Présentation de Vue.js et Nuxt.js

RÉALISATION D'UN MINI-BLOG

Réalisé pour : Epitech - Module Innovation

Réalisé par : Julien Léos, Étudiant 2ème année

14 Mai 2019

SOMMAIRE

INTRODUCTION

Objectif

Réaliser un mini-blog comprenant:

- Une page principale regroupant tous les « posts » du blog
- Une page pour créer un nouveau « post »
- Une page de visualisation d'un « post »

Languages

Nous serons amenés à utiliser les langages suivants:

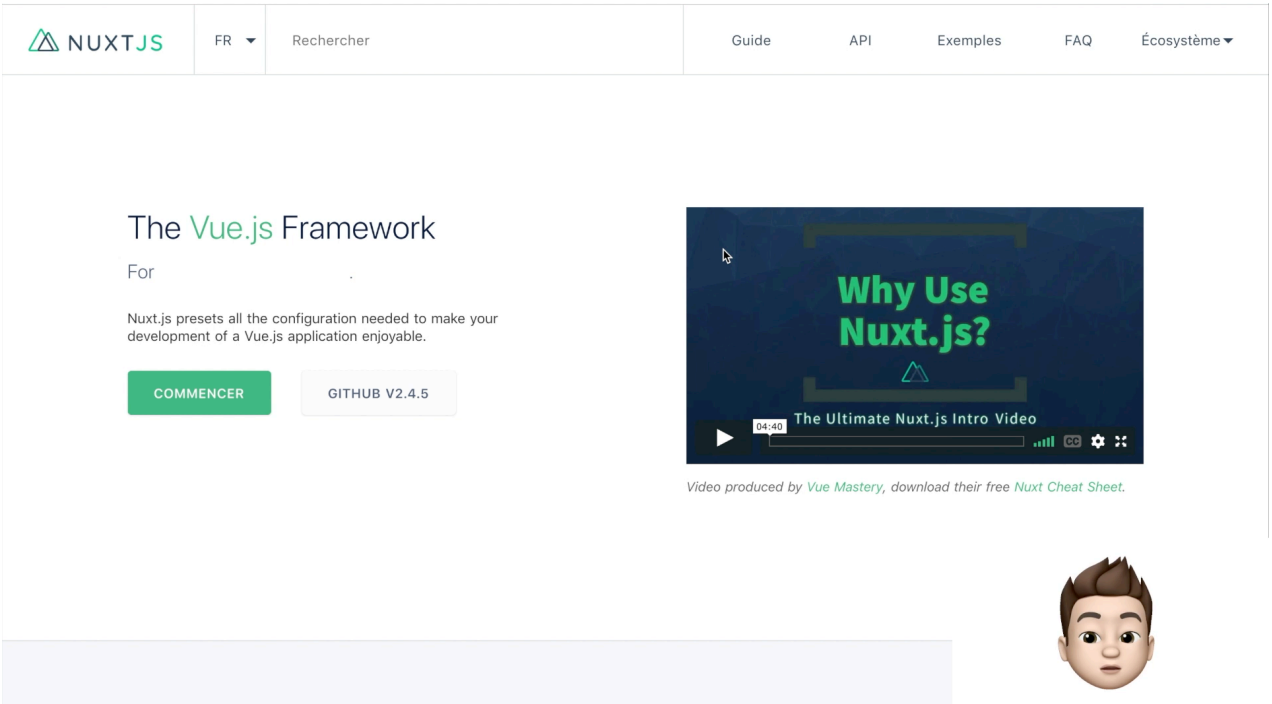
- HTML
- CSS
- JS

Ainsi que les frameworks suivants:

- Vue.js
- Nuxt.js

Vidéo

Toutes les instructions nécessaires à l'installation et à la mise en place d'un projet Nuxt.js sont expliquées dans cette vidéo. Vous pouvez également les retrouver ci-dessous. ***Lien YT***



Installation

Tout d'abord, il va vous falloir installer Vue.js grâce à la commande suivante

```
$ npm install vue
```

Ensuite, installez Vue CLI. Il va vous permettre d'installer un projet pré-fait pour notre blog.

```
$ npm install -g @vue/cli @vue/cli-init
```

Vous pouvez ensuite initier votre projet avec la commande suivante. Remplacez « project-name » par le nom de votre projet (e.g « mini-blog »).

```
$ vue init nuxt-community/starter-template <project-name>
```

Déplacez vous dans le dossier crée et installez toutes les dépendances nécessaires.

```
$ npm install
```

Vous pouvez ensuite lancer le projet Nuxt.js grâce à la commande suivante.

```
$ npm run dev
```

Enfin, rendez-vous sur votre localhost: http://localhost:3000. Vous devriez observer ce résultat.

INITIATION AU VUE.JS

Extension .vue

Vue.js est un framework de JS. Celui-ci ajoute une nouvelle extensions de fichier: .vue.

Un fichier .vue se découpe en 3 parties:

1. Le **template**: Cette partie va contenir l'ensemble du code HTML de votre fichier.
2. Le **script**: Cette partie va contenir l'ensemble du code JS de votre fichier.
3. Le **style**: Cette partie va contenir l'ensemble du code CSS de votre fichier.

Plutôt logique n'est-ce pas ?

Exemple

Voici un exemple d'une simple page affichant « Hello world »

```
<template>
  <div class="title">
    <h1>{{message}}</h1>
  </div>
</template>

<script>
  export default {
    data() {
      return {
        message: "Hello World!"
      }
    }
  }
</script>

<style>
  .title {
    font-family: "Quicksand";
    display: block;
    font-weight: 300;
    font-size: 100px;
    color: #35495e;
    letter-spacing: 1px;
  }
</style>
```

Tout d'abord, observez la partie **script**. Celle-ci comporte un champ « **data** », vous apprendrez par la suite qu'il existe de nombreux champs possible. Ce champs « data » renvoie un object contenant une clé « **message** » liée à la valeur « Hello World! ». C'est ainsi que l'on déclare des variables en Vue.js.

Regardez ensuite la partie **template**. Vous pouvez retrouvez notre variable « message » entre double accolades signifiant qu'il s'agit non pas de texte, mais bien d'une variable Vue.js.

La partie **style** se comporte comme n'importe quel fichier CSS.

Résultat

LES ATTRIBUTS VUE.JS

Comme dans tout bon framework Web qui se respecte, ils viennent souvent avec leur lot de balises et d'attributs HTML. Cette règle n'échappe pas au Vue.js qui redéfini bon nombre de logiques liées à la programmation à l'aide d'attributs spéciaux. En Vue.js, les attributs spéciaux sont reconnaissables à l'aide de leur mot clé « **v-** ». Ils peuvent également être abrégés avec un simple « **:** ».

La condition IF

Voyons un exemple simple: la condition IF.

```
<template>
  <div>
    <h1 v-if="show">{{message}}</h1>
  </div>
</template>

<script>
  export default {
    data() {
      return {
        message: "Hello World!",
        show: true
      }
    }
  }
</script>
```

L'attribut « **v-if** » va définir, en fonction de la valeur qui lui est passé en paramètre, si oui ou non, il va afficher la balise à laquelle il (l'attribut) est affecté. Ainsi, si la variable « **show** » est passé à « false », la balise « h1 » ne sera plus affichée.

La boucle FOR

```
<template>
  <div>
    <h1 v-for="university in data" v-bind:key="university">
      {{university}}
    </h1>
  </div>
</template>

<script>
  export default {
    data() {
      return {
        data: ["Berkley", "Concordia", "Long Beach", "Kent", "Boston"]
      }
    }
  }
</script>
```

Ici comme vous l'avez surement deviné, nous allons voir comment créer une boucle FOR. Il faut pour cela utiliser le mot clé « **v-for** » qui va contenir la variable contenant elle-même chaque itération de la boucle, suivi par le mot clé « in » et enfin, la variable contenant votre donnée sur laquelle boucler.

Vous remarquerez également la présence d'un attribue « **v-bind:key** » pouvant se raccourcir en « :key ». Celui-ci doit contenir un identifiant **UNIQUE** pour chaque itération de notre boucle. Ici, le contenu de chaque itération suffit. Mais dans le cas où l'une des université aurait été en double, il aurait fallu utiliser des « **id** ».

Berkley
Concordia
Long Beach
Kent
Boston

Alors, quelle université devrais-je choisir pour ma 4ème année 🤔 ?

LES COMPOSANTS VUE.JS

Création d'un composant

Un composant Vue.js est un « morceau de code » scalable et facilement réutilisable. Il se matérialise sous la forme d'une fichier .vue comportant les mêmes 3 parties citées précédemment. À l'exception que celui-ci va pouvoir recevoir des « **props** » correspondant à des attributs spéciaux propres à notre composant.

Commençons par définir un simple composant définissant notre nom, notre âge et nos centres d'intérêts.

```
<template>
  <div>
    <h1>My name is {{name}}, I'm {{age}} years old.</h1>
    <h2>My hobbies are:</h2>
    <p v-for="hobbie in hobbies" :key="hobbie">
      {{hobbie}}
    </p>
  </div>
</template>

<script>
  export default {
    name: 'description',
    props: {
      name: String,
      age: Number,
      hobbies: Array
    }
  }
</script>
```

Normalement, rien de nouveau dans la partie « template ». Cela se corse un peu au niveau de la partie « **script** », jetons un oeil aux nouveautés.

Tout d'abord, nous avons défini un champ « name » qui va comporter comme son nom l'indique: le nom du composant. En second lieu, nous retrouvons le champs « props » dont je vous parlais juste au dessus. Celui-ci va définir l'ensemble des attributs possibles de

notre composant ainsi que leur **TYPE**. Même s'ils ne sont pas obligatoires, ils est fortement conseillé de les préciser, cela évitera de pouvoir compiler un code qui risque de bugger si vous avez envoyé un booléen à la place d'une liste par exemple.

Implémentation d'un composant

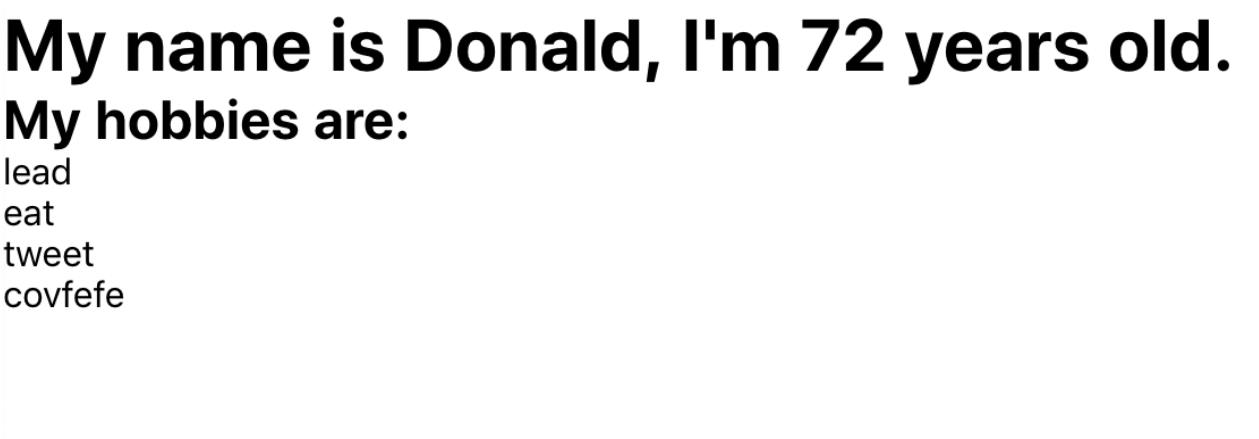
Maintenant que nous avons défini notre composant, il est temps de l'implémenter à notre fichier principal.

```
<template>
  <div>
    <description name="Donald" :age="72" :hobbies="hobbies"/>
 </div>
</template>

<script>
  import Description from "~/components/description.vue"

  export default {
    components: {
      Description
    },
    data() {
      return {
        hobbies: ["lead", "eat", "tweet", "covfefe"]
      }
    }
  }
</script>
```

Tout d'abord, il vous faudra importer votre composant dans votre partie « **script** ». Pour ce faire, saisissez le mot-clé « **import** » suivi du nom de l'object à importer (la norme veut que les noms d'« import » soit en « CamelCase ») puis à nouveau du mot-clé « **from** » avant de finir par le chemin vers le fichier à importer. (Si vous avez regardé la vidéo du début, vous saurez comprendre pourquoi notre fichier « description.vue » se trouve dans un dossier « components »… même si cela peut sembler assez logique tout compte fait).

Il vous faudra par la suite ajouter un champs « **components** » regroupant l'ensemble des composants importés.

Enfin, il vous suffira de créer une balise **HTML** du nom du composant et d'y passer l'ensemble (ou pas) des attributs définis dans celui-ci. N'oubliez pas le mot-clé « **v-bind** » ou son abréviation « **:** » devant chaque attribut nécessitant l'appel d'une variable Vue.js.
PS: Dans le cas de l'âge, étant donné que nous avons spécifié qu'il s'agissait d'un nombre et non d'une string, il est possible de directement passer un nombre sans avoir à déclarer une variable.



MINI-BLOG

Bien ! Trêve de théorie, passons à la pratique. Voilà ce que nous allons réaliser au cours de ce Workshop. ***Lien YT***

Pour rappel, ce mini-blog comportera:

- Une page principale regroupant tous les « posts » du blog
- Une page pour créer un nouveau « post »
- Une page de visualisation d'un « post »

La page principale

Comme vous pouvez le constater sur la vidéo, lorsque l'on ajoute un nouveau « post », celui-ci s'affiche sur la page principale indiquant son titre, sa description et son auteur.
Votre premier objectif est de créer le composant qui affichera ces informations.

Si ce n'est pas déjà fait, je vous invite vivement à visionner la première vidéo afin de comprendre comment est organisé votre projet Nuxt.js et où créer ce composant.
Je vous fourni également le CSS nécessaire pour le composant ainsi que le « layout » du projet (si le mot « layout » ne vous dis rien, c'est que vous n'êtes toujours pas allé voir cette vidéo 😠). ***Lien du CSS*** - ***Lien du layout***.

Une fois ceci fait, voici ce que vous devriez obtenir.

L'étape suivante consiste à **créer un bouton qui va générer un nouveau composant « post » à chaque fois qu'il sera cliqué**. Pour cela, il va nous falloir apprendre comment sont gérés les événements en Vue.js.

Tout comme « v-bind » permet d'indiquer à un attribut qu'il est associé à Vue.js, « **v-on** » et son raccourci « **@** » permettent de définir des événements liés à Vue.js. Voyons un exemple.

```
<template>
  <div>
    <h1 v-on:click="increase">{{counter}}</h1>
  </div>
</template>

<script>
  export default {
    data() {
      return {
        counter: 0,
      }
    },
    methods: {
      increase() {
        this.counter += 1;
      }
    }
  }
</script>
```

Si vous arrivez à suivre jusque la, vous n'aurez pas de mal à comprendre que lorsque l'on effectue un clic de souris sur la balise « h1 », « increase » est appelé. Mais alors, que représente « increase » ?
« increase » est une « **méthode** », c'est l'équivalent des fonctions en Vue.js. Celles-ci sont toutes regroupées dans un champ « **methods** ». Quand aux variables définies dans le champ « data », elles sont accessibles grâce à l'object « **this** ».
Ce code permet donc d'incrémenter le chiffre affiché lorsque l'on clic dessus.

Une fois le bouton ajouté, vous devriez avoir quelque chose qui ressemble à ça. ***Lien YT***

Bien sur, j'ai volontairement omis quelques étapes comme le fait de trouver un moyen de stocker l'ensemble des données (un tableau d'object devrait faire l'affaire) ou encore un moyen d'afficher tous ces « posts » (un indice… ça rime avec avec « chateau-fort »).

La page de visualisation

Nous pouvons ajouter de nouveaux posts et prévisualiser quelques informations à leur sujet… super. Maintenant, il va falloir songer à afficher la véritable page propre à tous nos « posts ». Bien sur, vous n'imaginez quand même pas que nous allons créer une à une les pages quasiment identiques (il n'y a que le contenu qui change) pour chaque « post » ?

Pour ça, il existe ce que l'on appelle les « **routes dynamiques** » de Nuxt.js. Afin d'économiser du papier et parce que j'ai des projets à avancer, je vous laisse vous documenter vous même sur comment les mettre en place grâce à ***ce lien***.

Revenons sur notre page principales pour connecter nos « posts » à leur page dédiée. Pour ce faire, Nuxt.js ajoute une balise très pratique: « **nuxt-link** ». Mais bien sur, vous le saviez déjà puisque vous êtes allés voir ma vidéo 😉.

Une implémentation simple (comme dans la vidéo) de « nuxt-link » ne va pas suffire étant donné qu'il s'agit d'une route dynamique et que nous devons par conséquent lui passer un « **id** » en paramètre. Voici un exemple d'implémentation d'une route dynamique à l'aide de « nuxt-link »:

```
<template>
  <div>
    <div v-for="num in 10" :key="num">
      <nuxt-link :to="{name: 'id', params: {id: num}}">
        Page number {{num}}
      </nuxt-link>
    </div>
  </div>
</template>
```

Ce code va simplement boucler 10 fois et créer 10 liens vers 10 pages dynamiques différentes. Vous pouvez constater que l'attribut « **to** » ne contient plus une string (comme dans la vidéo) mais un object.
Celui-ci possède une clé « **name** » qui correspond au **NOM** de la route visée (et non pas son chemin !). Le nom d'une route peut être trouvé dans le fichier « **router.js** » du dossier « **.nuxt** » de votre projet Nuxt.js.
La clé « **params** » correspond quand à elle aux paramètres que l'on souhaite passer à notre route dynamique, ici cela va de soit, nous lui passons son Id. Voici le résultat obtenu:

Page number 1
Page number 2
Page number 3
Page number 4
Page number 5
Page number 6
Page number 7
Page number 8
Page number 9
Page number 10

Voici venu le temps des cathé… trop tôt pardon… Voici venu le temps de remplir notre page de visualisation d'un « post ». Si vous avez déjà commencé à vouloir la remplir vous vous êtes surement heurtés à un problème… comment accéder aux données de mon « post » ? Une solution consisterait à passer l'intégralité de nos données en paramètre… vous voyez pourquoi ca ne va pas ? Non ? Bien… imaginons que votre « post »

comptabilise une soixantaine de lignes, vous imaginez la taille de l'URL si votre article est présent en intégralité dans celle-ci… ?

Pour y remédier, nous allons utiliser un system très pratique mis en place par Nuxt.js: **Vuex**. Celui-ci consiste à créer un « **store** » dans lequel nous allons stocker nos données. Ce « store » contiendra des « **setters** » et des « **getters** » nous permettant de respectivement ajouter (ou supprimer) de nouvelles données à notre « store » et de récupérer les données de notre « store ».

```
import Vuex from 'vuex'

const createStore = () => {
  return new Vuex.Store({
    state: {
      counter: 0
    },
    mutations: {
      increase(state, i) {
        state.counter += i;
      },
      decrease(state, i) {
        state.counter -= i;
      }
    },
    getters: {
      getCounter(state) {
        return state.counter
      }
    }
  })
}

export default createStore
```

Jetez un oeil à ce code, il représente un « store » assez simple contenant une variable « counter » que l'on va pouvoir incrémenter et décrémenter grâce à deux « **mutations** » (les « setters »). On va également pouvoir récupérer la valeur de « counter » grâce au « **getter** » « getCounter ».

```
<template>
  <div>
    <h1 @click="increase">{{counter}}</h1>
  </div>
</template>

<script>
  export default {
    methods: {
      increase() {
        this.$store.commit('increase', 1);
      }
    },
    computed: {
      counter() {
        return this.$store.getters.getCounter;
      }
    }
  }
</script>
```

Et voilà une implémentation basique du « store » précédemment crée. Nous pouvons accéder à notre « store » simplement depuis « **this.$store** » suivis de « **.getters** » pour les « getters » et de « **.commit** » pour les « setters ».
Vous avez également du remarquer l'apparition d'un nouveau champs « **computed** ». Celui-ci est très puissant puisqu'il permet de rendre les fonctions dynamiques. C'est à dire qu'elles vont se mettre à jour automatiquement si elles détectent qu'une des variables / fonctions / getters auxquelles elles font appel à changé. Ainsi, dans notre exemple, « counter » va se mettre à jour automatiquement et rappeler « getCounter » dès que la variable « counter » de notre « store » changera. Puissant n'est-ce pas ?

Maintenant, à vous de créer votre propre « store » qui devra, je vous le rappelle, contenir toutes les données des « posts » crées. Grâce à la super vidéo du début, vous savez où créer le fichier « **index.js** » qui va contenir notre store 😉. Il ne vous faudra pas oublier de **mettre à jour le bouton d'ajout d'un post** (« setter ») ainsi que la **boucle permettant d'afficher l'ensemble des « posts »** (getter).

Bon ! C'est bien beau tout ce bla-bla mais en attendant, nous n'avons toujours pas la page de visualisation d'un « post ». De plus, je ne vous ai pas dis comment l'on accède à l'Id d'une page dynamique, car sans l'Id, difficile de savoir quel « post » il va falloir récupérer dans notre « store » n'est-ce pas ? Et bien pour ça, je vous laisse chercher tout seul ! (Bon ok je suis gentil, voilà deux liens qui devraient vous être utiles: ***Ici*** et ***là***). Quand à la page de visualisation, je vous fais à nouveau cadeau ***du CSS***.

Après avoir mis tout ça en place, voici ce à quoi notre blog devrait ressembler. ***Lien YT***

Bien, notre blog commence à ressembler à quelque chose ! Et même si pour l'instant le contenu de nos « posts » sont les mêmes, vous pouvez remarquer que l'URL elle, change bien entre chaque « post ».

La page de création

Allez courage, on touche au but ! Pour la page d'ajout d'un nouveau post, rien de bien compliqué, voici la marche à suivre:

- Créer une page « standard » qui va nous permettre de renseigner:
  - Un titre
  - Une description
  - Un auteur
  - Un contenu
- Créer un bouton d'ajout connecté au « setter » d'ajout de notre « store ».
- Rediriger le bouton d'ajout de la page principal vers cette nouvelle page.

Et c'est tout ! Si vous avez tout bien fait depuis le début, cette dernière étape devrait être très facile à mettre en place. Bien sur, voici ***le CSS*** ainsi qu'***un lien*** qui vous sera très utile pour créer le « form » nécessaire au renseignement des différentes données d'un « post ».

Tadam ! Votre mini-blog est normalement terminé, j'espère que vous en êtes fier ! Moi en tout cas je suis fier de vous 😁 !

Bon, mais comme vous n'en avez jamais assez, voici une liste de bonus intéressants:

- Un bouton « delete » pour supprimer un « post ».
- Renseigner la date de création d'un « post ».
- Sauvegarder les données du « store » dans le « local storage » de votre navigateur.
- Faire un système de compte sans mot de passe où l'utilisateur peut renseigner son pseudo. (l'auteur n'est donc plus à préciser lors de la création d'un nouveau « post »).
- Faire un système de compte **AVEC** mot de passe où l'utilisateur doit se connecter ou créer un compte avant de pouvoir écrire un « post ».