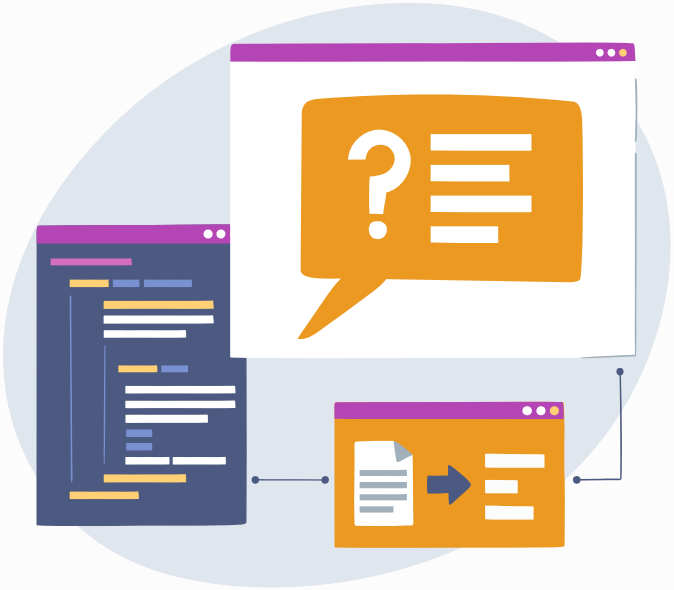
Vue 3

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.

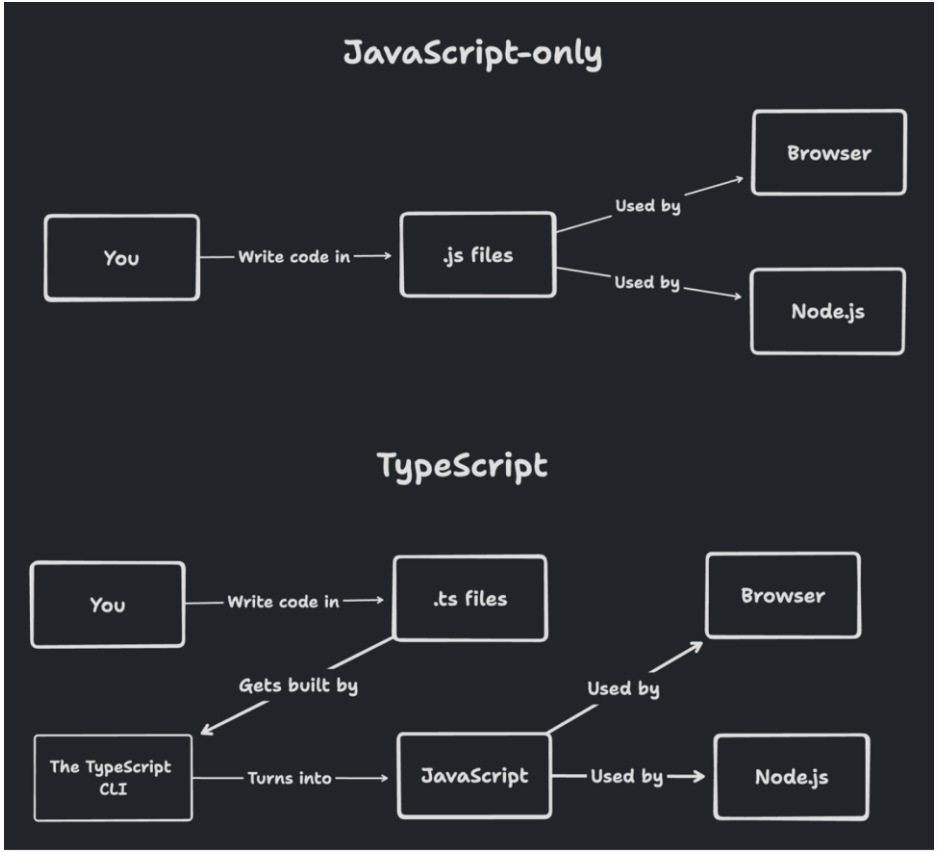
Le choix d'utiliser ou non TypeScript dépend de vos préférences en termes de sécurité et de flexibilité.

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.

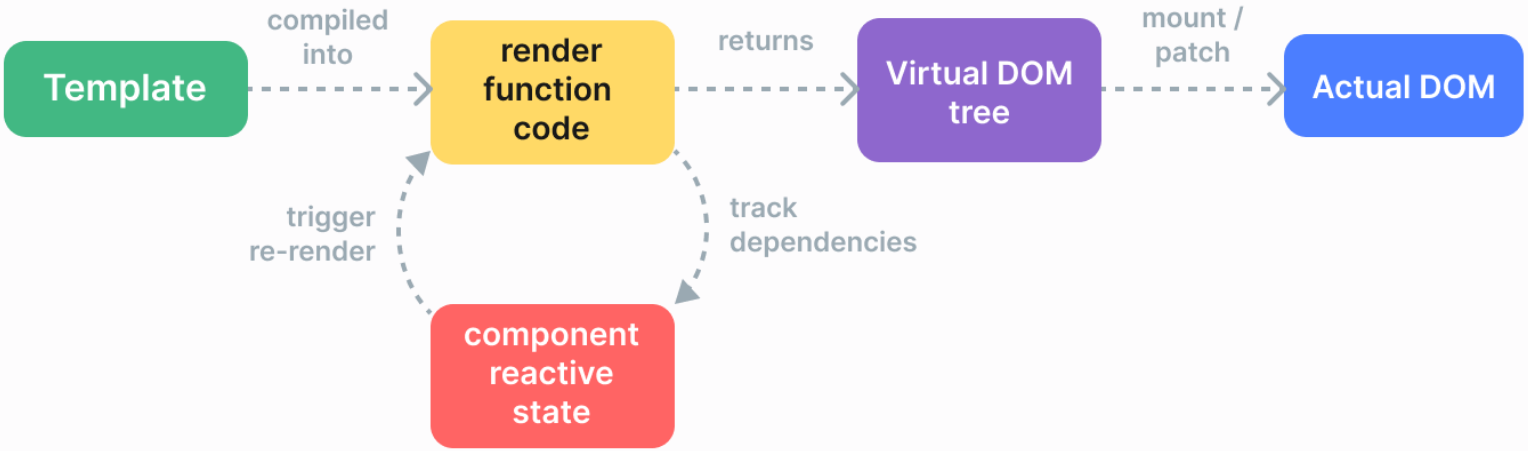
Comparaison avec React et Angular

- **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.

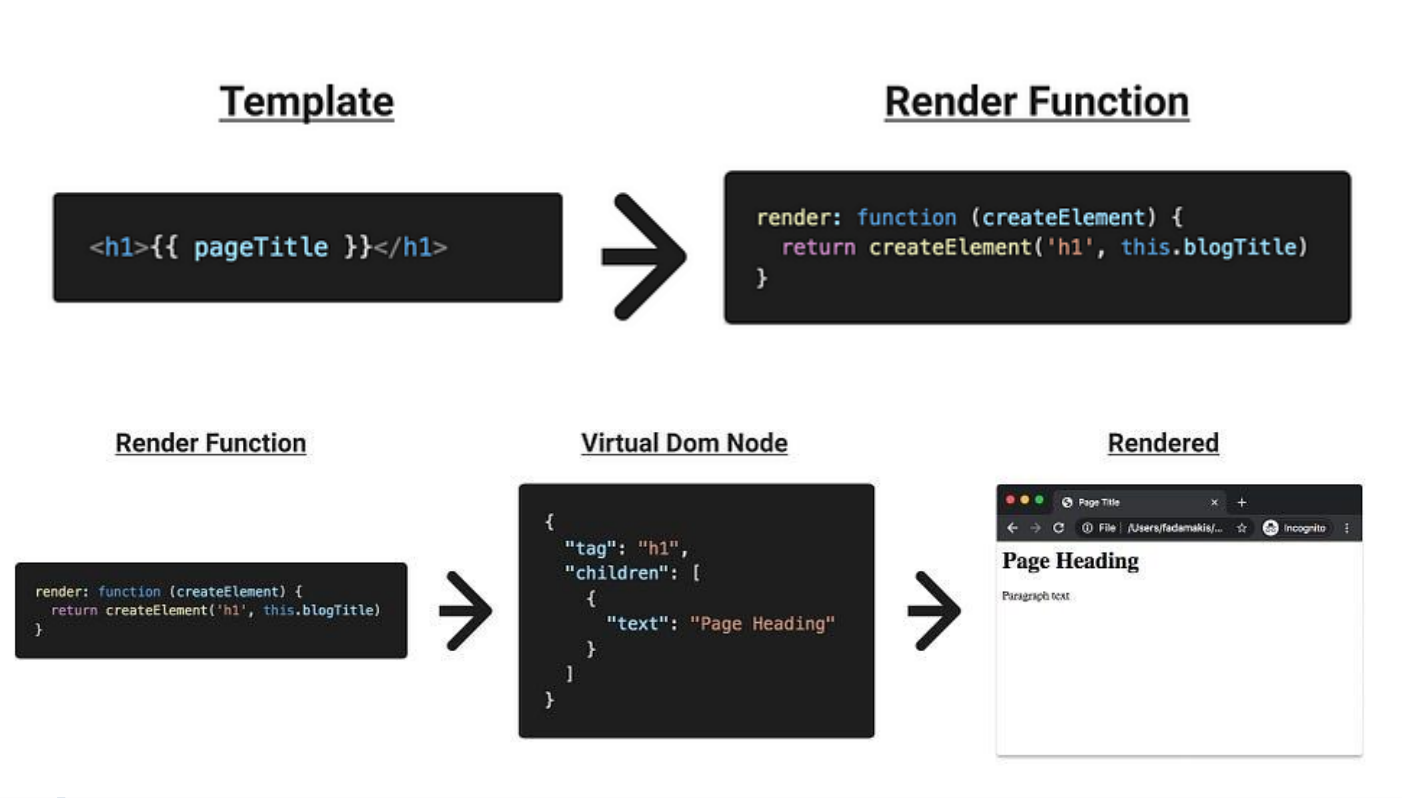
Conclusion et Adoption de Vue.js

- **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.

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

Vue Devtools

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

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

05

Les Composants

Introduction aux composants

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

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

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

Props dans les composants

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

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

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

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

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

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

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

Utiliser des modules ES6

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

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

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

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

Template Syntax

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

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

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

Introduction à la syntaxe des templates

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

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

Expressions dans les templates

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

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

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

Utilisation des Slots

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

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

<script setup>
</script>
```

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

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

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

```ts
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>
```