Vue

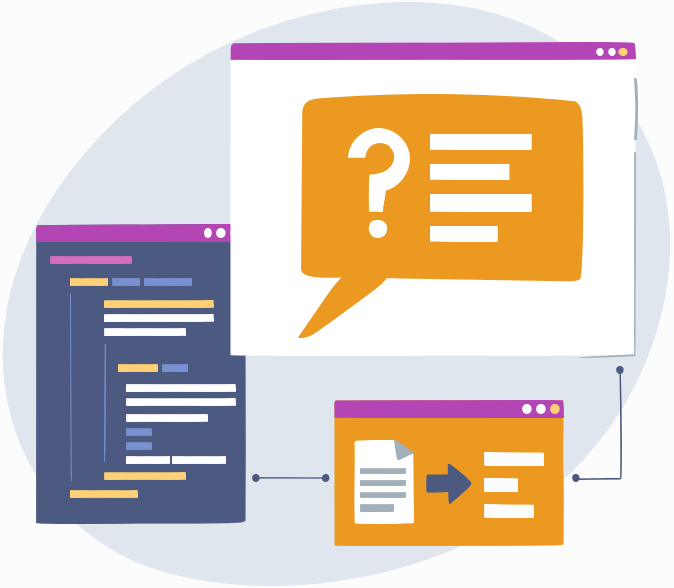
3

Support réalisé par Pierrick Hauguel

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/expanser 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 aplatir des tableaux imbriqués et appliquer une fonction, puis aplatir 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

TypeScript

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.

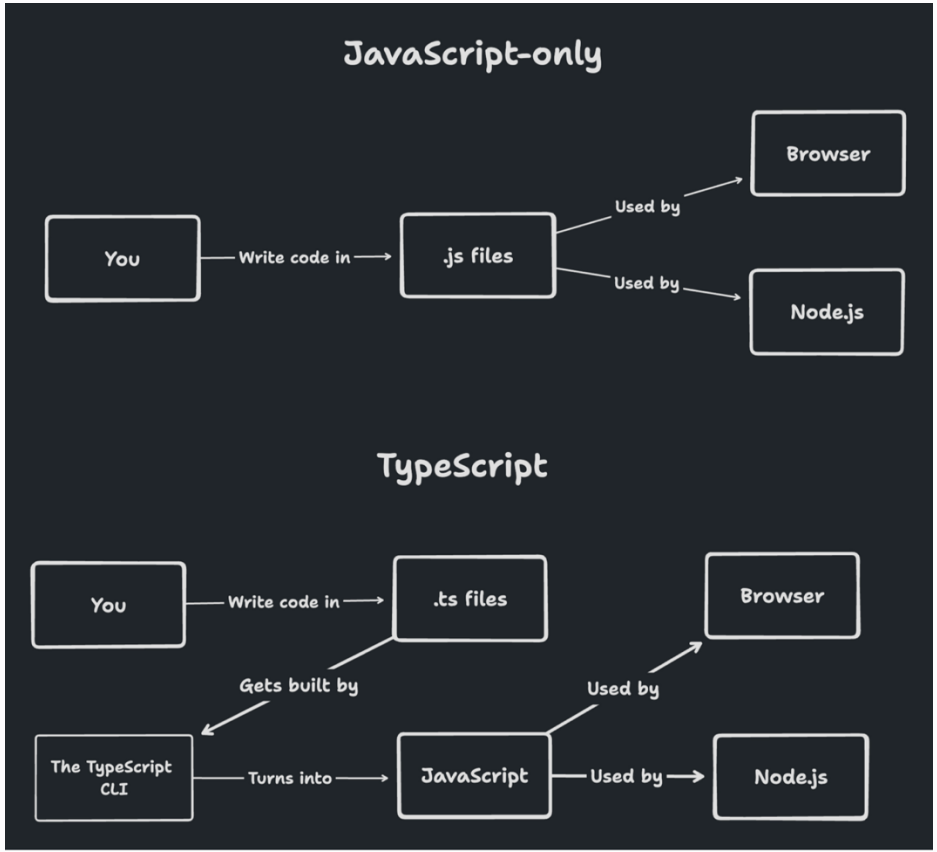
Le choix d’utiliser ou non TypeScript dépend de vos préférences en termes de sécurité et de flexibilité.

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 doSometing = (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/mattpocockuk

https://www.youtube.com/@mattpocockuk

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.

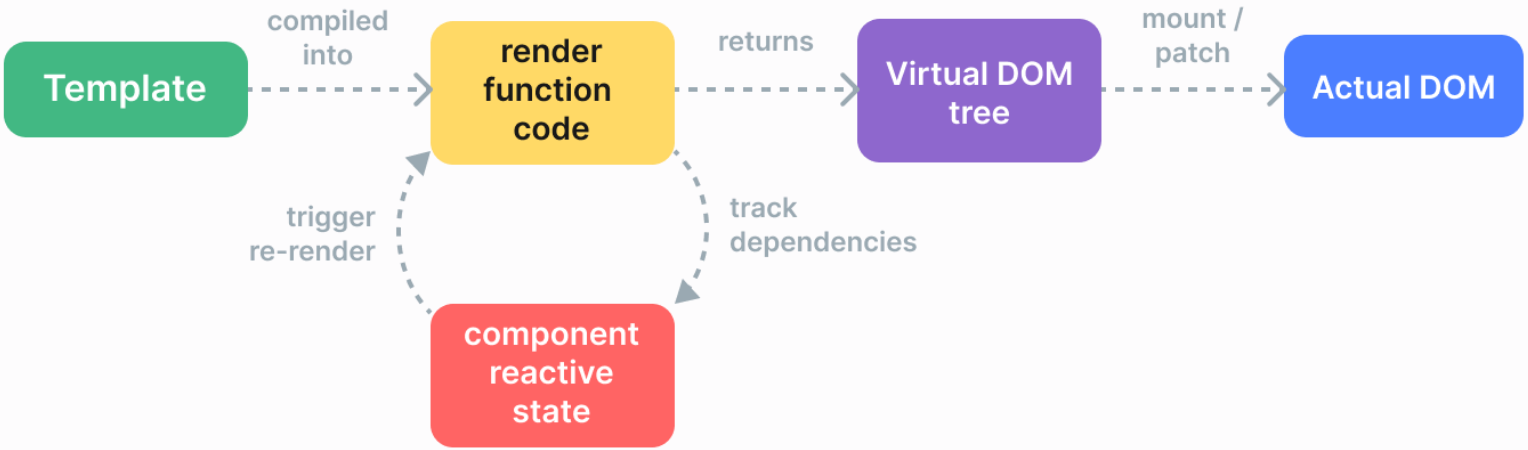
Comparaison avec React et Angular

- **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.

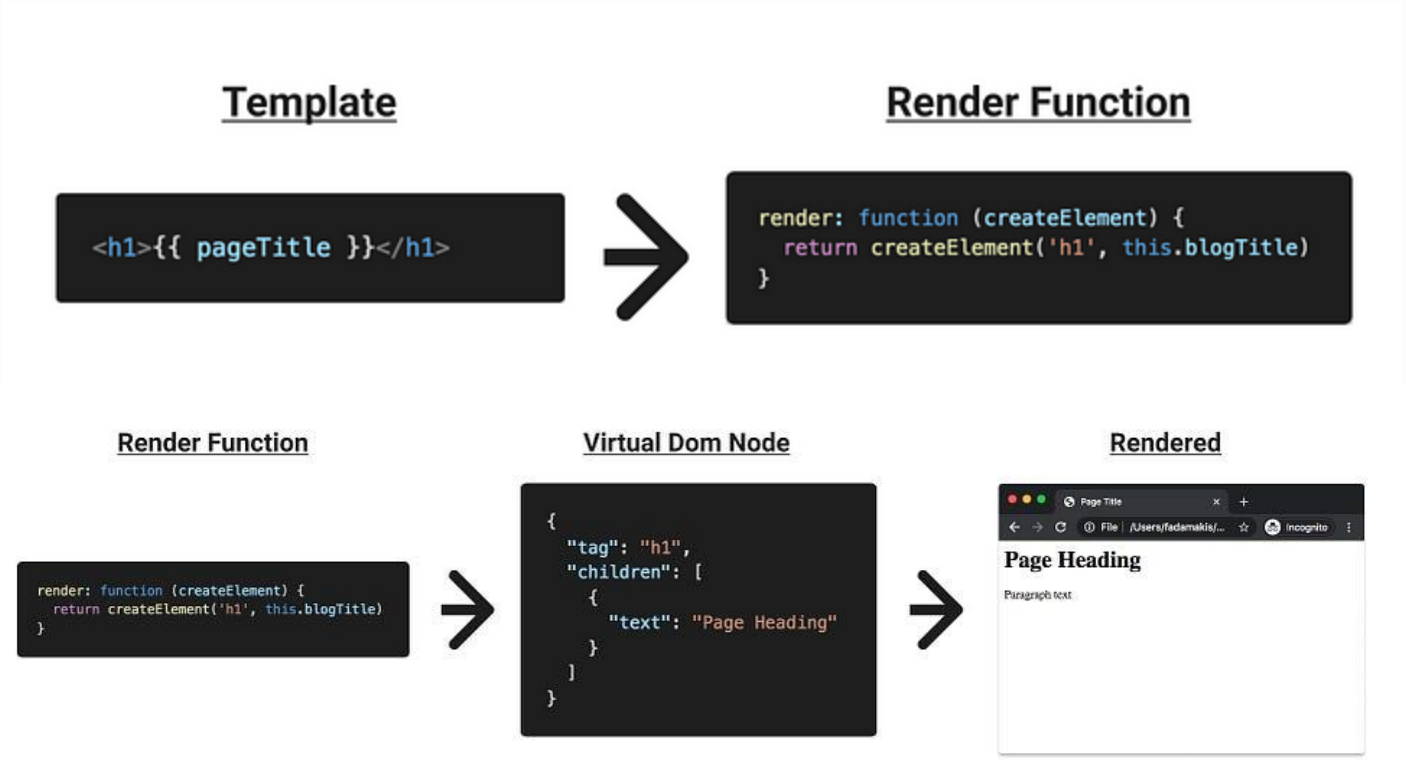
Conclusion et Adoption de Vue.js

- **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

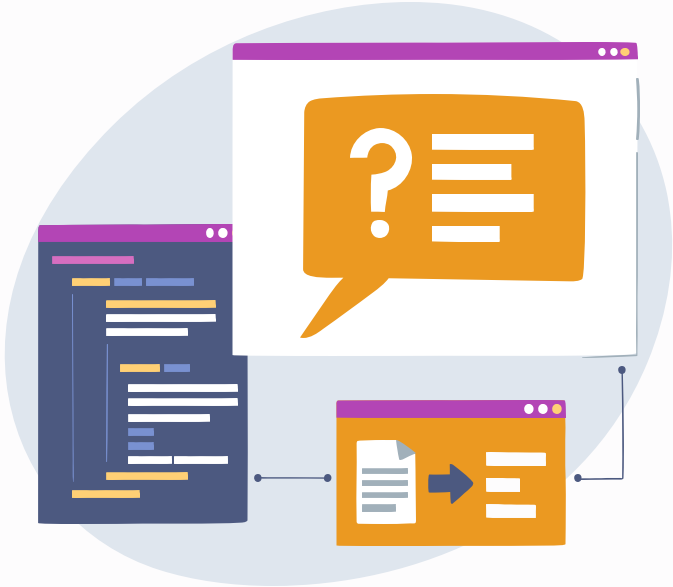


Render Pipeline



04

Installation de Vue

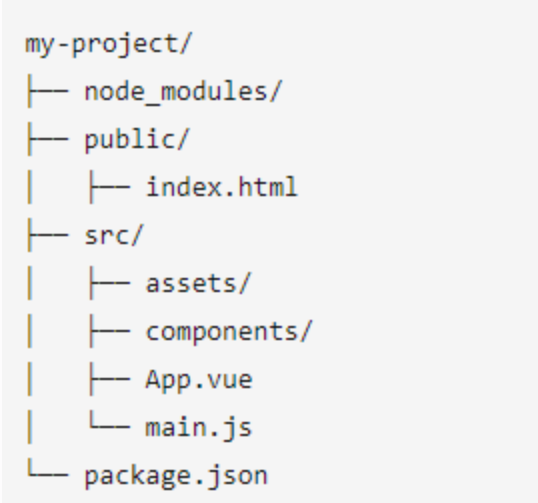


Installation de Vue CLI

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.



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

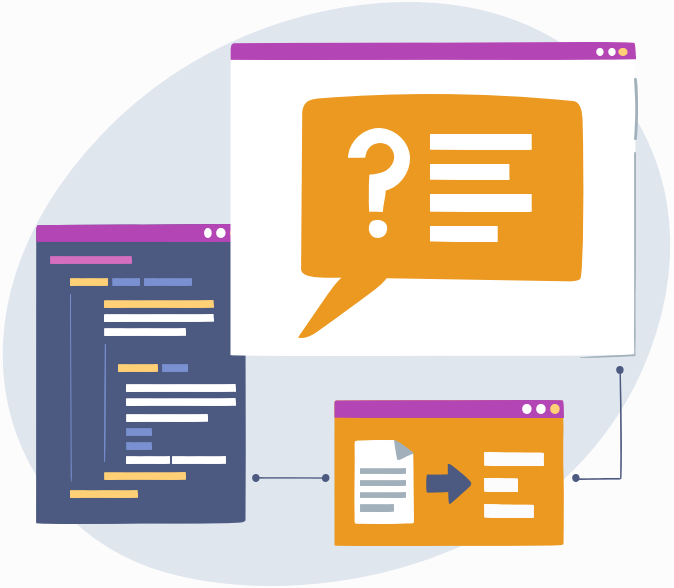
Les slots sont une fonctionnalité puissante de Vue.js permettant d'insérer du contenu dans un composant enfant depuis un composant parent. Ils sont utiles pour créer des composants réutilisables et flexibles.

```
<!-- 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>
```

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

```
class RefImpl<T> {
  private _value: T
  private _rawValue: T

  public dep?: Dep = undefined
  public readonly __v_isRef = true

  constructor(value: T, public readonly __v_isShallow: boolean) {
    this._rawValue = __v_isShallow ? value : toRaw(value)
    this._value = __v_isShallow ? value : toReactive(value)
  }

  get value() {
    trackRefValue(this)
    return this._value
  }

  set value(newVal) {
    newVal = this.__v_isShallow ? newVal : toRaw(newVal)
    if (hasChanged(newVal, this._rawValue)) {
      this._rawValue = newVal
      this._value = this.__v_isShallow ? newVal : toReactive(newVal)
      triggerRefValue(this, newVal)
    }
  }
}
```

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>
```

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>
```

09

Les cycles de vie

Introduction aux hooks de cycle 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.

Hook 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>
```

Hook 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>
```

Hook 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);
});
```

Hook destroyed()

Le hook destroyed() est appelé juste avant que le composant soit détruit. Il est utilisé pour effectuer des nettoyages comme la désinscription d'événements ou l'arrêt de timers.

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

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

const message = ref('Composant actif');

onUnmounted(() => {
  console.log('Composant détruit');
});
</script>
```

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);
  }
});
</script>
```

Gardes de navigation avant et après

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

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

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

Champs Meta de route

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

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

Utilisation des champs Meta

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

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

Récupération des données avec Vue Router

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

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

Récupération des données avant la navigation

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

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

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

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

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

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>
```

Utilisation de watchEffect

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

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

const data = ref(0);

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

Watchers et Réactivité

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

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

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

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

Syntaxe de base de watch

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

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

const message = ref('Hello');

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

Watchers multiples

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

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

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

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

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

Surveillance des propriétés imbriquées

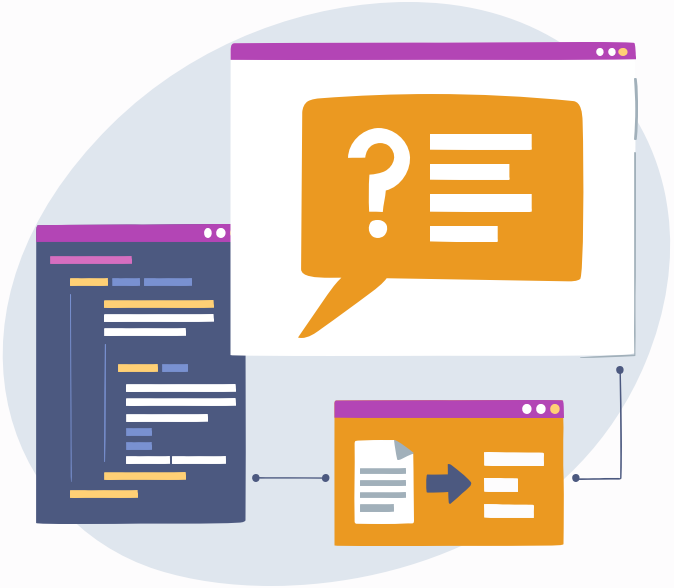
Les watchers peuvent surveiller des propriétés imbriquées d'objets réactifs, en utilisant une fonction pour accéder à la propriété à surveiller.

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

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

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

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>
```

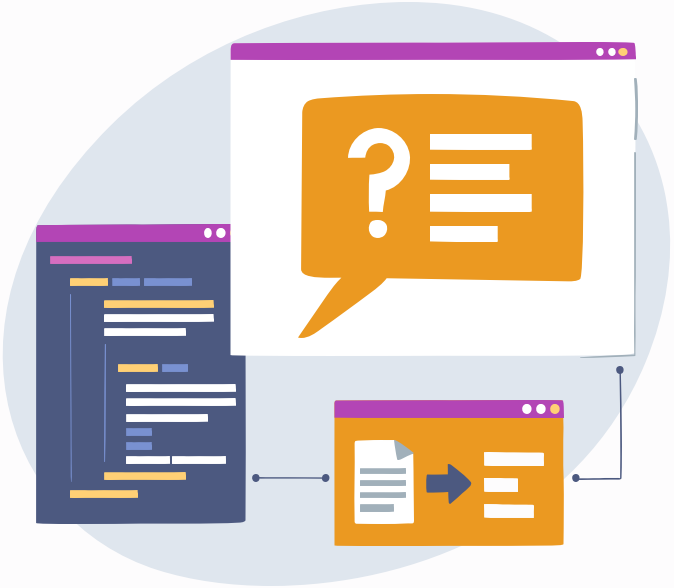
Computed Properties et cache

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.

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

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

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

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

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

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

Modèle de composant v-model

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

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

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

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

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

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

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

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

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

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.

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

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

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

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

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

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

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

Création de types TypeScript à partir de schémas Zod

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

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

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

Exemples pratiques d'inférence de schéma

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

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

type Product = z.infer<typeof productSchema>;

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

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>
```

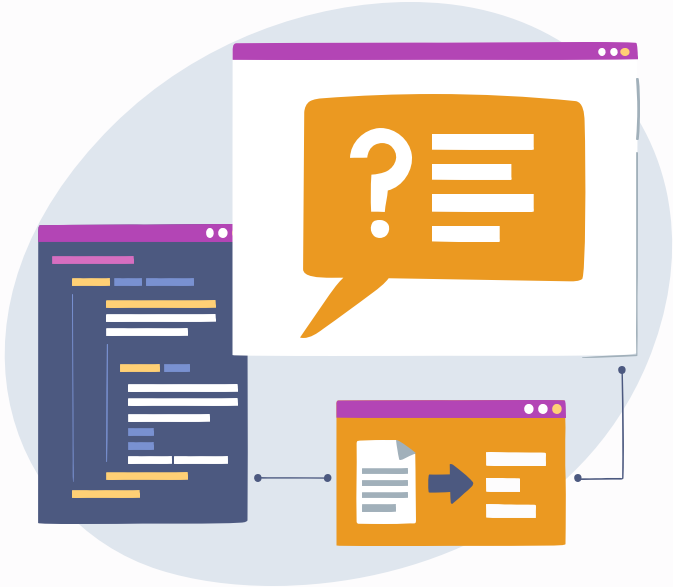
Utilisation de storeToRefs dans Pinia

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>
```

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'Ecriture

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>
```

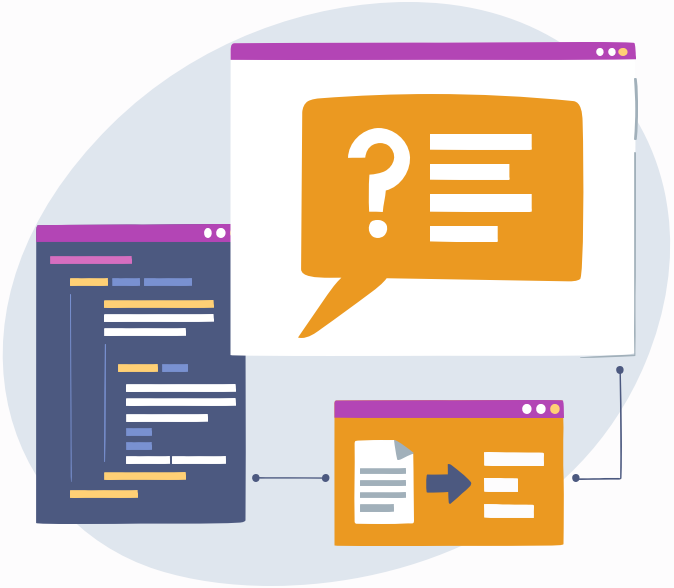
Pagination et Requêtes Infinies avec Tanstack Query

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.

```vue
<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

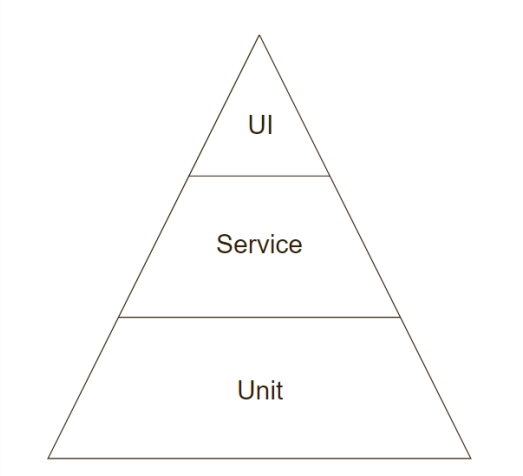
Introduction au test moderne de vue

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.

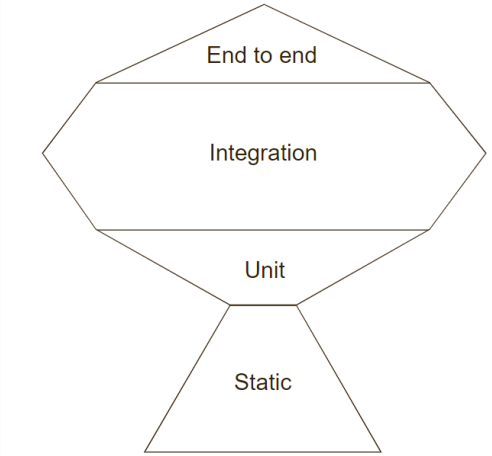
Présentation de la pyramide de tests

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

Introduction à Vitest

Vitest est un framework de test moderne et rapide conçu pour les applications front-end, particulièrement bien adapté aux projets utilisant Vite.

```
// Un test simple avec Vitest
import { test, expect } from 'vitest';

test('addition fonctionne correctement', () => {
  expect(1 + 1).toBe(2);
});
```

Configuration de Vitest

Configurez Vitest en ajoutant une section de configuration dans vite.config.js.

```
import { defineConfig } from 'vite';
import vue from '@vitejs/plugin-vue';
import { defineConfig } from 'vitest/config';

export default defineConfig({
  plugins: [vue()],
  test: {
    globals: true,
    environment: 'jsdom',
  },
});
```

Intégration de Vitest avec Vue 3

Vitest s'intègre facilement avec Vue 3 pour tester des composants et des fonctionnalités réactives.

```
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');
  });
});
```

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()],
});
```

Introduction aux tests unitaires

Les tests unitaires permettent de vérifier le bon fonctionnement des unités individuelles de votre code.

```
// Un test unitaire simple
import { test, expect } from 'vitest';

test('multiplication fonctionne correctement', () => {
  expect(2 * 2).toBe(4);
});
```

Création de tests unitaires avec Vitest

Créez des tests unitaires pour valider le comportement des fonctions et des composants individuels.

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

test('sum additionne correctement deux nombres', () => {
  expect(sum(1, 2)).toBe(3);
});
```

Utilisation des assertions de base

Vitest fournit des assertions de base pour vérifier les résultats des tests.

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

test('les chaînes de caractères sont identiques', () => {
  expect('vue').toBe('vue');
});
```

Tests de fonctions JavaScript simples

Testez des fonctions JavaScript simples pour vous assurer qu'elles retournent les résultats attendus.

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

test('isEven détecte les nombres pairs', () => {
  expect(isEven(2)).toBe(true);
  expect(isEven(3)).toBe(false);
});
```

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.

```vue
<!-- MyComponent.vue -->
<template>
  <div>Hello {{ name }}</div>
</template>

<script setup>
defineProps(['name']);
</script>
```

```js
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.

```vue
<!-- MyComponent.vue -->
<template>
  <div>
    <slot></slot>
  </div>
</template>
```

```js
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.

```vue
<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>
```

```js
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.

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'],
};
```