{ Web App avec Vue 3 }

Introduction

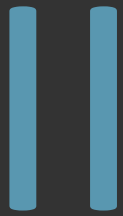
Les Frameworks JavaScript

Il existe énormément de UI Frameworks (ou client Frameworks) JavaScript

Pourquoi Vue

The Progressive JavaScript Framework

An approachable, performant and versatile framework for building web user interfaces.



Fonctionnement Général

Introduction au Composant

- L’architecture d’une application Vue (comme tout Framework JS moderne) repose sur des **composants**.

- Un **composant** est un élément d’interface indépendant. Il s’agit d’une brique permettant de **composer** une application Vue.

- Le composant s’utilise comme une balise HTML.

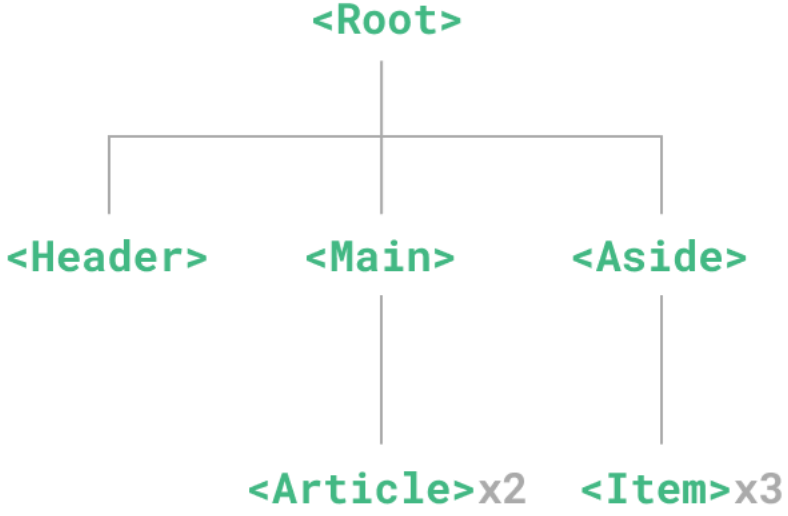
- Un **composant** a une apparence, un comportement et un nom unique. Il regroupe donc HTML, CSS et JavaScript.

Structure d'une Application Vue

C'est en « assemblant » les composants que l'on créé une application Vue.

La structure finale de l'application sera une arborescence de composants.

Le composant « Root » illustré est le composant principal et permet d'initialiser l'application.



Composant SFC

Pour créer un composant, il faut donc du JavaScript, du HTML et du CSS.

En VueJS, les trois peuvent être réunis en un seul fichier **.vue.**

Les composants déclarés en un fichier sont qualifiés de **S**ingle **F**ile **C**omponent (**SFC**).

Il est possible d'utiliser du TypeScript dans un fichier .vue à la place du JavaScript.

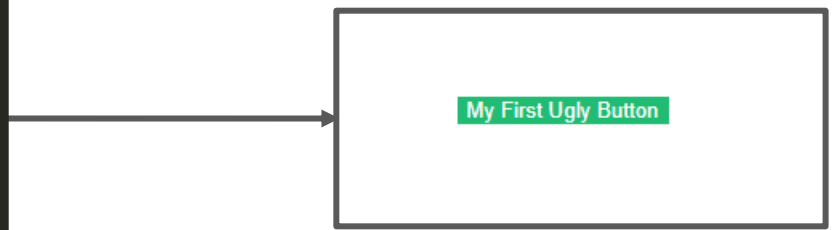
Exemple d'un SFC

```vue
<template>
  <!-- Le HTML de votre composant bouton -->
  <button>My First Ugly Button</button>
</template>

<script setup lang="ts">
/**
 * Code TypeScript pour gérer le comportement du bouton :
 * Réagir au click, au survol de la souris etc.
 */
</script>

<style>
button {
  /**
  * Le style de votre composant
  */
  background-color: rgb(34, 187, 116);
  color: White;
  border: none;
}
</style>
```

<template> pour le HTML

<script> pour le TypeScript

<style> pour le CSS



Démarrage de l'application

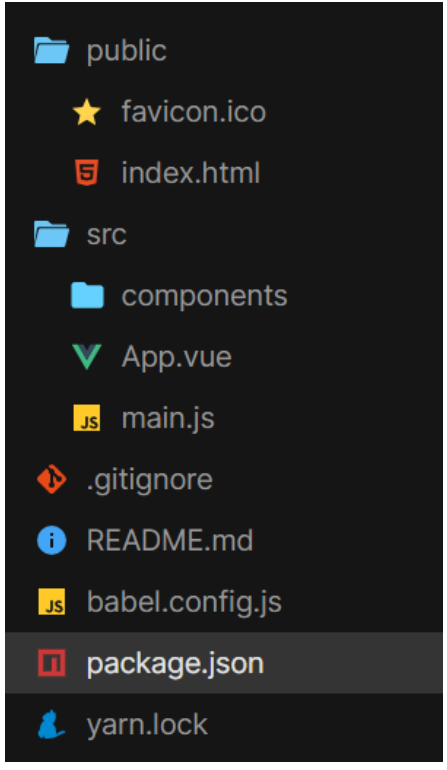
- Une application Vue doit être initialisée afin de pouvoir utiliser ses composants.

- VueJS s'initialise en partant d'un composant principale, le composant **root**.

- Le composant root sera le « point de départ » de votre interface.

Démarrage de l'application

Il vous faut donc d'une page index.html, d'un composant root ainsi que d'un script pour initialiser le Framework.



Sachant qu'il s'agit d'une SPA (Single Page Application) vous n'avez besoin que d'une seule page HTML

Composant Root App.vue

```vue
<template>
  <my-button></my-button>
</template>

<script setup>
import MyButton from "./components/Button.vue";
</script>

<style>
#app {
  text-align: center;
  margin-top: 60px;
}
</style>
```

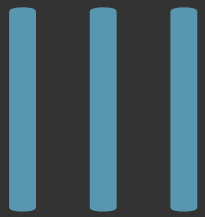
Le composant « Root » est toujours nommé App.vue par convention

Script de démarrage main.js

```
import { createApp } from "vue";
/**
 * App est le composant Root
 */
import App from "./App.vue";

/**
 * La fonction createApp initialise l'application vue à partir du composant root
 * ainsi qu'à partir d'une div.
 */
createApp(App).mount("#app");
```

Fichier index.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width,initial-scale=1.0" />
    <link rel="icon" href="favicon.ico" />
    <title>Ma Page</title>
  </head>
  <body>

    <!-- DIV dans laquelle l'application Vue doit s'afficher -->
    <div id="app"></div>

  </body>
</html>
```



Core API

Template interpolation

Avec Vue est il est possible d'interagir avec le JavaScript depuis le HTML.

Il est possible d'injecter le contenu d'une variable ou le résultat d'une expression JavaScript à l'intérieur du template : c'est l'**interpolation**.

```
<template>
  <button> {{ content }} </button>
</template>

<script setup lang="ts">
const content = "My Ugly Button";
</script>
```

L'usage des doubles accolades ou *brackets* permet d'injecter des valeurs dynamiquement

Réactivité

Il est primordiale que la vue se mette à jour à chaque fois que le contenu d'une variable change.

Vue doit donc être en mesure de détecter les changements de valeurs.

La réactivité ou *reactivity* fait référence à la capacité de Vue à réagir au changement d'une variable.

Pour bénéficier de la réactivité, il faut initialiser ses variables avec une fonction spéciale : **ref(**valeur**)**

Réactivité

La fonction **ref** initialise votre variable avec un objet qui contient votre valeur.

L'inconvénient est que pour accéder à votre valeur, vous devez systématiquement passer par la propriété **value** de l'objet retourné.

```
<template>
  <button>{{ content }}</button>
</template>

<script setup lang="ts">
import {ref} from "vue";

let content = ref("My");
content.value += " Button";
</script>
```

Réactivité

Dans le template la variable se manipule comme avant. La valeur est automatiquement extraite.

Il n’est pas nécessaire (et impossible) de passer par la propriété .value.

```
<template>
  <button>{{ content }}</button>
</template>
```

Vue via la fonction **ref()** va retourner un objet **Proxy** qui permet d’observer les changements effectués sur la propriété **value.**

Usage de fonction

Toutes les fonctions déclarées dans le script sont également accessible et invocables depuis le template.

```
<template>
  <strong> {{ name() }}</strong>
</template>

<script setup lang="ts">
let firstName = "John";
let lastName = "Doe";

function name() {
  return `${firstName} ${lastName}`;
}
</script>
```

Dans l'exemple, si les variables changent, l'interface ne se mettra pas à jour et ce même si les variables firstName et lastName sont initialisées avec **ref()**

Computed Properties

Il est possible de définir des **computed properties**, des variables dont le contenu est le résultat d'une expression JavaScript

```
<template>
  <strong> {{ name }}</strong>
</template>

<script setup lang="ts">
import { ref, computed } from "vue";

let firstName = ref("John");
let lastName = ref("Doe");

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

En utilisant la fonction **computed**, l'interface se mettra à jour si firstName ou lastName change.

Event Handling

A l'aide du mot clé **v-on** ou du symbole **@**, il est possible de s'abonner à tout évènement émis par un élément du DOM

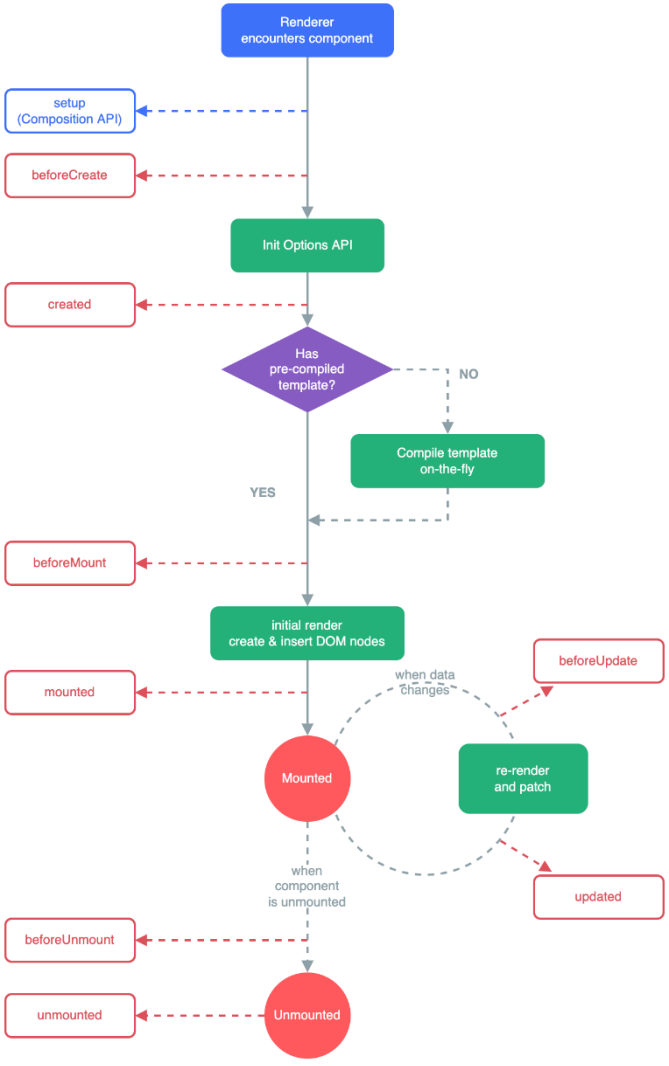
```vue
<template>
  <button @click="onBtnClick" @mouseover="onMouseOver()">
    Say Hi
  </button>
</template>

<script setup lang="ts">
function onBtnClick(e: PointerEvent) {
  alert("Hello World");
  console.log(e);
}
function onMouseOver() {}
</script>
```

Il n'est pas nécessaire d'appeler une fonction, tout expression JavaScript est acceptée

Component Lifecycle

Un composant Vue passe par une succession d’étapes depuis son initialisation jusqu’à sa destruction.

Event Handling



Lifecycle Hooks

Les **Lifecycle Hooks** sont des méthodes qui permettent de s'abonner à une de ces étapes.

```
<script setup lang="ts">
import { onMounted, onUnmounted } from "vue";

onMounted(() => {
  console.log("Composant inséré dans le DOM");
});

onUnmounted(() => {
  console.log("Composant retiré du DOM");
});
</script>
```

Les Props

Un composant peut être initialisé avec des paramètres. Ces paramètres sont appelés des props.

Les props sont des propriétés similaires aux attributs des balises HTML.

Les props vont servir à configurer votre composant lors de son utilisation.

```
<my-button shape="rounded"></my-button>
```

Les Props

Les props doivent être déclarées avec la fonction **defineProps()** pour pouvoir être utilisées.

```
<script setup lang="ts">
import { computed } from 'vue';

const props = defineProps({
    shape: {
        type: String,
        default: 'rounded',
        validator: (value: string) => ['rounded', 'square', "circle"].includes(value),
    },
});

const buttonClasses = computed(() => [
    `button-shape-${props.shape}`,
]);
</script>
```

Le Binding

La valeur passer à votre prop via un attribut sera **toujours une string.**

L'usage du **:** ou du mot clé **v-bind** permet de mettre une expression JavaScript ou une variable.

Lorsque cette variable est **réactive**, le composant pourra se mettre à jour en fonction des changements fait sur la variable.

Le binding peut s'utiliser également sur les attributs natifs d'un élément : id, src, href, etc.

Déclarer des évènements

Les composants ont la capacité d'émettre le propres évènements, en plus des évènements natifs. (click, mouseover, etc.)

La fonction **defineEmits**() permet de déclarer les évènements qu'on composant est capable d'émèttre.

Elle retourne une fonction permettant d'émettre les évènements déclarés.

Emettre un évènement

La fonction defineEmits prend en paramètre le nom des évènements à déclarés. Les évènements sont émis via la fonction retournée.

```
<script setup lang="ts">
import { defineEmits, ref } from "vue";

const count = ref(0);
const emit = defineEmits(["change"]);

setInterval(() => {
  count.value = count.value + 1;
  emit("change", count.value);
}, 1000);
</script>
```

Réagir aux changements

Lorsque, par exemple, une props est déclarée, il est possible que la valeur qui lui soit passée change.

La fonction **watch()** permet de s'abonner aux changements d'une variable réactive, qu'elle soit une **props** ou une variable déclarée via **ref()**

```
const count = ref(0);
watch(count, (count, prevCount) => {
  /**
   * count étant la nouvelle valeur,
   * prevCount l'ancienne
   */
});
```

Réagir aux changements

Dans le cas de l'objet **props**, le premier argument doit être une fonction retournant la propriété à surveiller :

```
const props = defineProps({
    shape: {
        type: String,
        default: 'rounded',
        validator: (value: string) => ['rounded', 'square', "circle"].includes(value),
    },
});

watch(() => props.shape, (count, prevCount) => {

  }
);
```

IV

Component Template

L'usage des slots

Le slot est une zone dans laquelle vous allez pouvoir injecter le contenu saisi entre la balise ouvrante et la balise fermante de votre composant.

```
<template>
  <my-button>
    My Ugly Button
  </my-button>
</template>
```

Contenu à restituer

App.vue

L'usage des slots

Le composant spécial **<slot></slot>** permet d'indiquer à Vue où restituer le contenu au sein de votre composant.

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

Endroit où injecter le contenu

Button.vue

Conditional Rendering

Il est possible de n'afficher certain composant que sous certaines conditions.
En vue il existe 2 directives pour gérer ce cas : **v-if** et **v-show**.

```vue
<script setup lang="ts">
import { ref } from "vue";

const isOpen = ref(false);
</script>
<template>
    <div v-if="isOpen">...</div>
</template>
```

```vue
<script setup lang="ts">
import { ref } from "vue";

const isOpen = ref(false);
</script>
<template>
    <div v-show="isOpen">...</div>
</template>
```

Conditional Rendering

La directive **v-if** va retirer ou insérer l'élément dans le DOM en fonction d'une expression JavaScript.

Si l'expression est évaluée à **true**, l'élément est inséré. A **false**, l'élément est retiré.

L'élément ne sera jamais inséré tant que l'expression ne passe pas à **true.**

Conditional Rendering

La directive **v-show** en revanche ne fait que jouer sur la propriété CSS display.

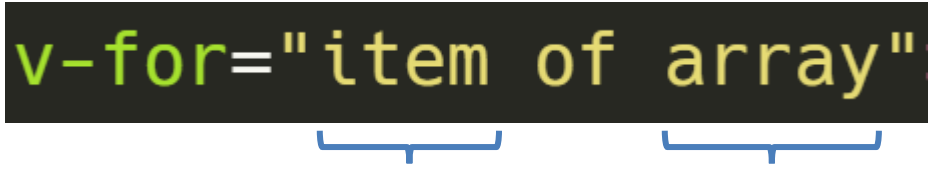
Si l'expression est évaluée à **true**, la propriété display inchangée. A **false**, **display** passera à *none*.

L'élément sera toujours présent dans le DOM, mais inséré qu'une seule fois à l'initialisation.

Itération

La directive **v-for** permet d'itérer sur un tableau et de dupliquer l'élément sur laquelle elle est appliquée autant de fois qu'il y a d'éléments dans le tableau.

L'expression comporte à gauche le nom de la variable qui reçoit les éléments du tableau et à droite le tableau sur lequel itérer.



Exemple avec un tableau d’objets

```
<script setup lang="ts">
const menu = [
    { title: "Home", url: "/home" },
    { title: "Profile", url: "/profile" },
    { title: "Settings", url: "/settings" },
];
</script>
<template>
    <ul>
        <li v-for="entry of menu">
            <a :href="entry.url">{{ entry.title }}</a>
        </li>
    </ul>
</template>
```

Class Binding

Il est possible d'utiliser le **binding** sur l'attribut class. La valeur passée peut être :

- Un objet contenant la liste des classes à appliquer/retirer
- Un tableau listant les classes à appliquer
- Une string contenant la ou les classes séparées par des espaces

Si le binding se fait avec une variable, il faut veiller à ce que cette dernière soit réactive pour que tout changements sur la variable impacte bien l'attribut class de l'élément.

Class Binding

```
<script setup lang="ts">
import { computed } from 'vue';

const props = defineProps({
    shape: {
        type: String,
        default: 'rounded',
    },
});

const buttonClasses = computed(() => [
    `button-shape-${props.shape}`,
]);
</script>

<template>
    <button class="iti-button" :class="buttonClasses">
        ...
    </button>
</template>
```

Binding avec un tableau de string

Class Binding

Binding avec un objet

```
<script setup lang="ts">
import { ref } from "vue";

const isOpen = ref(false);
</script>
<template>
    <div class="message-box" :class="{'is-open': isOpen }">
        ...
    </div>
</template>
```

Style Binding

Pareillement, le binding peut s'utiliser avec l'attribut **style.**

Le style se configure avec un objet dont les clés sont les propriétés CSS à modifier :

Les propriétés CSS peuvent être en camelCase ou en kebab-case

Style Binding

Example d’utilisation du style binding

```
<script setup lang="ts">
import { ref } from "vue";

const isOpen = ref(false);
</script>
<template>
    <div :style="{'display': isOpen ? : 'block' : 'none' }">
        ...
    </div>
</template>
```

Template Ref

Une template ref est une variable déclarée dans le template.

La template Ref est initialisée avec l'élément sur lequel elle s'applique.

La Template Ref est accessible dans le template via l'objet spécial **$refs**

```
<template>
    <input type="text" ref="myInput" />
    <div v-show="!$refs.myInput.validity.valid" >
        Invalid Input
    </div>
</template>
```

Template Ref & Composant

Pour accéder au contenu d'une Template Ref en dans script, il suffit d'initialiser une variable du même nom avec la fonction **ref(**null**).**

```
<template>
    <input type="text" ref="myInput" />
    <div v-show="!$refs.myInput.validity.valid" >
        Invalid Input
    </div>
</template>
<script setup lang="ts">
  import {onMounted} from "vue";
  const myInput = ref(null);

  onMounted(()=> {
    console.log(myInput.validity)
  });
</script>
```

Attention : La variable sera à null tant que le composant n'est pas mounted.

Template Ref & Composant

Lorsque qu'une **Template Ref** est appliquée sur un composant, elle permet de donner accès à son **instance.**

L'instance peut donner accès aux variables et fonctions publiques d'un composant

Toutefois, les variables et fonctions étant privées par défaut, il faut utiliser la fonction **defineExpose()** pour marquer variables ou fonctions comme étant accessible publiquement depuis une **Template Ref.**

V

Les Formulaires

Les formulaires

Les formulaires jouent une importance cruciale dans une Web App :

Ils permettent aux utilisateurs de saisir et de soumettre des informations.

Lorsque l'on gère des données saisies par l'utilisateur, il est important de :

- Pouvoir récupérer la donnée saisie depuis un input
- Valider la donnée saisie
- Communiquer sur les erreurs de validation

Les Inputs

Le binding fonctionne avec tous les attributs HTML, y compris l'attribut **value** de la balise input :

```
<script setup lang="ts">
import { ref } from "vue";

const inputValue = ref("mon texte");

function onChange(e) {
  inputValue.value = e.srcElement.value;
}
</script>
<template>
  <input type="text" :value="inputValue" @input="onChange" />
</template>
```

La directive v-model

Pour simplifier la gestion des formulaires, Vue introduit une directive spéciale : la directive **v-model.**

v-model permet de faire du ***two-way binding*** :

- lorsque la valeur change dans le Javascript, l'input est mis à jour.
- lorsque la valeur de l'input change, la variable est mise à jour dans le Javascript

```
<input type="text" v-model="inputValue" />
```

La directive v-model

Il n’est donc plus nécessaire de s’abonner à l’évènement input :

```
<script setup lang="ts">
import { ref } from "vue";

const inputValue = ref("mon texte");
</script>
<template>
  <input type="text" v-model="inputValue" />
</template>
```

La directive v-model

De plus, v-model fonctionne avec toutes les balises de saisie :

Tous les types d’inputs, textarea et select.

```
<script setup lang="ts">
import { ref } from "vue";
const inputValue = ref(1);
</script>
<template>
    <select type="text" v-model="inputValue">
    <option value="1">One</option>
    <option value="2">Two</option>
    <option value="3">Three</option>
  </select>
</template>
```

v-model et sélection multiple

La directive v-model fonctionne aussi avec plusieurs valeurs pour les select et les inputs de type **checkbox**.

Validation des formulaires

VueJS ne gère pas la validation des formulaires nativement. Il faut donc se tourner vers des librairies spécifiques :

- Vee Validate
- Vuelidate

VI

Le Routing

La gestion des pages

Vue permet par défaut de faire des SPA.

Les routes sont donc gérées par le Framework.

Le principe est d'associer une portion de l'URL à un composant.

Les « pages » de l'application seront donc des composants liés à une route.

Le router de Vue est disponible dans package séparé **vue-router**

Configuration

Pour configurer le routing il faut donc :

- Un composant par route
- Un objet pour de configuration pour faire la correspondance entre les composants et les routes
- Indiquer à Vue à quel endroit de votre application insérer les pages dynamiquement

Configuration

La configuration des routes se fait via un tableau.

Le routeur étant un plugin Vue, il faut la rajouter à l'application avec la méthode **use**() de votre instance Vue

```
const router = VueRouter.createRouter({
  history: VueRouter.createWebHashHistory(),
  routes: [...]
});

const app = Vue.createApp({})
app.use(router);
app.mount('#app');
```

Configuration

Une route se configure en associant un path de l'URL à un composant :

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

router-view

Pour indiquer à Vue où insérer les pages dans votre application, il faut utiliser le composant **<router-view></router-view>**

```
<div id="app">
  <h1>Hello App!</h1>

  <!-- component matched by the route will render here -->
  <router-view></router-view>
</div>
```

Navigation avec router-link

Pour la navigation, il faut abandonner la balise **<a>…** au profit du composant **router-link**

```
<div id="app">
  <h1>Hello App!</h1>
  <p>
    <router-link to="/">Go to Home</router-link>
    <router-link to="/about">Go to About</router-link>
  </p>

  <!-- component matched by the route will render here -->
  <router-view></router-view>
</div>
```

Router API

Au sein d'un composant, vous pouvez également utiliser l'instance du router pour changer de pages.

La fonction **useRouter()** retourne l'instance du routeur. La méthode **router.push(**url**)** permet de naviguer vers une autre URL.

```
import { useRouter } from "vue-router";
const router = useRouter();

router.push("/home");
```

Paramètres

Les routes peuvent être configurés avec des paramètres.

La valeur des paramètres est accessible sur l'objet route accessible avec **useRoute()** à ne pas confondre avec **useRouter()**

```
import { useRoute } from "vue-router";
const route = useRouter( );

const articleId = route.params.articleId;
```