads: 2,  
},
```

Utilisation de plugins Vitest

Vitest prend en charge divers plugins pour étendre ses fonctionnalités.

```
// Utilisation du plugin de couverture de code
import { defineConfig } from 'vitest/config';
import { plugin as coveragePlugin } from 'vite-plugin-istanbul';

export default defineConfig({
  plugins: [coveragePlugin()],
});
```

Configuration de vue-test-utils avec Vitest

Pour tester les composants Vue, configurez vue-test-utils avec Vitest.

```
// Exemples de tests de composants
import { mount } from '@vue/test-utils';
import { describe, it, expect } from 'vitest';
import MyComponent from './MyComponent.vue';

describe('MyComponent', () => {
  it('renders properly', () => {
    const wrapper = mount(MyComponent);
    expect(wrapper.text()).toContain('Hello Vue 3');
  });
});
```

Tests de composants Vue de base

Testez les composants Vue pour vérifier leur rendu et leur comportement.

```
import { mount } from '@vue/test-utils';
import MyComponent from './MyComponent.vue';

test('renders the correct text', () => {
  const wrapper = mount(MyComponent);
  expect(wrapper.text()).toContain('Hello Vue 3');
});
```

Tests des props des composants

Vérifiez que les composants reçoivent et utilisent correctement les props.

```
<!-- MyComponent.vue -->
<template>
  <div>Hello {{ name }}</div>
</template>

<script setup>
defineProps(['name']);
</script>
```

```
import { mount } from '@vue/test-utils';
import MyComponent from './MyComponent.vue';

test('affiche le nom correctement', () => {
  const wrapper = mount(MyComponent, {
    props: { name: 'Vue' },
  });
  expect(wrapper.text()).toContain('Hello Vue');
});
```

Tests des événements des composants

Testez les événements émis par les composants pour vérifier les interactions.

```
<!-- MyButton.vue -->
<template>
  <button @click="$emit('click')">Click me</button>
</template>
```

```
import { mount } from '@vue/test-utils';
import MyButton from './MyButton.vue';

test('émet l\'événement click', async () => {
  const wrapper = mount(MyButton);
  await wrapper.find('button').trigger('click');
  expect(wrapper.emitted()).toHaveProperty('click');
});
```

Tests des slots des composants

Vérifiez que les slots sont correctement rendus et utilisés dans les composants.

```
<!-- MyComponent.vue -->
<template>
  <div>
    <slot></slot>
  </div>
</template>
```

```
import { mount } from '@vue/test-utils';
import MyComponent from './MyComponent.vue';

test('rend le contenu du slot', () => {
  const wrapper = mount(MyComponent, {
    slots: {
      default: '<p>Contenu du slot</p>',
    },
  });
  expect(wrapper.html()).toContain('<p>Contenu du slot</p>');
});
```

Tests des propriétés réactives

Testez les propriétés réactives pour vérifier qu'elles réagissent correctement aux changements.

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

const count = ref(0);

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

</script>

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

```
import { mount } from '@vue/test-utils';
import MyComponent from './MyComponent.vue';

test('test reactive properties', async () => {
  const wrapper = mount(MyComponent);
  expect(wrapper.text()).toContain('0');
  await wrapper.find('button').trigger('click');
  expect(wrapper.text()).toContain('1');
});
```

Tests des routes avec Vue Router et Vitest

Testez la navigation et le comportement des routes avec Vue Router et Vitest.

```
import { mount } from '@vue/test-utils';
import { createRouter, createWebHistory } from 'vue-router';
import { routes } from './router';
import App from './App.vue';

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

test('test route navigation', async () => {
  router.push('/about');
  await router.isReady();
  const wrapper = mount(App, {
    global: {
      plugins: [router],
    },
  });
  expect(wrapper.html()).toContain('About Page');
});
```

Tests des stores Vuex/Pinia avec Vitest

Testez les stores Vuex/Pinia pour vérifier leur comportement et leur intégration avec les composants.

```
import { setActivePinia, createPinia } from 'pinia';
import { useStore } from './store';
import { mount } from '@vue/test-utils';
import MyComponent from './MyComponent.vue';

setActivePinia(createPinia());

test('test Pinia store', () => {
  const store = useStore();
  const wrapper = mount(MyComponent);
  store.increment();
  expect(store.count).toBe(1);
});
```

Utiliser des Mocks de Données

Les mocks de données permettent de simuler des réponses d'API ou des données complexes dans vos tests.

```
// tests/MyComponent.test.js
import { mount } from '@vue/test-utils';
import MyComponent from '../src/components/MyComponent.vue';
import { ref } from 'vue';

test('renders with mock data', () => {
  const mockData = ref([{ id: 1, name: 'Test Item' }]);
  const wrapper = mount(MyComponent, {
    setup() {
      return { data: mockData };
    },
  });
  expect(wrapper.text()).toContain('Test Item');
});
```

Mock de Modules

Vous pouvez mocker des modules entiers pour simuler des fonctions ou des classes dans vos tests.

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

vi.mock('../src/api', () => ({
  fetchData: () => [{ id: 1, name: 'Mocked Item' }],
}));

test('uses mocked module', async () => {
  const wrapper = mount(MyComponent);
  await wrapper.vm.$nextTick();
  expect(wrapper.text()).toContain('Mocked Item');
});
```

Mock de Fonctions

Mocker des fonctions individuelles permet de contrôler le comportement de certaines méthodes pendant les tests.

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

const mockFunction = vi.fn(() => 'Mocked Value');

test('calls mocked function', () => {
  const wrapper = mount(MyComponent, {
    setup() {
      return { myFunction: mockFunction };
    },
  });
  wrapper.vm.myFunction();
  expect(mockFunction).toHaveBeenCalled();
});
```

Utilisation de vi.spyOn

vi.spyOn permet de surveiller et de mocker les implémentations des méthodes d'objets.

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

const mockFetch = vi.spyOn(window, 'fetch').mockResolvedValue({
  json: () => Promise.resolve([{ id: 1, name: 'SpyOn Item' }]),
});

test('fetches data using spyOn', async () => {
  const wrapper = mount(MyComponent);
  await wrapper.vm.fetchData();
  expect(wrapper.text()).toContain('SpyOn Item');
});
```

Mock d'API

Les mocks d'API permettent de simuler des appels réseau et de contrôler les réponses pour les tests.

```
// tests/MyComponent.test.js
import { mount } from '@vue/test-utils';
import MyComponent from '../src/components/MyComponent.vue';
import { ref } from 'vue';

const mockApi = ref([{ id: 1, name: 'Mocked API Item' }]);

test('renders with mocked API data', () => {
  const wrapper = mount(MyComponent, {
    setup() {
      return { apiData: mockApi };
    },
  });
  expect(wrapper.text()).toContain('Mocked API Item');
});
```

Mock de Composants Enfants

Mocker des composants enfants permet de tester un composant parent sans dépendre des implémentations des enfants.

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

test('renders with mocked child component', () => {
  const wrapper = mount(MyComponent, {
    global: {
      stubs: {
        ChildComponent: {
          template: '<div>Mocked Child Component</div>',
        },
      },
    },
  ));
  expect(wrapper.html()).toContain('Mocked Child Component');
});
```

Introduction aux tests E2E

Les tests End-to-End (E2E) permettent de tester une application entière, du front-end au back-end, pour vérifier le fonctionnement complet.

```
// Un test E2E basique avec Cypress
describe('My First Test', () => {
  it('visits the app root url', () => {
    cy.visit('/');
    cy.contains('h1', 'Welcome to Your Vue.js + TypeScript App');
  });
});
```

Création d'un premier test avec Cypress

Créez un fichier de test dans le dossier cypress/integration et écrivez votre premier test.

```
// cypress/integration/sample_spec.js
describe('Page d\'accueil', () => {
  it('doit afficher le titre de la page', () => {
    cy.visit('/');
    cy.contains('h1', 'Bienvenue sur votre application Vue.js');
  });
});
```

Interactions de base avec Cypress

Cypress permet de simuler des interactions utilisateur telles que les clics, la saisie de texte, et la navigation entre les pages.

```
describe('Formulaire de connexion', () => {
  it('permet à l\'utilisateur de se connecter', () => {
    cy.visit('/login');
    cy.get('input[name="username"]').type('monNomUtilisateur');
    cy.get('input[name="password"]').type('monMotDePasse');
    cy.get('button[type="submit"]').click();
    cy.url().should('include', '/dashboard');
  });
});
```

Utilisation des fixtures pour les données de test

Les fixtures sont utilisées pour fournir des données de test statiques à vos tests. Créez un fichier JSON dans le dossier cypress/fixtures.

```
// cypress/fixtures/user.json
{
  "username": "testuser",
  "password": "password123"
}
```

```
describe('Formulaire de connexion', () => {
  beforeEach(() => {
    cy.fixture('user').then((user) => {
      this.user = user;
    });
  });

  it('permet à l\'utilisateur de se connecter', function() {
    cy.visit('/login');
    cy.get('input[name="username"]').type(this.user.username);
    cy.get('input[name="password"]').type(this.user.password);
    cy.get('button[type="submit"]').click();
    cy.url().should('include', '/dashboard');
  });
});
```

Tests des requêtes API avec Cypress

Cypress permet de tester les requêtes API et de vérifier les réponses pour s'assurer qu'elles sont correctes.

```
describe('Requêtes API', () => {
  it('vérifie la réponse de l\'API', () => {
    cy.request('GET', '/api/users').then((response) => {
      expect(response.status).to.eq(200);
      expect(response.body).to.have.length.greaterThan(0);
    });
  });
});
```

Utilisation de commandes personnalisées

Les commandes personnalisées permettent de réutiliser des séquences d'actions dans différents tests. Elles sont définies dans cypress/support/commands.js.

```
// cypress/support/commands.js
Cypress.Commands.add('login', (username, password) => {
  cy.visit('/login');
  cy.get('input[name="username"]').type(username);
  cy.get('input[name="password"]').type(password);
  cy.get('button[type="submit"]').click();
});

// Utilisation de la commande personnalisée dans un test
describe('Tableau de bord', () => {
  it('affiche le tableau de bord après connexion', () => {
    cy.login('testuser', 'password123');
    cy.url().should('include', '/dashboard');
  });
});
```

Premier test avec Playwright

Exemple de test basique avec Playwright pour vérifier le rendu de la page d'accueil.

```
// tests/home.spec.js
const { test, expect } = require('@playwright/test');

test('La page d\'accueil doit afficher le titre', async ({ page }) => {
  await page.goto('http://localhost:3000');
  await expect(page).toHaveTitle(/Bienvenue à Playwright avec Vue 3/);
});
```

Test d'interactions utilisateur

Exemple de test d'une interaction utilisateur sur un composant.

```
// tests/counter.spec.js
const { test, expect } = require('@playwright/test');

test('Le bouton doit incrémenter le compteur', async ({ page }) => {
    await page.goto('http://localhost:3000');
    await page.click('button');
    await expect(page.locator('button')).toHaveText('Compteur: 1');
});
```

Test de soumission de formulaire

Vérifier la soumission de formulaire avec Playwright.

```
// tests/form.spec.js
const { test, expect } = require('@playwright/test');

test('Le formulaire doit soumettre le nom d\'utilisateur', async ({ page }) => {
    await page.goto('http://localhost:3000');
    await page.fill('input', 'testuser');
    await page.click('button[type="submit"]');
    page.on('dialog', dialog => {
        expect(dialog.message()).toBe("Nom d'utilisateur: testuser");
        dialog.dismiss();
    });
});
```

Mocking des requêtes HTTP

Utiliser Playwright pour intercepter et mocker les requêtes HTTP.

```
// tests/mock.spec.js
const { test, expect } = require('@playwright/test');

test('Mocker une requête API', async ({ page }) => {
    await page.route('https://api.example.com/data', route =>
        route.fulfill({
            contentType: 'application/json',
            body: JSON.stringify({ data: 'Mocked Data' }),
        })
    );
    await page.goto('http://localhost:3000');
    // Continue with further assertions
});
```

Introduction à Storybook

Storybook est un outil open-source pour développer et documenter des composants UI de manière isolée. Il permet aux développeurs de tester, visualiser et organiser les composants dans un environnement sandbox.

Installation de Storybook pour Vue 3

L'installation de Storybook pour un projet Vue 3 se fait via npm ou yarn. Cela inclut l'ajout des dépendances nécessaires et la configuration initiale pour démarrer Storybook.

```
npx storybook init
```

Configuration Initiale

La configuration initiale de Storybook implique la modification des fichiers de configuration pour s'assurer que Storybook peut charger et rendre les composants Vue correctement.

```
my-project/
├── src/
│   ├── components/
│   │   └── MyComponent.vue
│   └── stories/
│       └── MyComponent.stories.js
└── .storybook/
    ├── main.js
    └── preview.js
```

Création de Stories

Les stories sont des représentations visuelles des composants. Chaque story décrit un état ou une variation d'un composant, permettant de les visualiser et de les tester en isolation.

Création de votre première story

Pour créer une première story, il suffit de définir un composant, de créer un template, et de spécifier les arguments (props) nécessaires pour cette instance particulière du composant.

```
import { fn } from '@storybook/test';

import Task from './Task.vue';

export const ActionsData = {
  onPinTask: fn(),
  onArchiveTask: fn(),
};

export default {
  component: Task,
  title: 'Task',
  tags: ['autodocs'],
  //👉 Our exports that end in "Data" are not stories.
  excludeStories: /.*Data$/,
  args: {
    ...ActionsData
  }
};
```

```
export const Default = {
  args: {
    task: {
      id: '1',
      title: 'Test Task',
      state: 'TASK_INBOX',
    },
  },
};

export const Pinned = {
  args: {
    task: {
      ...Default.args.task,
      state: 'TASK_PINNED',
    },
  },
};
```

Stories et Composition API

La Composition API de Vue 3 permet de structurer le code de manière plus flexible et modulaire. Les stories peuvent intégrer la Composition API pour créer des composants plus complexes et maintenables.

```
// src/stories/CompositionAPI.stories.js
import { ref } from 'vue';
import MyComponent from '../components/MyComponent.vue';

export default {
  title: 'Example/CompositionAPI',
  component: MyComponent,
};

const Template = (args) => ({
  components: { MyComponent },
  setup() {
    const state = ref(args.initialState);
    return { state };
  },
  template: '<MyComponent :state="state" />',
});

export const UsingCompositionAPI = Template.bind({});
UsingCompositionAPI.args = {
  initialState: 'State from Composition API',
};
```

Utilisation de la Composition API dans les stories

La Composition API peut être utilisée dans les stories pour gérer les états locaux, les réactifs, et les computed properties, rendant les stories plus interactives et dynamiques.

```
// src/stories/UseCompositionAPI.stories.js
import { ref, computed } from 'vue';
import MyComponent from '../components/MyComponent.vue';

export default {
  title: 'Example/UseCompositionAPI',
  component: MyComponent,
};

const Template = (args) => ({
  components: { MyComponent },
  setup() {
    const state = ref(args.initialState);
    const doubled = computed(() => state.value * 2);
    return { state, doubled };
  },
  template: '<MyComponent :state="state" :doubled="doubled" />',
});

export const DynamicState = Template.bind({});
DynamicState.args = {
  initialState: 2,
};
```

Intégration des hooks de la Composition API

Les hooks de la Composition API, tels que ref, reactive, computed, et watch, peuvent être intégrés dans les stories pour créer des scénarios interactifs et sophistiqués.

```
// src/stories/CompositionHooks.stories.js
import { ref, watch } from 'vue';
import MyComponent from '../components/MyComponent.vue';

export default {
  title: 'Example/CompositionHooks',
  component: MyComponent,
};

const Template = (args) => ({
  components: { MyComponent },
  setup() {
    const count = ref(args.initialCount);
    watch(count, (newVal) => {
      console.log('Count changed:', newVal);
    });
    return { count };
  },
  template: '<MyComponent :count="count" />',
});

export const UsingHooks = Template.bind({});
UsingHooks.args = {
  initialCount: 0,
};
```

Addons de Storybook

Les addons augmentent les capacités de Storybook en ajoutant des fonctionnalités supplémentaires comme les actions, les knobs, les backgrounds, etc. Ils permettent d'améliorer l'expérience de développement et de documentation.

Configuration des addons populaires

Certains addons sont très populaires et souvent utilisés, comme `@storybook/addon-actions` pour gérer les événements et `@storybook/addon-knobs` pour manipuler les props dynamiques des composants.

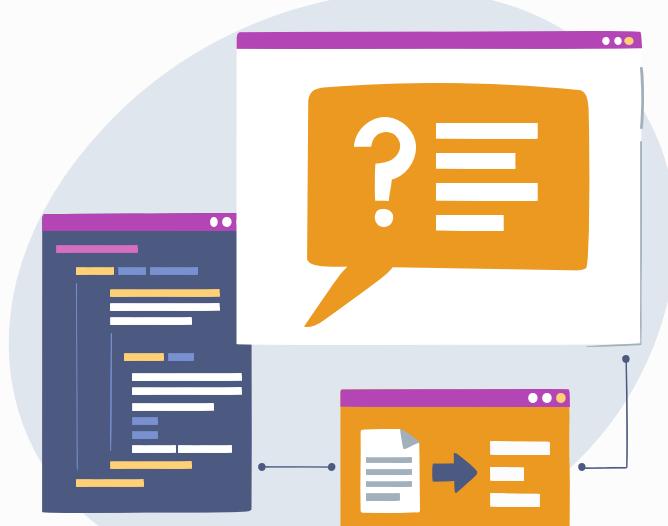
```
// .storybook/main.js
module.exports = {
  addons: ['@storybook/addon-knobs', '@storybook/addon-essentials'],
};
```

Documentation des composants avec addon-info

@storybook/addon-info permet de documenter automatiquement les composants et leurs props. Cela inclut les descriptions, les types de props, et les valeurs par défaut, offrant une documentation complète et interactive.

```
// .storybook/main.js
module.exports = {
  addons: ['@storybook/addon-info'],
};
```

E2E avec Cypress



Introduction à Cypress

Cypress est un outil puissant pour les tests end-to-end (E2E) conçu pour les applications web modernes.

```
npm install cypress --save-dev
```

Lancer Cypress

Une fois installé, Cypress peut être lancé en mode interactif ou en mode headless.

```
npx cypress open    # Ouvre L'interface graphique de Cypress  
npx cypress run     # Exécute Les tests en mode headLess
```

Configuration de Base de Cypress

Configuration de Cypress dans cypress.config.js pour adapter le comportement des tests (timeout, viewport, etc.).

```
// cypress.config.js
module.exports = {
  e2e: {
    baseUrl: 'http://localhost:3000',
    viewportWidth: 1280,
    viewportHeight: 720,
  },
};
```

Premier Test End-to-End (E2E)

Écrire un test simple avec Cypress qui vérifie le rendu de la page d'accueil d'une application.

```
describe('Homepage Test', () => {
  it('should load the homepage', () => {
    cy.visit('/');
    cy.contains('Welcome to My App').should('be.visible');
  });
});
```

Sélection des Éléments avec Cypress

Utiliser cy.get() pour sélectionner des éléments dans le DOM. Sélection par classe, ID, attributs, etc.

```
cy.get('.button-class').click();
cy.get('#element-id').should('have.text', 'Expected Text');
cy.get('[data-cy=submit-button]').click();
```

Assertions avec Cypress

Cypress utilise Chai pour les assertions. Tester les états des éléments, les URL, etc.

```
cy.url().should('include', '/dashboard');
cy.get('.alert').should('have.class', 'success');
cy.get('input').should('have.value', 'Cypress Test');
```

Interaction Utilisateur avec Cypress

Simuler des actions utilisateur comme click(), type(), clear().

```
cy.get('input[name="email"]').type('test@example.com');
cy.get('button[type="submit"]').click();
```

Tests Asynchrones avec Cypress

Cypress gère automatiquement les actions asynchrones. Il attend que les éléments soient disponibles avant d'exécuter les actions suivantes.

```
cy.get('button').click();
cy.get('.loading-spinner').should('not.exist');
cy.get('.result').should('contain', 'Success');
```

Gérer les Requêtes HTTP avec cy.intercept()

Intercepter et mocker les requêtes HTTP pour simuler des réponses d'API.

```
cy.intercept('GET', '/api/data', { fixture: 'data.json' }).as('getData');
cy.visit('/');
cy.wait('@getData');
```

Utilisation des Fixtures

Les fixtures sont des fichiers JSON ou autres formats qui contiennent des données de test pour simuler des réponses de l'API.

```
cy.fixture('user.json').then((user) => {
  cy.get('input[name="username"]').type(user.username);
  cy.get('input[name="password"]').type(user.password);
});
```

Tests de Formulaires avec Cypress

Tester des formulaires en validant les inputs, soumissions et réponses.

```
cy.get('input[name="email"]').type('test@example.com');
cy.get('input[name="password"]').type('password123');
cy.get('form').submit();
cy.get('.success-message').should('be.visible');
```

Assertions d'URL et de Navigation

Vérifier que l'URL et la navigation sont correctes après certaines actions.

```
cy.url().should('include', '/dashboard');
cy.get('a[href="/profile"]').click();
cy.url().should('include', '/profile');
```

Gestion des Cookies et Local Storage

Cypress permet d'interagir avec les cookies et le local storage pour gérer des sessions d'utilisateur.

```
cy.setCookie('session_id', '123ABC');
cy.getCookie('session_id').should('have.property', 'value', '123ABC');

cy.window().then((win) => {
  win.localStorage.setItem('token', 'authToken');
});
```

Tests avec Authentication

Gérer l'authentification avec Cypress, simuler des connexions utilisateurs et des sessions.

```
cy.request('POST', '/login', {
  username: 'user1',
  password: 'password',
}).then((response) => {
  cy.setCookie('authToken', response.body.token);
  cy.visit('/dashboard');
});
```

Intégration avec CI/CD

Intégration de Cypress dans un pipeline CI/CD comme GitHub Actions ou GitLab CI.

```
name: Cypress Tests

on: [push]

jobs:
  cypress-run:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - name: Install dependencies
        run: npm install
      - name: Run Cypress tests
        run: npx cypress run
```

Mocks avec MSW



Installation de MSW

Installer MSW pour simuler les requêtes réseau dans les tests d'intégration.

```
npm install --save-dev msw
```

Initialisation du Service Worker

Générer le script du Service Worker avec la commande msw init.

```
import * as msw from 'msw';
import { setupWorker } from 'msw/browser';
import { handlers } from './handlers';

export const worker = setupWorker(...handlers);
window.msw = { worker, ...msw };
```

Démarrage du Service Worker

Démarrer le Service Worker en mode développement.

```
async function enableMocking() {
  if (process.env.NODE_ENV !== 'development') return;

  const { worker } = await import('./mocks/browser');
  return worker.start();
}

enableMocking().then(() => {
  const root = createRoot(document.getElementById('root'));
  root.render(<App />);
});
```

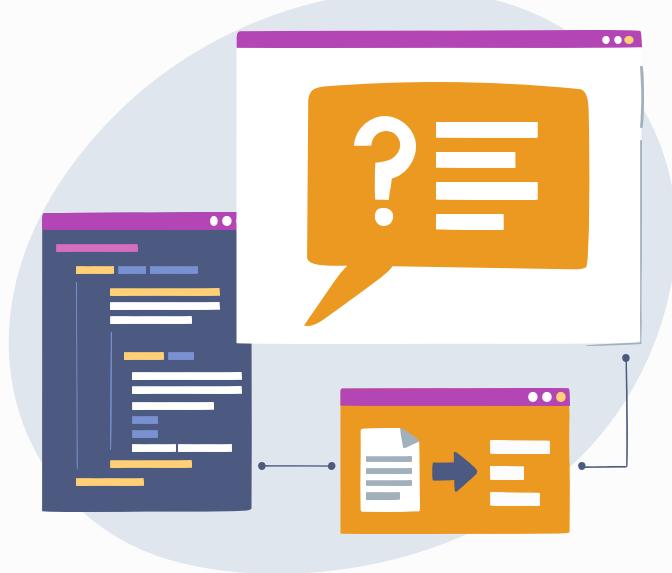
Handlers MSW

Créer des définitions de mocks pour les requêtes réseau.

```
import { http, HttpResponse } from 'msw';

export const handlers = [
  http.get('https://httpbin.org/anything', () => {
    return HttpResponse.json({
      args: { ingredients: ['bacon', 'tomato', 'mozzarella', 'pineapples'] }
    });
  })
];
```

E2E avec Playwright



Qu'est-ce que Playwright ?

Playwright est un outil de test de bout en bout qui permet de tester des applications web en simulant des interactions utilisateur réelles.

```
const { test, expect } = require('@playwright/test');

test('example test', async ({ page }) => {
    await page.goto('https://example.com');
    await expect(page).toHaveTitle('Example Domain');
});
```

Avantages de Playwright

Playwright offre une expérience de test et de débogage optimale, permet d'inspecter la page à tout moment et supporte plusieurs navigateurs.

```
const { chromium } = require('playwright');

(async () => {
    const browser = await chromium.launch();
    const page = await browser.newPage();
    await page.goto('https://example.com');
    await browser.close();
})();
```

Playwright vs Cypress

Playwright offre une API plus cohérente, une configuration plus simple, supporte les multi-onglets et est plus rapide que Cypress.

```
const { test, expect } = require('@playwright/test');

test('compare with cypress', async ({ page }) => {
    await page.goto('https://example.com');
    await expect(page.locator('h1')).toContainText('Example Domain');
});
```

Fichier de configuration Playwright

Définir les paramètres globaux et les projets pour chaque navigateur.

```
const { defineConfig, devices } = require('@playwright/test');

module.exports = defineConfig({
  testDir: './tests',
  fullyParallel: true,
  reporter: 'html',
  use: {
    baseURL: 'http://localhost:3000',
    trace: 'on-first-retry'
  },
  projects: [
    { name: 'chromium', use: { ...devices['Desktop Chrome'] } },
    { name: 'firefox', use: { ...devices['Desktop Firefox'] } },
    { name: 'webkit', use: { ...devices['Desktop Safari'] } }
  ],
  webServer: {
    command: 'npm run start',
    url: 'http://localhost:3000',
    reuseExistingServer: !process.env.CI
  }
});
```

Test de base avec Playwright

Créer un test de base pour vérifier le texte "welcome back".

```
const { test, expect } = require('@playwright/test');

test('hello world', async ({ page }) => {
    await page.goto('/');
    await expect(page.getByText('welcome back')).toBeVisible();
});
```

Requête d'éléments avec Playwright

Utiliser des sélecteurs sémantiques pour requêter les éléments DOM.

```
await page.getByRole('button', { name: 'submit' }).click();
```

Tester les interactions de base

Tester la navigation et l'interaction de base.

```
const { test, expect } = require('@playwright/test');

test('navigates to another page', async ({ page }) => {
    await page.goto('/');
    await page.getByRole('link', { name: 'remotepizza' }).click();
    await expect(page.getByRole('heading', { name: 'pizza' })).toBeVisible();
});
```

Tester les formulaires avec Playwright

Utiliser des locators pour remplir et soumettre des formulaires.

```
const { test, expect } = require('@playwright/test');

test('should show success page after submission', async ({ page }) => {
    await page.goto('/signup');
    await page.getLabel('first name').fill('Chuck');
    await page.getLabel('last name').fill('Norris');
    await page.getLabel('country').selectOption({ label: 'Russia' });
    await page.getLabel('subscribe to our newsletter').check();
    await page.getRole('button', { name: 'sign in' }).click();
    await expect(page.getText('thank you for signing up')).toBeVisible();
});
```

Tester les liens ouvrant dans un nouvel onglet

Vérifier l'attribut href ou obtenir le handle de la nouvelle page après avoir cliqué sur le lien.

```
const pagePromise = context.waitForEvent('page');
await page.getByRole('link', { name: 'terms and conditions' }).click();
const nextPage = await pagePromise;
await expect(nextPage.getText("I'm baby")).toBeVisible();
```

Nuxt 3



01

Introduction



Introduction à Nuxt

- Nuxt est un framework basé sur Vue.js, conçu pour créer des applications performantes et optimisées, que ce soit en mode serveur (SSR) ou statique. Il simplifie considérablement le développement grâce à ses conventions prédéfinies, sa modularité et ses outils intégrés. Nuxt rend également le SEO plus accessible en facilitant la génération de pages statiques et le rendu côté serveur.

Qu'est-ce que Nuxt ?

- Nuxt est un framework open-source qui étend Vue.js en offrant des fonctionnalités supplémentaires comme le rendu serveur (SSR), le routage automatique basé sur les fichiers et la gestion optimisée des performances. Il permet de créer rapidement des applications robustes avec une excellente gestion SEO. En simplifiant des concepts complexes comme le routage ou la gestion d'état, Nuxt accélère considérablement le processus de développement.

Pourquoi utiliser Nuxt ?

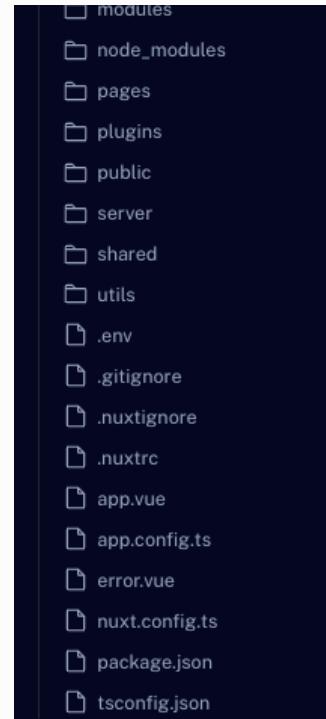
- Nuxt améliore l'expérience utilisateur en optimisant les performances avec le rendu serveur, réduisant ainsi les temps de chargement initiaux. Il simplifie également le développement grâce au routage automatique et à une gestion d'état intuitive. Enfin, Nuxt facilite le SEO et la configuration des métadonnées, ce qui améliore le classement dans les moteurs de recherche.

Installation et création d'un projet Nuxt

- Pour démarrer avec Nuxt, il suffit d'exécuter la commande officielle de création `npx nuxi init mon-projet`, suivie par l'installation des dépendances via `npm install`. Cette commande initialise un projet Nuxt avec une structure de base prête à être utilisée immédiatement. Lancer ensuite votre serveur de développement avec `npm run dev` pour commencer à travailler.

Structure d'un projet Nuxt

- Un projet Nuxt suit une structure précise avec des dossiers dédiés comme pages, layouts, components et assets. Chaque dossier a un rôle spécifique permettant une organisation claire et logique. Cette structure préétablie simplifie l'ajout de nouvelles fonctionnalités et garantit une bonne maintenabilité du projet.



Comprendre les fichiers clés : nuxt.config.ts

- Le fichier `nuxt.config.ts` centralise toute la configuration d'un projet Nuxt, incluant des réglages sur les modules, plugins, routes, et bien plus. Il permet de personnaliser le comportement global de votre application très facilement. Modifier ce fichier est essentiel pour ajuster Nuxt précisément à vos besoins.

Rendu par défaut du projet

Le mode de rendu par défaut d'un projet Nuxt 3 est défini dans nuxt.config.ts :

```
export default defineNuxtConfig({
  ssr: true, // true pour SSR, false pour CSR
  target: 'server' // ou 'static' pour du SSG (uniquement en mode SSR)
})
```

Choisir le mode de rendu route par route

Dans chaque pages/xxx.vue, vous pouvez utiliser definePageMeta pour choisir un mode de rendu spécifique.

```
<script setup>
definePageMeta({
  prerender: false // Désactive le SSG et le SSR pour cette page
})
</script>
```

Forcer une route en SSG (Static Site Generation)

Si vous souhaitez pré-générer la page lors du build (nuxt generate en mode statique) :

```
<script setup>
definePageMeta({
  prerender: true // Active le pré-rendu (SSG) pour cette page
})
</script>
```

Rendre une route en SSR uniquement

Si vous voulez forcer le rendu sur le serveur (et éviter le SSG), vous pouvez ajouter :

```
<script setup>
definePageMeta({
  ssr: true
})
</script>
```

02

Routing



Pages et routage

- Nuxt gère le routage automatiquement via le dossier pages. Chaque fichier créé dans ce dossier correspond à une route spécifique de l'application. Cette approche simplifie énormément la gestion du routage sans avoir à configurer manuellement les routes.

Routage basé sur les fichiers (File-based routing)

- Le routage basé sur les fichiers consiste à créer automatiquement les routes selon la structure du dossier pages. Chaque fichier représente une page unique, éliminant le besoin de configurations complexes. Cette méthode accélère le développement tout en restant claire et intuitive.

Pages dynamiques avec Nuxt

- Nuxt permet de créer facilement des pages dynamiques grâce aux fichiers nommés avec une syntaxe particulière comme [id].vue. Ces pages peuvent afficher du contenu personnalisé basé sur des paramètres d'URL. Cela est particulièrement utile pour les applications avec des données variables comme des blogs ou des produits.

```
<template>
  <div>
    <h1>Produit {{ id }}</h1>
    <p>Détails du produit pour l'ID : {{ id }}</p>
  </div>
</template>

<script setup>
import { useRoute } from 'vue-router'
const route = useRoute()
const id = route.params.id
</script>
```

Paramètres de routes avancés

- Nuxt offre une gestion avancée des paramètres d'URL grâce à des syntaxes comme [...slug].vue pour capturer plusieurs segments d'URL. Vous pouvez également valider ces paramètres ou appliquer des transformations. Cela permet de gérer des cas complexes de manière simple et élégante.

```
<template>
  <div>
    <h1>Article</h1>
    <p>Chemin capturé : {{ slug.join('/') }}</p>
  </div>
</template>

<script setup>
import { useRoute } from 'vue-router'
const route = useRoute()
const slug = route.params.slug || []
</script>
```

Routes imbriquées (nested routes)

- Les routes imbriquées permettent de créer des sous-pages et des structures hiérarchiques simplement en imbriquant des dossiers dans le dossier pages. Cette structure facilite l'organisation des routes complexes et permet une gestion claire et modulaire des composants associés.

```
pages/
└ blog/
    └ index.vue      // Liste des articles
    └ _id.vue        // Détail d'un article
```

Gestion des erreurs (pages 404 et erreurs personnalisées)

- Nuxt simplifie la gestion des erreurs grâce aux pages spéciales comme `error.vue`, qui captent automatiquement les erreurs et les affichent de manière conviviale. Vous pouvez personnaliser complètement ces pages pour offrir une meilleure expérience utilisateur en cas de problème. Cela améliore l'UX et garantit un site robuste.

```
<template>
<div>
  <h1>Oups, une erreur est survenue !</h1>
  <p v-if="error.statusCode === 404">La page demandée n'existe pas.</p>
  <p v-else>Veuillez réessayer plus tard.</p>
</div>
</template>

<script setup>
const props = defineProps(['error'])
</script>
```

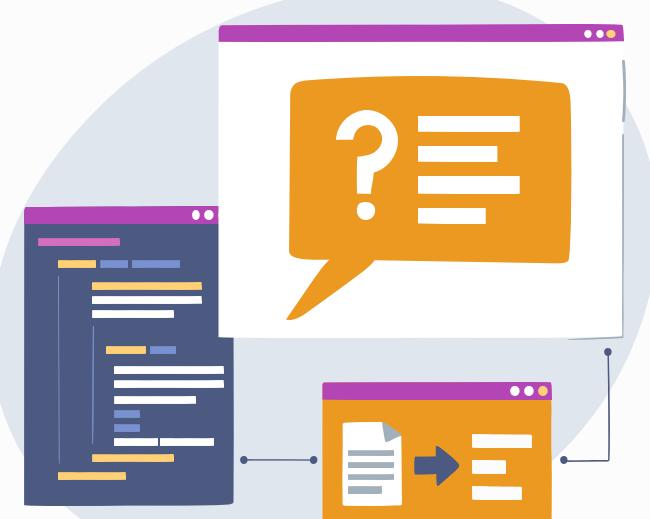
Personnaliser les routes avec le fichier routes

- Nuxt permet de personnaliser les routes en utilisant un fichier de configuration dédié. Cela offre un contrôle total sur la manière dont les URL sont mappées aux composants, permettant des configurations avancées et spécifiques.

```
export default defineNuxtConfig({
  routeRules: {
    '/ma-route-personnalisee': { redirect: '/custom' }
  }
})
```

03

Layouts & components



Layouts et Composants

- Les layouts dans Nuxt permettent de structurer l'apparence globale de votre application en définissant des mises en page réutilisables. Ils sont essentiels pour maintenir une cohérence visuelle et faciliter la gestion des composants communs comme les en-têtes et pieds de page.

```
<template>
  <div>
    <header>
      <h1>Mon Application</h1>
    </header>
    <NuxtPage />
    <footer>
      <p>Pied de page</p>
    </footer>
  </div>
</template>
```

Comprendre les layouts dans Nuxt

- Les layouts sont des composants spéciaux qui enveloppent vos pages, permettant de définir une structure de base pour votre application. Ils sont stockés dans le dossier `layouts` et peuvent être appliqués globalement ou à des pages spécifiques. Utiliser des layouts aide à organiser le code et à maintenir une apparence uniforme.

```
<template>
  <div>
    <h1>Contactez-nous</h1>
    <p>Page de contact.</p>
  </div>
</template>

<script setup>
definePageMeta({
  layout: 'default' // Utilisation du layout par défaut
})
</script>
```

Créer et appliquer plusieurs layouts

- Nuxt permet de créer plusieurs layouts pour répondre à différents besoins de mise en page. Par exemple, vous pouvez avoir un layout pour les pages d'administration et un autre pour les pages publiques. Pour appliquer un layout spécifique à une page, il suffit de définir la propriété `layout` dans le composant de la page.

Composants globaux automatiques

- Nuxt facilite l'utilisation de composants globaux grâce à l'auto-importation. En plaçant vos composants dans le dossier `components`, Nuxt les rend disponibles automatiquement dans toute l'application. Cela réduit la nécessité d'importer manuellement chaque composant, accélérant ainsi le développement.

```
<template>
  <div>
    <h1>Accueil</h1>
    <!-- Utilisation directe sans import explicite -->
    <MyButton>Accueil</MyButton>
  </div>
</template>
```

Optimiser les composants avec lazy loading

- Le lazy loading permet de charger les composants uniquement lorsqu'ils sont nécessaires, améliorant ainsi les performances de l'application. Nuxt supporte cette fonctionnalité nativement, ce qui permet de réduire le temps de chargement initial et d'optimiser l'expérience utilisateur.

```
<template>
  <div>
    <h1>Composant en lazy loading</h1>
    <Suspense>
      <template #default>
        <LazyComponent />
      </template>
      <template #fallback>
        <p>Chargement...</p>
      </template>
    </Suspense>
  </div>
</template>

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

const LazyComponent = defineAsyncComponent(() =>
  import('../components/LazyComponent.vue')
)
</script>
```

Slots avancés dans Nuxt

- Les slots dans Nuxt permettent de créer des composants flexibles et réutilisables en définissant des emplacements où du contenu peut être inséré. Les slots scoped et nommés offrent encore plus de contrôle, permettant de passer des données et de structurer le contenu de manière dynamique.

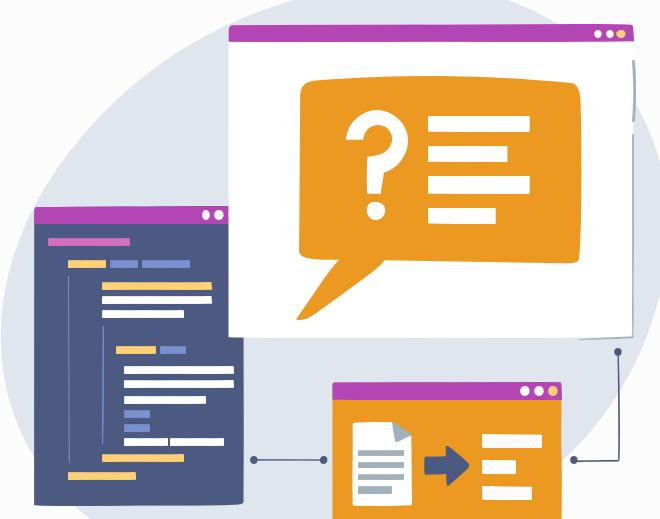
```
<template>
  <div class="card">
    <header class="card-header">
      <!-- Slot nommé "header" -->
      <slot name="header">Titre par défaut</slot>
    </header>
    <div class="card-body">
      <!-- Slot par défaut -->
      <slot>Contenu de la carte</slot>
    </div>
    <footer class="card-footer">
      <!-- Slot scoped pour transmettre des données -->
      <slot name="footer" :info="cardInfo">
        Informations supplémentaires
      </slot>
    </footer>
  </div>
</template>

<script setup>
const cardInfo = { date: new Date().toLocaleDateString() }
</script>

<style scoped>
.card { border: 1px solid #ccc; border-radius: 4px; padding: 1rem; }
.card-header { font-weight: bold; }
.card-footer { font-size: 0.9rem; color: #666; }
</style>
```

04

Assets



Gestion des Assets

- La gestion des assets dans Nuxt est essentielle pour optimiser les performances et maintenir une application réactive. Les assets incluent les images, les polices, et les fichiers statiques nécessaires au fonctionnement de l'application. *Fichier : static/logo.png*
(Fichier image placé dans le dossier static)

```
<template>
  <div>
    <h1>Utilisation des assets</h1>
    <!-- L'image est accessible via "/logo.png" -->
    
  </div>
</template>
```

Optimisation automatique d'images avec Nuxt Image

- Le module Nuxt Image permet d'optimiser automatiquement les images en les redimensionnant et en les compressant selon les besoins. Cela réduit la taille des fichiers et améliore les performances de l'application sans compromettre la qualité visuelle.

```
export default defineNuxtConfig({
  modules: ['@nuxt/image']
})
```

```
<template>
  <div>
    <h1>Nuxt Image</h1>
    <!-- Composant NuxtImg pour l'optimisation d'images -->
    <NuxtImg src="/logo.png" alt="Logo optimisé" width="200" />
  </div>
</template>
```

Gestion des polices et styles globaux

- Nuxt permet d'importer facilement des polices et des styles globaux pour maintenir une apparence cohérente dans toute l'application. En configurant ces ressources dans le fichier `nuxt.config.ts`, vous pouvez les appliquer automatiquement à toutes les pages.

```
export default defineNuxtConfig({
  css: [
    '@/assets/css/global.css' // Fichier CSS global
  ],
  app: {
    head: {
      link: [
        { rel: 'stylesheet', href: 'https://fonts.googleapis.com/css2?family=Roboto:wght@400;700' }
      ]
    }
  }
})
```

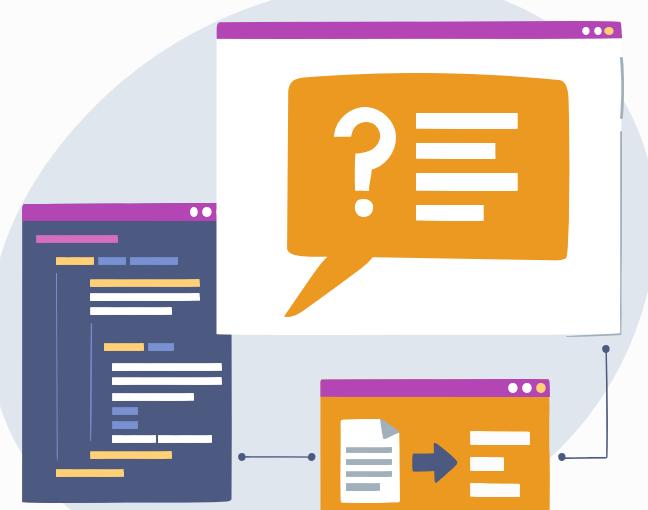
Configuration avancée des assets dans Nuxt

- Pour des besoins spécifiques, Nuxt offre des options de configuration avancées pour les assets. Vous pouvez définir des règles de chargement, de transformation, et de cache pour optimiser davantage les performances et l'expérience utilisateur.

```
export default defineNuxtConfig({
  vite: {
    build: {
      assetsInlineLimit: 4096 // Augmentez la limite d'inclusion des assets
    }
  }
})
```

05

Styles



CSS et Styles dans Nuxt

- Nuxt offre une gestion flexible des styles, permettant d'ajouter du CSS globalement ou localement selon les besoins de votre application. Cela permet de structurer les styles de manière modulaire et maintenable.

Ajouter CSS globalement ou localement

- Vous pouvez ajouter des styles CSS globalement en les important dans le fichier `nuxt.config.ts` ou localement en utilisant des fichiers `.vue` avec des balises `

Utiliser des préprocesseurs CSS (Sass, PostCSS)

- Nuxt supporte nativement les préprocesseurs CSS comme Sass et PostCSS, permettant d'utiliser des fonctionnalités avancées comme les variables, les mixins, et les nestings. Cela facilite la gestion des styles complexes et améliore la maintenabilité du code.

Intégrer TailwindCSS avec Nuxt

- TailwindCSS peut être facilement intégré avec Nuxt pour utiliser une approche utility-first dans la gestion des styles. Cela permet de créer des interfaces utilisateur réactives et modernes avec un contrôle précis sur chaque élément.

Scoped CSS et gestion avancée des styles

- Le scoped CSS permet de limiter la portée des styles à un composant spécifique, évitant ainsi les conflits de styles dans l'application. Nuxt facilite l'utilisation de cette fonctionnalité pour une gestion avancée et modulaire des styles.

Thématisation dynamique avec Nuxt

- Nuxt permet de créer des thèmes dynamiques en utilisant des variables CSS et des configurations conditionnelles. Cela permet de changer l'apparence de l'application en fonction de l'état ou des préférences de l'utilisateur, comme le mode sombre ou clair.

06

State Management



State Management avec Nuxt

- La gestion d'état est cruciale pour les applications modernes, permettant de maintenir une source de vérité unique pour les données. Nuxt offre plusieurs options pour gérer l'état de manière efficace et intuitive.

State management natif dans Nuxt (useState)

- Nuxt propose une solution native pour la gestion d'état avec l'API `useState`. Cela permet de créer et de gérer des états réactifs de manière simple et intuitive, sans avoir besoin de bibliothèques externes.

Partage de l'état entre composants

```
<template>
  <div>
    <button @click="increment">Augmenter depuis le contrôle</button>
  </div>
</template>

<script setup>
const counter = useState('counter', () => 0)
const increment = () => {
  counter.value++
}
</script>
```

Utilisation de computed avec useState

```
<template>
  <div>
    <button @click="increment">Augmenter depuis le contrôle</button>
  </div>
</template>

<script setup>
const counter = useState('counter', () => 0)
const increment = () => {
  counter.value++
}
</script>
```

Réinitialiser et modifier l'état global

```
<template>
  <div>
    <h1>Gestion de l'état global</h1>
    <p>Valeur du compteur : {{ counter }}</p>
    <button @click="increment">Incrémenter</button>
    <button @click="resetCounter">Réinitialiser</button>
  </div>
</template>

<script setup>
const counter = useState('counter', () => 0)
const increment = () => {
  counter.value++
}
const resetCounter = () => {
  counter.value = 0
}
</script>
```

Introduction à Pinia avec Nuxt

- Pinia est une bibliothèque de gestion d'état qui peut être facilement intégrée avec Nuxt. Elle offre une API moderne et flexible pour gérer l'état de manière réactive et modulaire, adaptée aux applications de toutes tailles.

Attention !

- Comme avec onServerPrefetch(), vous pouvez appeler une action de store dans le composable callOnce(). Cela permettra à Nuxt d'exécuter l'action une seule fois et évite de refetcher des données qui sont déjà présentes.

```
<script setup>
const store = useStore()
// we could also extract the data, but it's already present in the store
await callOnce('user', () => store.fetchUser())
</script>
```

07

Les composables Nuxt



Composables Nuxt (use...)

- Les composables sont des fonctions réutilisables qui encapsulent une logique spécifique. Nuxt propose plusieurs composables natifs pour faciliter le développement et améliorer la réutilisabilité du code.

Composables natifs Nuxt (useFetch, useAsyncData)

- Nuxt propose des composables natifs comme `useFetch` et `useAsyncData` pour gérer les requêtes HTTP et les données asynchrones de manière simple et efficace. Ces composables permettent de récupérer des données et de les intégrer facilement dans vos composants.

```
<template>
<div>
  <h1>Données asynchrones avec useAsyncData</h1>
  <div v-if="error">
    <p>Erreur : {{ error.message }}</p>
  </div>
  <div v-else-if="pending">
    <p>Chargement...</p>
  </div>
  <div v-else>
    <ul>
      <li v-for="item in data" :key="item.id">{{ item.name }}</li>
    </ul>
  </div>
</div>
</template>

<script setup>
const { data, error, pending } = await useAsyncData('users', () =>
  $fetch('https://jsonplaceholder.typicode.com/users')
)
</script>
```

Gestion avancée des requêtes HTTP (useFetch)

- Le composable `useFetch` permet de gérer des requêtes HTTP complexes avec des fonctionnalités avancées comme le cache, la révalidation, et la gestion des erreurs. Cela permet d'optimiser les performances et d'améliorer l'expérience utilisateur.

```
<template>
<div>
  <h1>Données asynchrones avec useAsyncData</h1>
  <div v-if="error">
    <p>Erreur : {{ error.message }}</p>
  </div>
  <div v-else-if="pending">
    <p>Chargement...</p>
  </div>
  <div v-else>
    <ul>
      <li v-for="item in data" :key="item.id">{{ item.name }}</li>
    </ul>
  </div>
</div>
</template>

<script setup>
const { data, error, pending } = await useAsyncData('users', () =>
  $fetch('https://jsonplaceholder.typicode.com/users')
)
</script>
```

Composables personnalisés et bonnes pratiques

- Vous pouvez créer vos propres composables pour encapsuler une logique spécifique à votre application. Suivre les bonnes pratiques permet de garantir que vos composables sont réutilisables, testables, et maintenables.

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

export const useCounter = (initialValue = 0) => {
  const count = ref(initialValue)
  const increment = () => count.value++
  const decrement = () => count.value--
  const reset = () => (count.value = initialValue)

  return { count, increment, decrement, reset }
}
```

Gestion des erreurs avec les composables

- Les composables permettent également de gérer les erreurs de manière centralisée et cohérente. Vous pouvez capturer les erreurs, les traiter, et afficher des messages conviviaux à l'utilisateur pour améliorer l'expérience globale.

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

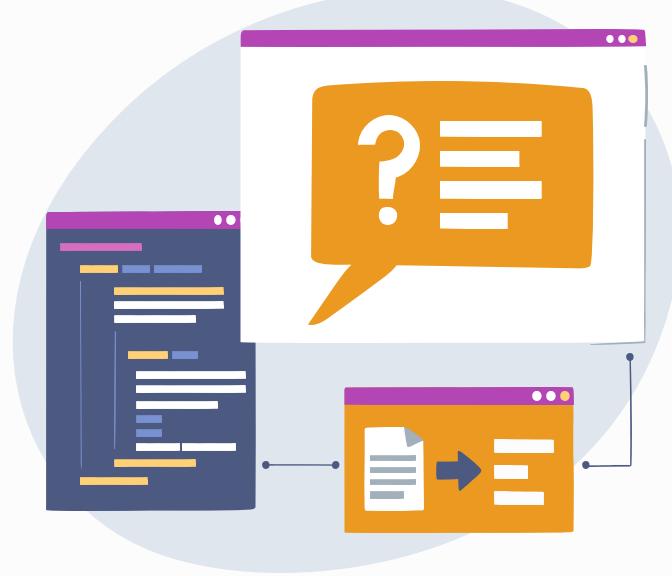
const useCounter = (initialValue = 0) => {
  const count = ref(initialValue)
  const increment = () => count.value++
  const decrement = () => count.value--
  const reset = () => (count.value = initialValue)

  return { count, increment, decrement, reset }
}

export default useCounter
```

08

Allons + loin sur le data fetching



Data fetching avancé

- Nuxt offre des fonctionnalités avancées pour le data fetching, comme la récupération conditionnelle de données, le cache, et la révalidation. Cela permet d'optimiser les performances et de garantir que les données sont toujours à jour.

Utilisation approfondie de useAsyncData

- Le composable `useAsyncData` permet de récupérer des données asynchrones de manière efficace et réactive. Il offre des fonctionnalités avancées comme le cache, la révalidation, et la gestion des erreurs pour optimiser les performances.

```
<template>
  <div>
    <h1>Données asynchrones</h1>
    <div v-if="pending">
      <p>Chargement...</p>
    </div>
    <div v-else-if="error">
      <p>Erreur : {{ error.message }}</p>
    </div>
    <div v-else>
      <ul>
        <li v-for="item in data" :key="item.id">{{ item.title }}</li>
      </ul>
    </div>
  </div>
</template>

<script setup>
const { data, error, pending } = await useAsyncData('posts', () =>
  $fetch('https://jsonplaceholder.typicode.com/posts')
)
</script>
```

Gestion avancée du cache

- Le cache permet de stocker des données temporairement pour améliorer les performances et réduire les temps de chargement. Nuxt offre des outils pour gérer le cache de manière efficace et configurer des stratégies de cache avancées.

```
<template>
<div>
  <h1>Cache avancé avec useAsyncData</h1>
  <div v-if="pending">
    <p>Chargement...</p>
  </div>
  <div v-else-if="error">
    <p>Erreur : {{ error.message }}</p>
  </div>
  <div v-else>
    <ul>
      <li v-for="user in data" :key="user.id">{{ user.name }}</li>
    </ul>
    <p><em>Données mises en cache et réactualisées toutes les 10 secondes.</em></p>
  </div>
</div>
</template>

<script setup>
const { data, error, pending } = await useAsyncData('users', () =>
  $fetch('https://jsonplaceholder.typicode.com/users'),
{
  // Activer la révalidation toutes les 10 secondes
  revalidate: 10
}
)
</script>
```

SWR (Stale-While-Revalidate) dans Nuxt

- SWR est une stratégie de cache qui permet de servir des données mises en cache tout en les révalidant en arrière-plan. Nuxt supporte cette stratégie pour améliorer les performances et garantir que les données sont toujours à jour.

```
<script setup>
const { data, error, pending, refresh } = await useAsyncData('users-swr', () =>
  $fetch('https://jsonplaceholder.typicode.com/users'),
{
  revalidate: 10, // La réactualisation s'effectue toutes les 10 secondes en arrière-plan
  lazy: false,    // Lance immédiatement la requête dès le montage du composant
}
)
</script>
```

Middleware

09



Middleware et gestion d'accès

- Les middlewares dans Nuxt permettent d'exécuter du code avant le rendu de chaque page, offrant ainsi un moyen de gérer l'accès et de valider les requêtes.

```
// middleware/logger.global.js
export default defineNuxtRouteMiddleware((to, from) => {
  console.log(`Navigation de ${from.fullPath} vers ${to.fullPath}`)
})
```

Middleware dans Nuxt : Principes de base

- Les middlewares sont des fonctions qui s'exécutent avant le rendu de chaque page ou groupe de pages. Ils permettent de valider les requêtes, de gérer l'authentification, et d'effectuer des redirections.

```
// middleware/auth.js
export default defineNuxtRouteMiddleware((to, from) => {
  // Supposons que l'état 'user' contient les informations de connexion
  const user = useState('user').value
  if (!user) {
    // Rediriger vers la page de connexion si non authentifié
    return navigateTo('/login')
  }
})
```

Authentification simple avec les middlewares

- Les middlewares peuvent être utilisés pour implémenter une authentification simple, en vérifiant l'état de connexion de l'utilisateur avant de rendre une page. Cela permet de protéger certaines parties de l'application.

```
// middleware/admin.js
export default defineNuxtRouteMiddleware((to, from) => {
  const user = useState('user').value
  // Vérifier si l'utilisateur est connecté et s'il a le rôle 'admin'
  if (!user || user.role !== 'admin') {
    // Redirection vers une page d'erreur ou d'accès refusé
    return navigateTo('/access-denied')
  }
})
```

Middleware avancé : authentification et autorisation

- Pour des besoins plus complexes, les middlewares peuvent être utilisés pour gérer l'authentification et l'autorisation de manière avancée. Cela permet de contrôler l'accès aux ressources en fonction des rôles et des permissions des utilisateurs.

```
// middleware/logger.global.js (déjà vu dans le slide 1)
export default defineNuxtRouteMiddleware((to, from) => {
  console.log(`Navigation de ${from.fullPath} vers ${to.fullPath}`)
})
```

```
<template>
<div>
  <h1>Dashboard Utilisateur</h1>
  <p>Contenu réservé aux utilisateurs authentifiés.</p>
</div>
</template>

<script setup>
definePageMeta({
  middleware: 'auth'
})
</script>
```

Middlewares globaux et spécifiques aux pages

- **Global** : Placez le middleware dans le dossier middleware avec le suffixe .global.js pour qu'il s'applique à toutes les routes.
- **Spécifique** : Utilisez la propriété middleware dans definePageMeta d'une page pour appliquer un middleware particulier.

10 SEO



SEO et Meta-data avec Nuxt

- Le SEO est crucial pour améliorer la visibilité de votre application dans les moteurs de recherche. Nuxt offre des outils pour gérer les métadonnées et optimiser le SEO de manière efficace.

Gestion automatique des balises <head>

- Nuxt permet de gérer automatiquement les balises `<head>` pour chaque page, en définissant des métadonnées comme le titre, la description, et les mots-clés. Cela améliore le référencement et la visibilité de l'application.

```
<template>
<div>
  <h1>Accueil - Mon Application Nuxt</h1>
  <p>Contenu de la page d'accueil...</p>
</div>
</template>

<script setup>
useHead({
  title: 'Accueil - Mon Application Nuxt',
  meta: [
    { name: 'description', content: 'Description de la page d'accueil pour améliorer le référencement' },
    { name: 'keywords', content: 'Nuxt, SEO, application web, référencement' }
  ]
})
</script>
```

SEO avancé avec Nuxt

- Pour des besoins SEO avancés, Nuxt offre des outils pour gérer les redirections, les canonicals, et les hreflangs. Cela permet d'optimiser le référencement pour les applications multilingues et de gérer les contenus dupliqués.

```
<template>
<div>
  <h1>Page SEO Avancé</h1>
  <p>Contenu optimisé pour le SEO avancé.</p>
</div>
</template>

<script setup>
useHead({
  title: 'SEO Avancé – Mon Application Nuxt',
  meta: [
    // Balise canonical
    { rel: 'canonical', href: 'https://www.monsite.com/advanced' },
    // Hreflang pour les versions linguistiques
    { hid: 'hreflang-en', rel: 'alternate', hreflang: 'en', href: 'https://www.monsit
    { hid: 'hreflang-fr', rel: 'alternate', hreflang: 'fr', href: 'https://www.monsit
  ]
})
</script>
```

Open Graph et réseaux sociaux

- Nuxt permet de configurer les métadonnées Open Graph pour améliorer l'apparence des liens partagés sur les réseaux sociaux. Cela permet d'attirer plus de visiteurs et d'améliorer l'engagement.

```
<template>
  <div>
    <h1>Partage sur les réseaux sociaux</h1>
    <p>Cette page est optimisée pour les réseaux sociaux avec Open Graph.</p>
  </div>
</template>

<script setup>
useHead({
  title: 'Partage sur les réseaux sociaux - Mon Application Nuxt',
  meta: [
    { property: 'og:title', content: 'Partage sur les réseaux sociaux' },
    { property: 'og:description', content: 'Découvrez notre contenu optimisé pour le' },
    { property: 'og:image', content: 'https://www.monsite.com/images/og-image.jpg' },
    { property: 'og:url', content: 'https://www.monsite.com/social' },
    { property: 'og:type', content: 'website' }
  ]
})
</script>
```

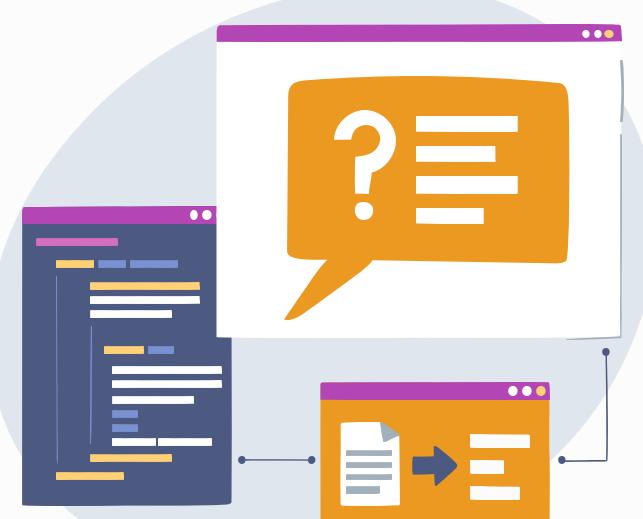
Structurer vos données pour les moteurs de recherche (JSON-LD)

- Nuxt permet de structurer vos données en utilisant le format JSON-LD, ce qui améliore la compréhension de votre contenu par les moteurs de recherche. Cela peut améliorer le classement et la visibilité de votre application.

```
<script setup>
useHead({
  title: 'Produit XYZ - Mon Application Nuxt',
  script: [
    {
      type: 'application/ld+json',
      children: JSON.stringify({
        '@context': 'https://schema.org',
        '@type': 'Product',
        name: 'Produit XYZ',
        image: 'https://www.monsite.com/images/produit-xyz.jpg',
        description: 'Description détaillée du produit XYZ.',
        brand: {
          '@type': 'Brand',
          name: 'Marque ABC'
        },
        offers: {
          '@type': 'Offer',
          priceCurrency: 'EUR',
          price: '29.99',
          availability: 'https://schema.org/InStock'
        }
      })
    }
  ]
})
</script>
```

11

Nuxt Modules



Nuxt Modules et Plugins

- Les modules et plugins dans Nuxt permettent d'étendre les fonctionnalités de l'application de manière modulaire et réutilisable.

Comprendre les modules dans Nuxt

- Les modules sont des extensions qui ajoutent des fonctionnalités spécifiques à Nuxt, comme l'optimisation des images ou la gestion des contenus. Ils peuvent être intégrés facilement via le fichier `nuxt.config.ts`.

```
export default defineNuxtConfig({
  // Activation de modules officiels et tiers
  modules: [
    '@nuxt/image',    // Optimisation des images
    '@nuxt/content'  // Gestion des contenus
  ]
})
```

Intégration des modules officiels Nuxt (Nuxt UI, Nuxt Image, Nuxt Content)

- Nuxt propose plusieurs modules officiels pour étendre les fonctionnalités de l'application, comme Nuxt UI pour les composants d'interface utilisateur, Nuxt Image pour l'optimisation des images, et Nuxt Content pour la gestion des contenus.

```
export default defineNuxtConfig({
  modules: [
    '@nuxt/image',
    '@nuxt/content'
  ],
  image: {
    // Options pour Nuxt Image
    domains: ['example.com'],
    presets: {
      avatar: { modifiers: { format: 'webp', width: 100, hei
    }
  },
  content: {
    // Options pour Nuxt Content
    highlight: {
      theme: 'dracula'
    }
  }
})
```

Création d'un module personnalisé

- Vous pouvez créer vos propres modules pour encapsuler des fonctionnalités spécifiques et les réutiliser dans plusieurs projets. Cela permet de maintenir un code modulaire et réutilisable.

```
export default defineNuxtConfig({
  modules: [
    '@nuxt/image',
    '@nuxt/content'
  ],
  image: {
    // Options pour Nuxt Image
    domains: ['example.com'],
    presets: {
      avatar: { modifiers: { format: 'webp', width: 100, height: 100 } }
    },
    content: {
      // Options pour Nuxt Content
      highlight: {
        theme: 'dracula'
      }
    }
})
```

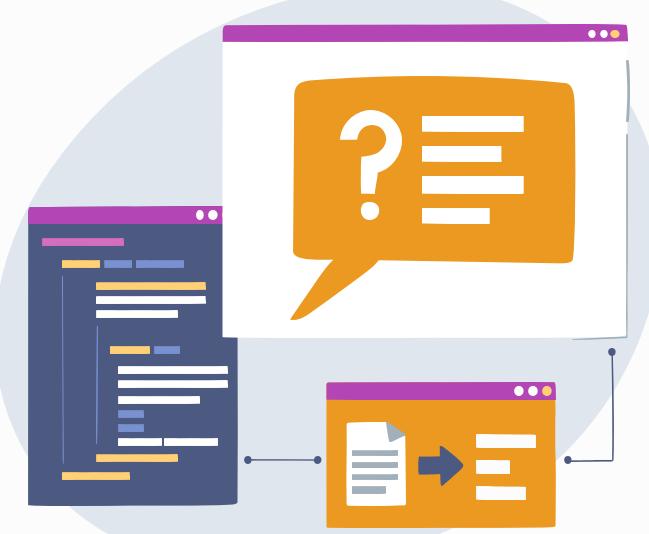
Plugins Nuxt : quand et comment les utiliser

- Les plugins permettent d'ajouter des fonctionnalités spécifiques à l'application, comme des bibliothèques JavaScript ou des configurations personnalisées. Ils peuvent être intégrés facilement via le fichier `nuxt.config.ts`.

```
// plugins/myPlugin.client.ts
export default defineNuxtPlugin((nuxtApp) => {
    // Exemple : Ajout d'une bibliothèque tierce côté client
    nuxtApp.provide('myLibrary', {
        greet: (name: string) => `Bonjour, ${name}!`
    })
})
```

Autres composants

12



NuxtLoading

- La gestion du chargement dans Nuxt est essentielle pour une expérience utilisateur fluide et réactive.
- Nuxt propose des outils pour personnaliser et optimiser le comportement de chargement des composants et des pages.

Personnaliser Nuxt Loading

- Personnalisation de l'indicateur de chargement.
- Définition d'animations, de messages, ou de styles spécifiques.

```
<script setup lang="ts">
  const { progress, isLoading, start, finish, clear } = useLoadingIndicator({
    duration: 2000,
    throttle: 200,
    // This is how progress is calculated by default
    estimatedProgress: (duration, elapsed) => (2 / Math.PI * 100) * Math.atan(elapsed /
duration * 100 / 50)
  })
</script>
```

Client-only components dans Nuxt

- Composants rendus uniquement côté client.
- Réduction de la charge côté serveur et amélioration des performances.

Balise <client-only> et cas d'usages

- Encapsulation de composants ou de code exécutés uniquement côté client.
- Utilisation pour des fonctionnalités spécifiques au client.

```
<template>
  <div>
    <h1>Page avec composant client-only</h1>
    <client-only>
      <ClientWidget />
    </client-only>
  </div>
</template>

<script setup>
import ClientWidget from '~/components/ClientWidget.vue'
</script>
```

Gestion fine du cache avec `useAsyncData`

- Utilisation du composable `useAsyncData` pour optimiser les performances.
- Stratégies de cache personnalisées : cache-first, network-first.

Modifier dynamiquement le cache

- Invalidation et révalidation des données mises en cache.
- Garantir des données à jour même en cas de modifications fréquentes.

```
<script setup lang="ts">
// Exemple simple avec invalidate et revalidate
const { data, refresh, pending } = useAsyncData('user', async () => {
  return await $fetch('/api/user')
})

function updateUserCache() {
  // Invalide le cache pour forcer une nouvelle requête
  invalidate('user')
  // Optionnel : revalidate('user') peut aussi être appelé si besoin
}
</script>

<template>
<div>
  <p v-if="pending">Chargement de l'utilisateur...</p>
  <p v-else>{{ data?.name }}</p>
  <button @click="updateUserCache">Mettre à jour le cache</button>
  <button @click="refresh">Rafraîchir</button>
</div>
</template>
```

Gestion avancée des cookies côté API routes

- Options de gestion des cookies : durée de vie, domaine, chemin.
- Contrôle précis du comportement des cookies pour chaque requête.

```
// server/api/set-cookie.post.ts
export default defineEventHandler(async (event) => {
  const body = await readBody(event)
  if (!body.username) {
    throw createError({ statusCode: 400, statusMessage: 'Username requis' })
  }
  // Création d'un cookie sécurisé
  setCookie(event, 'session_token', 'token_exemple', {
    httpOnly: true,
    secure: true,
    maxAge: 60 * 60 * 24, // 1 jour
    path: '/'
  })
  return { message: 'Cookie défini avec succès' }
})
```

Utilisation des cookies sécurisés

- Attributs HttpOnly et Secure pour protéger les données sensibles.
- Protection contre les attaques courantes.

```
// server/api/secure-login.post.ts
export default defineEventHandler(async (event) => {
    // ... check user credentials ...

    // Définition d'un cookie sécurisé
    setCookie(event, 'secure-auth', 'secure-value', {
        httpOnly: true,
        secure: true,          // Assure que le cookie ne transite qu'en HTTPS
        sameSite: 'strict',   // Protection contre CSRF
        path: '/',
        maxAge: 60 * 60 * 24
    })

    return { status: 'ok' }
})
```

Cookies et authentification persistante

- Utilisation des cookies pour maintenir une authentification persistante.
- Expérience utilisateur fluide entre les sessions.

```
<script setup lang="ts">
import { useCookie } from '#imports'

// Lecture d'un cookie défini côté serveur
const authToken = useCookie('auth-token')

// Exemple : vérifier si l'utilisateur est authentifié
const isAuthenticated = computed(() => !!authToken.value)

// ...
</script>

<template>
<div>
    <p>Utilisateur connecté ? {{ isAuthenticated ? 'Oui' : 'Non' }}</p>
</div>
</template>
```

Authentification

13



Authentification avancée

- L'authentification est cruciale pour protéger les données sensibles et garantir l'accès sécurisé à l'application. Nuxt offre des outils pour implémenter l'authentification de manière avancée et sécurisée.

Authentification JWT avec Nuxt

- Nuxt permet d'implémenter l'authentification JWT pour protéger les données sensibles et garantir l'accès sécurisé à l'application. Cela inclut la génération, la validation, et la gestion des tokens JWT.

```
import jwt from 'jsonwebtoken'

export default defineEventHandler(async (event) => {
  const body = await readBody(event)
  // Exemple de validation simple des identifiants
  if (body.username === 'user' && body.password === 'pass') {
    // Génération d'un token JWT avec une durée de validité de 1 heure
    const token = jwt.sign({ username: body.username, role: 'user' }, 'votre_clé_secrète', { expiresIn: '1h' })
    // Optionnel : Définir le token dans un cookie
    setCookie(event, 'token', token, { httpOnly: true, maxAge: 3600 })
    return { token }
  }
  throw createError({ statusCode: 401, statusMessage: 'Identifiants invalides' })
})
```

Gestion de sessions côté serveur

- Nuxt permet de gérer les sessions côté serveur pour garantir l'accès sécurisé à l'application. Cela inclut la gestion des cookies, des sessions, et des données utilisateur.

```
import jwt from 'jsonwebtoken'

export default defineEventHandler((event) => {
    // Récupérer le token depuis le cookie
    const token = getCookie(event, 'token')
    if (!token) {
        throw createError({ statusCode: 401, statusMessage: 'Non authentifié' })
    }
    try {
        // Vérifier et décoder le token
        const decoded = jwt.verify(token, 'votre_clé_secrète')
        return { user: decoded }
    } catch (err) {
        throw createError({ statusCode: 401, statusMessage: 'Token invalide ou expiré' })
    }
})
```

OAuth et Nuxt : intégration avec GitHub/Google

- Nuxt permet d'intégrer OAuth pour l'authentification avec des fournisseurs comme GitHub ou Google. Cela permet de garantir l'accès sécurisé à l'application en utilisant des comptes externes.

```
export default defineNuxtConfig({
  modules: ['@sidebase/nuxt-auth'],
  auth: {
    strategies: {
      github: {
        clientId: 'VOTRE_GITHUB_CLIENT_ID',
        clientSecret: 'VOTRE_GITHUB_CLIENT_SECRET'
      },
      google: {
        clientId: 'VOTRE_GOOGLE_CLIENT_ID',
        clientSecret: 'VOTRE_GOOGLE_CLIENT_SECRET'
      }
    }
  }
})
```

Stratégies d'authentification avancées

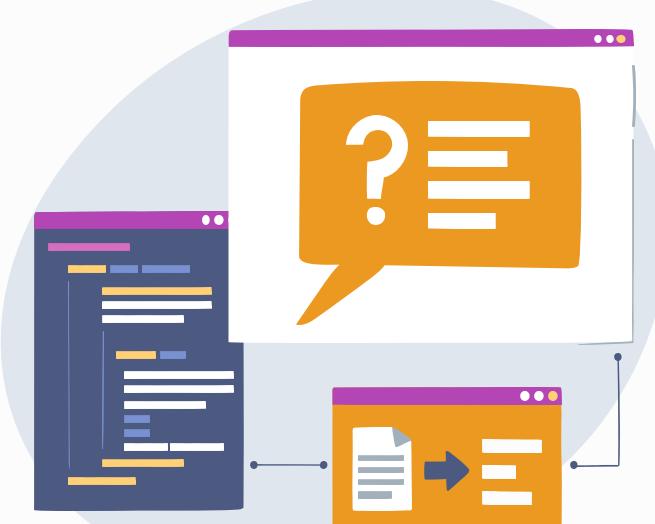
- Nuxt permet d'implémenter des stratégies d'authentification avancées pour répondre à des besoins spécifiques. Cela inclut l'authentification multi-facteurs, la gestion des rôles et des permissions, et la protection des données sensibles.

```
import jwt from 'jsonwebtoken'

export default defineNuxtRouteMiddleware((to, from) => {
  const token = useCookie('token').value
  if (!token) {
    return navigateTo('/login')
  }
  try {
    // Décoder le token pour vérifier le rôle de l'utilisateur
    const decoded = jwt.verify(token, 'votre_clé_secrète')
    // Exemple : restreindre l'accès aux pages réservées aux administrateurs
    if (to.meta.requiresAdmin && decoded.role !== 'admin') {
      return navigateTo('/unauthorized')
    }
  } catch (err) {
    return navigateTo('/login')
  }
})
```

14

API Route



API Routes dans Nuxt (Serveur)

- Les API routes permettent de créer des points de terminaison côté serveur pour gérer les requêtes et les données de manière efficace. Nuxt offre des outils pour créer et gérer les API routes de manière simple et flexible.

Introduction aux routes API

- Nuxt permet de créer des routes API pour gérer les requêtes et les données de manière efficace. Cela inclut la création de points de terminaison, la gestion des requêtes, et la réponse aux clients.

```
// server/api/hello.get.ts
export default defineEventHandler((event) => {
  return { message: 'Bonjour depuis l\'API Nuxt!' }
})
```

Gestion avancée des requêtes API

- Nuxt permet de gérer les requêtes API de manière avancée, en utilisant des techniques comme la validation des données, la gestion des erreurs, et la protection des points de terminaison. Cela permet de garantir la sécurité et la fiabilité des API.

```
// server/api/data.ts
export default defineEventHandler(async (event) => {
  if (event.req.method === 'GET') {
    // Traitement pour GET
    return { status: 'success', data: [1, 2, 3] }
  } else if (event.req.method === 'POST') {
    const body = await readBody(event)
    // Validation simple : vérifier que body contient une propriété 'name'
    if (!body.name) {
      throw createError({ statusCode: 400, statusMessage: 'Le champ "name" est requis' })
    }
    // Traitement pour POST
    return { status: 'success', message: `Bonjour ${body.name}` }
  } else {
    throw createError({ statusCode: 405, statusMessage: 'Méthode non autorisée.' })
  }
})
```

Créer une API REST dans Nuxt

- Nuxt permet de créer une API REST complète pour gérer les requêtes et les données de manière structurée et cohérente. Cela inclut la création de points de terminaison, la gestion des ressources, et la réponse aux clients.

```
// server/api/articles/index.get.ts
export default defineEventHandler(() => {
  // Exemple de données articles
  return [
    { id: 1, title: 'Article 1', content: 'Contenu de l\'article 1' },
    { id: 2, title: 'Article 2', content: 'Contenu de l\'article 2' }
  ]
})
```

Authentification sécurisée sur les API Nuxt

- Nuxt permet de sécuriser les API en utilisant des techniques comme l'authentification, l'autorisation, et la validation des requêtes. Cela permet de garantir que seules les requêtes autorisées peuvent accéder aux données sensibles.

```
// server/api/secure-data.get.ts
export default defineEventHandler((event) => {
  // Exemple de vérification d'un token d'authentification dans les headers
  const authHeader = getHeader(event, 'authorization')
  if (!authHeader || authHeader !== 'Bearer mon-token-secret') {
    throw createError({ statusCode: 401, statusMessage: 'Non autorisé' })
  }
  return { data: 'Données sécurisées accessibles uniquement aux utilisateurs authentifiés' }
})
```

Gestion avancée des erreurs côté API

- Nuxt permet de gérer les erreurs côté API de manière avancée, en utilisant des techniques comme la validation des données, la gestion des exceptions, et la réponse aux clients avec des messages d'erreur clairs et informatifs. Cela permet de garantir la fiabilité et la robustesse des API.

```
// server/api/error-demo.get.ts
export default defineEventHandler(async (event) => {
  try {
    // Simuler une opération susceptible d'échouer (ex. accès à une base de données)
    throw new Error('Erreur simulée lors de l\'accès aux données.')
  } catch (err) {
    // Renvoi d'une erreur structurée avec un message clair
    throw createError({
      statusCode: 500,
      statusMessage: 'Erreur interne du serveur. Veuillez réessayer plus tard.',
      data: { error: err.message }
    })
  }
})
```

Advanced Routing

15



Advanced Routing

- Le routage avancé dans Nuxt permet de gérer des scénarios complexes et de personnaliser le comportement des routes pour répondre à des besoins spécifiques.

Routage personnalisé avancé avec Nuxt

- Nuxt permet de créer des routes personnalisées pour répondre à des besoins spécifiques, comme des routes dynamiques complexes ou des redirections conditionnelles. Cela offre une flexibilité accrue pour structurer votre application selon vos exigences.

```
export default defineNuxtPlugin((nuxtApp) => {
  const router = useRouter()
  // Ajout d'une route personnalisée vers une page spécifique
  router.addRoute({
    path: '/custom-route',
    name: 'CustomRoute',
    // Le composant est importé dynamiquement
    component: () => import '~/pages/custom.vue'
  })
})
```

Lazy-loading de routes complexes

- Le lazy-loading permet de charger les composants de route uniquement lorsqu'ils sont nécessaires, optimisant ainsi les performances de l'application. Nuxt facilite cette approche pour les routes complexes, améliorant le temps de chargement initial.

```
import { defineAsyncComponent } from 'vue'

export default defineNuxtPlugin((nuxtApp) => {
  const router = useRouter()
  // Ajout d'une route dont le composant est chargé en lazy-loading
  router.addRoute({
    path: '/lazy-route',
    name: 'LazyRoute',
    component: defineAsyncComponent(() => import '~/pages/lazy-route.vue')
  })
})
```

Utilisation de ZOD

```
import { z } from 'zod'

const userSchema = z.object({
  name: z.string().default('Guest'),
  email: z.string().email(),
})

export default defineEventHandler(async (event) => {
  const result = await readValidatedBody(event, body => userSchema.safeParse(body))

  if (!result.success)
    throw result.error.issues

  // User object is validated and typed!
  return result.data
})
```