

# Vue.js : formulaire et validation avec vee-validate

**Achref El Mouelhi**

Docteur de l'université d'Aix-Marseille  
Chercheur en programmation par contrainte (IA)  
Ingénieur en génie logiciel

`elmouelhi.achref@gmail.com`



# Plan

- 1 Introduction
- 2 Liaison (binding) bidirectionnelle
- 3 Soumission de formulaires
- 4 Validation de formulaires
- 5 `vee-validate`
  - Installation
  - `Form` et `Field`
  - Récupération de valeurs
  - Gestion de formulaires
  - Validation de formulaires
  - Messages d'erreur
  - Validateurs globaux
  - Schéma de validation
  - Valeurs initiales
  - Soumission invalide

- 6 yup
  - Définition de validateurs
  - Modification de `Locale`
  - Messages d'erreur personnalisés
  - Schéma de validation
- 7 Gestion de formulaires avec `ref` et `v-slot`
  - Utilisation de `ref`
  - Utilisation de `v-slot`
- 8 Configuration de la validation
- 9 Formulaires dynamiques
- 10 Formulaires imbriqués
  - Objet imbriqué
  - Tableau imbriqué

## Formulaire

- Outil graphique que nous créons avec le langage **HTML**
- Permettant à l'utilisateur d'interagir avec l'application en
  - saisissant de données
  - cochant des cases
  - sélectionnant des options
- Solution pour soumettre les données vers
  - une autre page/composant,
  - une ressource externe (base de données...)

## Apport de **Vue.js** ?

- Récupération de données saisies (**Form binding**)
- Validation et contrôle de valeurs saisies
- Gestion d'erreurs
- ...

## Dans la suite

Nous allons

- 1 créer le composant `PersonneAdd` dans `components`,
- 2 définir `PersonneAdd` comme composant enfant de `PersonneShowView`,
- 3 préparer `PersonneAdd` pour ajouter des nouvelles personnes (dans la base de données).

Commençons par définir `PersonneAdd` comme enfant de `PersonneShowView`

```

<template>
  <h1>Gestion de personnes</h1>
  <PersonneAdd />
  <ul>
    <li v-for="(elt, index) in personnes" :key="index">
      <router-link :to="{ name: 'personne-details', params: { id: elt.id } }">
        {{ elt.nom }} {{ elt.prenom }}
      </router-link>
    </li>
  </ul>
</template>

<script>
import PersonneAdd from '@components/PersonneAdd.vue';
export default {
  name: 'PersonneShowView',
  components: {
    PersonneAdd
  },
  data() {
    return {
      personnes: [
        { id: 1, nom: 'Wick', prenom: 'John', age: 45 },
        { id: 2, nom: 'Dalton', prenom: 'Jack', age: 40 },
        { id: 3, nom: 'Dupont', prenom: 'Sophie', age: 30 }
      ]
    }
  }
}
</script>

```

# Vue.js

## Le template de `PersonneAdd`

```
<template>
  <h1>Ajouter une nouvelle personne</h1>
  <form >
    <div>
      Nom : <input type=text>
    </div>
    <div>
      Prénom : <input type=text>
    </div>
    <div>
      Age : <input type=number>
    </div>
    <div>
      <button type="button" @click="ajouterPersonne()">
        Ajouter
      </button>
    </div>
  </form>
</template>
```



# Vue.js

Pour assurer le binding, on utilise `v-model`

```
<template>
  <h1>Ajouter une nouvelle personne</h1>
  <form >
    <div>
      Nom : <input type=text v-model="nom">
    </div>
    <div>
      Prénom : <input type=text v-model="prenom">
    </div>
    <div>
      Age : <input type=number v-model="age">
    </div>
    <div>
      <button type="button" @click="ajouterPersonne()" >
        Ajouter
      </button>
    </div>
  </form>
</template>
```

# Vue.js

Pour éviter de récupérer les valeurs saisies dans des variables séparées, nous pourrions récupérer les valeurs dans un objet

```
<template>
  <h1>Ajouter une nouvelle personne</h1>
  <form >
    <div>
      Nom : <input type=text v-model="personne.nom">
    </div>
    <div>
      Prénom : <input type=text v-model="personne.prenom">
    </div>
    <div>
      Age : <input type=number v-model="personne.age">
    </div>
    <div>
      <button type="button" @click="ajouterPersonne()" >
        Ajouter
      </button>
    </div>
  </form>
</template>
```

# Vue.js

Dans la partie `script`, déclarons l'objet `personne` dans `data` et préparons la méthode `ajouterPersonne()`

```
<script>
export default {
  name: 'PersonneAdd',
  data() {
    return {
      personne: { prenom: '', nom: '', age: null }
    }
  },
  methods: {
    ajouterPersonne() {
      console.log(this.personne)
    }
  },
}
</script>
```

# Vue.js

Dans la partie `script`, déclarons l'objet `personne` dans `data` et préparons la méthode `ajouterPersonne()`

```
<script>
export default {
  name: 'PersonneAdd',
  data() {
    return {
      personne: { prenom: '', nom: '', age: null }
    }
  },
  methods: {
    ajouterPersonne() {
      console.log(this.personne)
    }
  },
}
</script>
```

Remplissez les champs du formulaire, cliquez sur le bouton et vérifiez que les valeurs saisies s'affichent dans la console du navigateur.

## Question 1

Et si on voulait aussi soumettre le formulaire en cliquant sur la touche Entrée ?

© Achref EL MOUADJID

## Question 1

Et si on voulait aussi soumettre le formulaire en cliquant sur la touche Entrée ?

## Réponse

On utilise un bouton de soumission

## Question 2

Et si on voulait avoir plusieurs boutons de soumission dans un même formulaire qui renvoient vers la même méthode ?

© Achref EL ME

## Question 2

Et si on voulait avoir plusieurs boutons de soumission dans un même formulaire qui renvoient vers la même méthode ?

## Réponse

On remonte l'évènement à la balise `<form>`.



# Vue.js

Utilisons `@submit` pour la soumission du formulaire

```
<template>
  <h1>Ajouter une nouvelle personne</h1>
  <form @submit="ajouterPersonne">
    <div>
      Nom : <input type="text" v-model="personne.nom">
    </div>
    <div>
      Prénom : <input type="text" v-model="personne.prenom">
    </div>
    <div>
      Age : <input type="number" v-model="personne.age">
    </div>
    <div>
      <button>
        Ajouter
      </button>
    </div>
  </form>
</template>
```

# Vue.js

Utilisons `@submit` pour la soumission du formulaire

```
<template>
  <h1>Ajouter une nouvelle personne</h1>
  <form @submit="ajouterPersonne">
    <div>
      Nom : <input type="text" v-model="personne.nom">
    </div>
    <div>
      Prénom : <input type="text" v-model="personne.prenom">
    </div>
    <div>
      Age : <input type="number" v-model="personne.age">
    </div>
    <div>
      <button>
        Ajouter
      </button>
    </div>
  </form>
</template>
```

## Remarque

En cliquant sur le bouton, la page est rechargée de nouveau (Ce qui est contradictoire avec **SPA**).

# Vue.js

Pour éviter de recharger la page, on utilise le modificateur `prevent`

```
<template>
  <h1>Ajouter une nouvelle personne</h1>
  <form @submit.prevent="ajouterPersonne">
    <div>
      Nom : <input type="text" v-model="personne.nom">
    </div>
    <div>
      Prénom : <input type="text" v-model="personne.prenom">
    </div>
    <div>
      Age : <input type="number" v-model="personne.age">
    </div>
    <div>
      <button>
        Ajouter
      </button>
    </div>
  </form>
</template>
```

## Pour la validation de formulaires : quelques packages disponibles

- **Vee-Validate**
- Vuelidate
- ...

## Pourquoi nous utiliserons **Vee-Validate** ?

- Plus étoilé sur **GitHub** (plus populaire)
- Plus de versions, plus de mises à jour...
- Facile à utiliser et à mettre en place

© Achref

## Pourquoi nous utiliserons **Vee-Validate** ?

- Plus étoilé sur **GitHub** (plus populaire)
- Plus de versions, plus de mises à jour...
- Facile à utiliser et à mettre en place

## Lien vers une étude comparative plus détaillée

<https://npm-compare.com/vee-validate,vuelidate>

# Vue.js

## Pour installer

```
npm install vee-validate
```

© Achref EL MOU

# Vue.js

## Pour installer

```
npm install vee-validate
```

## Lien vers la documentation officielle

<https://vee-validate.logaretm.com/v4/>



# Vue.js

Commençons par importer les composants `Form` et `Field` dans la partie `script`

```
<script>
import { Form, Field } from 'vee-validate';

export default {
  name: 'PersonneAdd',
  components: {
    Form,
    Field,
  },
  data() {
    return {
      personne: { prenom: '', nom: '', age: null }
    },
  },
  methods: {
    ajouterPersonne() {
      console.log(this.personne)
    }
  },
}
</script>
```

Dans `template`, remplaçons `form` par `Form`, `input` par `Field` et supprimons le modificateur `prevent` associé à l'évènement `submit`

```
<template>
  <h1>Ajouter une nouvelle personne</h1>
  <Form @submit="ajouterPersonne">
    <div>
      Nom :
      <Field type=text name="nom" v-model="personne.nom" />
    </div>
    <div>
      Prénom :
      <Field type=text name="prenom" v-model="personne.prenom" />
    </div>
    <div>
      Age :
      <Field type=number name="age" v-model="personne.age" />
    </div>
    <div>
      <button>
        Ajouter
      </button>
    </div>
  </Form>
</template>
```

Dans `template`, remplaçons `form` par `Form`, `input` par `Field` et supprimons le modificateur `prevent` associé à l'évènement `submit`

```
<template>
  <h1>Ajouter une nouvelle personne</h1>
  <Form @submit="ajouterPersonne">
    <div>
      Nom :
      <Field type=text name="nom" v-model="personne.nom" />
    </div>
    <div>
      Prénom :
      <Field type=text name="prenom" v-model="personne.prenom" />
    </div>
    <div>
      Age :
      <Field type=number name="age" v-model="personne.age" />
    </div>
    <div>
      <button>
        Ajouter
      </button>
    </div>
  </Form>
</template>
```

Pour le composant `Field`, l'attribut `name` et la fermeture de la balise avec `/` sont obligatoires.

## Remarque

**vee-validate** permet de récupérer plus facilement les valeurs saisies dans un formulaire (sans passer par la directive `v-model`)

## Commençons par supprimer les v-model du formulaire précédent

```
<template>
  <h1>Ajouter une nouvelle personne</h1>
  <Form @submit="ajouterPersonne">
    <div>
      Nom :
      <Field type=text name="nom" />
    </div>
    <div>
      Prénom :
      <Field type=text name="prenom" />
    </div>
    <div>
      Age :
      <Field type=number name="age" />
    </div>
    <div>
      <button>
        Ajouter
      </button>
    </div>
  </Form>
</template>
```

# Vue.js

Dans `script`, supprimons la partie `data` et ajoutons un paramètre `values` dans `ajouterPersonne()` qui contiendra les valeurs saisies par l'utilisateur

```
<script>
import { Form, Field } from 'vee-validate';

export default {
  name: 'PersonneAdd',
  components: {
    Form,
    Field,
  },
  methods: {
    ajouterPersonne(values) {
      console.log(values)
    }
  },
}
</script>
```

## Question

À quoi sert-il l'objet `proxy` dans `values` ?

L'objet `proxy` est utilisé par **vee-validate** pour

- **la réactivité** : il permet de détecter les changements de valeur des champs en temps réel.
- **la gestion des erreurs** : il permet de stocker des informations supplémentaires sur chaque champ, telles que les erreurs de validation.
- **la validation à la volée** : il permet de surveiller les changements de valeur et appliquer les règles de validation en arrière-plan, sans que l'utilisateur ait besoin de soumettre le formulaire.



## Question 1

Pourrions-nous utiliser des `input` dans le Form de **vee-validate** ?

© Achref EL MOUELHI

## Question 1

Pourrions-nous utiliser des `input` dans le `Form` de **vee-validate** ?

## Réponse

Non, car `Field` de **vee-validate** :

- s'enregistre automatiquement auprès du `Form` parent de **vee-validate**,
- transmet automatiquement sa valeur à la méthode associée à la soumission du formulaire,
- gère les validations, les erreurs...

## Cependant

Le champ `input` ne s'enregistre pas automatiquement auprès de **vee-validate**.  
Donc :

- sa valeur ne sera pas disponible dans l'objet passé à la méthode associée à la soumission du formulaire,
- aucune validation automatique ne s'appliquera non plus.

# Vue.js

## Pour vider le formulaire après soumission

```
ajouterPersonne(values, actions) {  
  console.log(values)  
  actions.resetForm()  
}
```

© Achref EL MOUELHI ©

# Vue.js

## Pour vider le formulaire après soumission

```
ajouterPersonne(values, actions) {  
  console.log(values)  
  actions.resetForm()  
}
```

## Pour initialiser quelques (ou tous les) champs du formulaire

```
ajouterPersonne(values, actions) {  
  console.log(values)  
  actions.setValues({ nom: 'Doe', prenom: 'John' })  
}
```

# Vue.js

## Pour vider le formulaire après soumission

```
ajouterPersonne(values, actions) {  
  console.log(values)  
  actions.resetForm()  
}
```

## Pour initialiser quelques (ou tous les) champs du formulaire

```
ajouterPersonne(values, actions) {  
  console.log(values)  
  actions.setValues({ nom: 'Doe', prenom: 'John' })  
}
```

## Pour initialiser un seul champ du formulaire

```
ajouterPersonne(values, actions) {  
  console.log(values)  
  actions.setFieldValue('nom', 'Doe')  
}
```

# Vue.js

Pour soumettre le formulaire, il faut qu'il soit valide

- les noms et prénoms sont obligatoires et doivent commencer par une lettre en majuscule
- l'age est obligatoire et doit contenir une valeur comprise entre 18 et 120

© Achref EL MOUADJID

# Vue.js

## Pour soumettre le formulaire, il faut qu'il soit valide

- les noms et prénoms sont obligatoires et doivent commencer par une lettre en majuscule
- l'âge est obligatoire et doit contenir une valeur comprise entre 18 et 120

## Démarche

- Créer une fonction de validation pour chaque contrainte qui retourne :
  - `true` si la contrainte est respectée,
  - un message d'erreur sinon.
- Associer la fonction au `Field` via l'attribut : `rules`



Commençons par définir une méthode `validateName` pour les champs `nom` et `prenom`

```
validateName(value) {  
  if (!value) {  
    return "Ce champ est obligatoire"  
  }  
  if (value[0] < 'A' || value[0] > 'Z') {  
    return "Ce champ doit commencer par une lettre en majuscule"  
  }  
  return true  
}
```

© Achref EL MOU

Commençons par définir une méthode `validateName` pour les champs `nom` et `prenom`

```
validateName(value) {  
  if (!value) {  
    return "Ce champ est obligatoire"  
  }  
  if (value[0] < 'A' || value[0] > 'Z') {  
    return "Ce champ doit commencer par une lettre en majuscule"  
  }  
  return true  
}
```

Et une deuxième `validateAge` pour l'âge

```
validateAge(value) {  
  if (!value) {  
    return "L'age est obligatoire"  
  }  
  if (value < 18 || value > 120) {  
    return "Votre age doit être entre 18 et 120"  
  }  
  return true  
}
```

## Associations ces méthodes aux différents champs

```
<template>
  <h1>Ajouter une nouvelle personne</h1>
  <Form @submit="ajouterPersonne">
    <div>
      Nom :
      <Field type=text name="nom" :rules="validateName" />
    </div>
    <div>
      Prénom :
      <Field type=text name="prenom" :rules="validateName" />
    </div>
    <div>
      Age :
      <Field type=number name="age" :rules="validateAge" />
    </div>
    <div>
      <button>
        Ajouter
      </button>
    </div>
  </Form>
</template>
```

## Testez et vérifiez que

- le formulaire est seulement soumis si les champs sont tous valides,
- les messages d'erreur ne s'affichent pas.

# Vue.js

Pour afficher les messages d'erreurs, on commence par importer

ErrorMessage

```
<script>
import { Form, Field, ErrorMessage } from 'vee-validate';

export default {
  name: 'PersonneAdd',
  components: {
    Form,
    Field,
    ErrorMessage
  },
  methods: {
    // les méthodes précédentes
  },
}

</script>
```

# Vue.js

## Utilisons `MessageError` dans le template

```
<template>
  <h1>Ajouter une nouvelle personne</h1>
  <Form @submit="ajouterPersonne">
    <div>
      Nom :
      <Field type=text name="nom" :rules="validateName" />
      <ErrorMessage name="nom" />
    </div>
    <div>
      Prénom :
      <Field type=text name="prenom" :rules="validateName" />
      <ErrorMessage name="prenom" />
    </div>
    <div>
      Age :
      <Field type=number name="age" :rules="validateAge" />
      <ErrorMessage name="age" />
    </div>
    <div>
      <button>
        Ajouter
      </button>
    </div>
  </Form>
</template>
```

# Vue.js

Par défaut le message d'erreur est rendu dans une `span`, pour choisir une autre balise, on utilise l'attribut `as`

```
<template>
  <h1>Ajouter une nouvelle personne</h1>
  <Form @submit="ajouterPersonne">
    <div>
      Nom :
      <Field type=text name="nom" :rules="validateName" />
      <ErrorMessage name="nom" as="div" />
    </div>
    <div>
      Prénom :
      <Field type=text name="prenom" :rules="validateName" />
      <ErrorMessage name="prenom" />
    </div>
    <div>
      Age :
      <Field type=number name="age" :rules="validateAge" />
      <ErrorMessage name="age" />
    </div>
    <div>
      <button>
        Ajouter
      </button>
    </div>
  </Form>
</template>
```

# Vue.js

## Pour afficher les messages d'erreur dans un composant personnalisé

```
<template>
  <h1>Ajouter une nouvelle personne</h1>
  <Form @submit="ajouterPersonne">
    <div>
      Nom :
      <Field type=text name="nom" :rules="validateName" />
      <ErrorMessage name="nom" v-slot="{ message }">
        <CustomError>{{ message }}</CustomError>
      </ErrorMessage>
    </div>
    <div>
      Prénom :
      <Field type=text name="prenom" :rules="validateName" />
      <ErrorMessage name="prenom" />
    </div>
    <div>
      Age :
      <Field type=number name="age" :rules="validateAge" />
      <ErrorMessage name="age" />
    </div>
    <div>
      <button>
        Ajouter
      </button>
    </div>
  </Form>
</template>
```



# Vue.js

## Le composant CustomError

```
<template>
  <div class="error-box">
    <slot />
  </div>
</template>

<style scoped>
.error-box {
  color: red;
  font-weight: bold;
}
</style>
```

# Vue.js

**Sans oublier d'importer et déclarer** `CustomError`

```
import CustomError from './CustomError.vue';

export default {
  components: {
    Form,
    Field,
    ErrorMessage,
    CustomError
  },

  // + suite
}
```

# Vue.js

## Hypothèse

Supposant que le champs `age` soit présent dans plusieurs composants de notre application (bien sûr avec les mêmes contraintes).

© Achref EL MOUELHI

# Vue.js

## Hypothèse

Supposant que le champs `age` soit présent dans plusieurs composants de notre application (bien sûr avec les mêmes contraintes).

## Question

Faudrait-il (re-)définir la fonction de validation dans tous les composants ?

# Vue.js

## Hypothèse

Supposant que le champs `age` soit présent dans plusieurs composants de notre application (bien sûr avec les mêmes contraintes).

## Question

Faudrait-il (re-)définir la fonction de validation dans tous les composants ?

## Réponse

Non, on peut le définir comme validateur global.

# Vue.js

Dans un dossier `validators` (à créer dans `src`) créons le fichier `min-max.js` avec le contenu suivant

```
import { defineRule } from 'vee-validate';

defineRule('minMax', (value, [min, max]) => {
  if (!value || !value.length) {
    return "Ce champ est obligatoire";
  }
  if (Number(value) < min) {
    return `Ce champ doit contenir une valeur supérieure ou égale à ${min}`;
  }
  if (Number(value) > max) {
    return `Ce champ doit contenir une valeur inférieure ou égale à ${max}`;
  }
  return true;
});
```

# Vue.js

Dans un dossier `validators` (à créer dans `src`) créons le fichier `min-max.js` avec le contenu suivant

```
import { defineRule } from 'vee-validate';

defineRule('minMax', (value, [min, max]) => {
  if (!value || !value.length) {
    return "Ce champ est obligatoire";
  }
  if (Number(value) < min) {
    return `Ce champ doit contenir une valeur supérieure ou égale à ${min}`;
  }
  if (Number(value) > max) {
    return `Ce champ doit contenir une valeur inférieure ou égale à ${max}`;
  }
  return true;
});
```

Importons d'une manière globale ce nouveau validateur dans `main.js`

```
import '@/validators/min-max';
```

# Vue.js

## Pour utiliser le nouveau validateur dans le template

```
<template>
  <h1>Ajouter une nouvelle personne</h1>
  <Form @submit="ajouterPersonne">
    <div>
      Nom :
      <Field type=text name="nom" :rules="validateName" />
      <ErrorMessage name="nom" />
    </div>
    <div>
      Prénom :
      <Field type=text name="prenom" :rules="validateName" />
      <ErrorMessage name="prenom" />
    </div>
    <div>
      Age :
      <Field type=number name="age" rules="'minMax:18,120'" />
      <ErrorMessage name="age" />
    </div>
    <div>
      <button>
        Ajouter
      </button>
    </div>
  </Form>
</template>
```



## Remarque

Nous pourrons aussi définir un schéma de validation et l'associer au formulaire (**pas aux champs**).

© Achref EL MOUELHANI

## Remarque

Nous pourrons aussi définir un schéma de validation et l'associer au formulaire (**pas aux champs**).

## Démarche

- Définir un attribut `simpleSchema` de type objet dans la fonction `data`
- Chaque clé dans `simpleSchema` correspond à un nom de champ dans le formulaire, la valeur correspond au validateur
- Définir l'attribut `simpleSchema` comme valeur de l'attribut `:validation-schema` de la balise `Form`

# Vue.js

Dans `data`, commençons par définir un schéma de validation

```
data() {  
  return {  
    simpleSchema: {  
      nom(value) {  
        if (!value) {  
          return "Ce champ est obligatoire"  
        }  
        if (value[0] < 'A' || value[0] > 'Z') {  
          return "Ce champ doit commencer par une lettre en majuscule"  
        }  
        return true  
      },  
      prenom(value) {  
        if (!value) {  
          return "Ce champ est obligatoire"  
        }  
        if (value[0] < 'A' || value[0] > 'Z') {  
          return "Ce champ doit commencer par une lettre en majuscule"  
        }  
        return true  
      },  
      age: "minMax:18,120"  
    }  
  }  
},
```

Dans `methods`, gardons uniquement la méthode `ajouterPersonne()`

```
methods: {  
  ajouterPersonne(values) {  
    console.log(values)  
  },  
}
```

# Vue.js

Dans `template`, associons le schéma de validation au formulaire via l'attribut `:validation-schema`

```
<template>
  <h1>Ajouter une nouvelle personne</h1>
  <Form
    @submit="ajouterPersonne"
    :validation-schema="simpleSchema">
    <div>
      Nom :
      <Field type=text name="nom" />
      <ErrorMessage name="nom" />
    </div>
    <div>
      Prénom :
      <Field type=text name="prenom" />
      <ErrorMessage name="prenom" />
    </div>
    <div>
      Age :
      <Field type=number name="age" />
      <ErrorMessage name="age" />
    </div>
    <div>
      <button>
        Ajouter
      </button>
    </div>
  </Form>
</template>
```

# Vue.js

Pour initialiser les champs du formulaires avec des valeurs par défaut, on utilise `:initial-values`

```
<template>
  <h1>Ajouter une nouvelle personne</h1>
  <Form
    @submit="ajouterPersonne"
    :validation-schema="simpleSchema"
    :initial-values="valeurs">
    <div>
      Nom :
      <Field type=text name="nom" />
      <ErrorMessage name="nom" />
    </div>
    <div>
      Prénom :
      <Field type=text name="prenom" />
      <ErrorMessage name="prenom" />
    </div>
    <div>
      Age :
      <Field type=number name="age" />
      <ErrorMessage name="age" />
    </div>
    <div>
      <button>
        Ajouter
      </button>
    </div>
  </Form>
</template>
```

**Sans oublier de définir** valeurs **dans** data

```
valeurs: {  
  nom: 'Doe',  
  prenom: 'John',  
  age: 45  
}
```

# Vue.js

Pour spécifier le nom d'une méthode à exécuter dans le cas d'une soumission d'un formulaire invalide

```
<template>
  <h1>Ajouter une nouvelle personne</h1>
  <Form
    @submit="ajouterPersonne"
    :validation-schema="simpleSchema"
    :initial-values="valeurs"
    @invalid-submit="onInvalidSubmit">
    <div>
      Nom :
      <Field type=text name="nom" />
      <ErrorMessage name="nom" />
    </div>
    <div>
      Prénom :
      <Field type=text name="prenom" />
      <ErrorMessage name="prenom" />
    </div>
    <div>
      Age :
      <Field type=number name="age" />
      <ErrorMessage name="age" />
    </div>
    <div>
      <button>
        Ajouter
      </button>
    </div>
  </Form>
</template>
```



**Sans oublier de définir** `onInvalidSubmit` **dans** `methods`

```
onInvalidSubmit({ values, errors, results }) {  
  console.log(values);  
  // contenant les valeurs actuelles du formulaire  
  
  console.log(errors);  
  // contenant le nom du champ et le premier message d'erreur  
  
  console.log(results);  
  // contenant un mapping détaillé entre les champs et leurs  
    messages d'erreur  
}
```

Pour simplifier l'implémentation de validateurs, on peut utiliser **Yup**

- Librairie **JavaScript** pour la validation de valeurs
- Écrite en **JavaScript** et **TypeScript**
- Page **GitHub** : <https://github.com/jquense/yup>

© Achref

Pour simplifier l'implémentation de validateurs, on peut utiliser **Yup**

- Librairie **JavaScript** pour la validation de valeurs
- Écrite en **JavaScript** et **TypeScript**
- Page **GitHub** : <https://github.com/jquense/yup>

Pour installer

```
npm install yup
```

## Démarche

- Supprimer les méthodes de validations définies dans le schéma de validation
- Importer **Yup**
- Définir les nouveaux validateurs dans `data` en utilisant **Yup**
- Référencer les validateurs dans `template` comme valeur de l'attribut `:rules`

# Vue.js

## Nouveau contenu de la partie `script`

```
<script>
import { Form, Field, ErrorMessage } from 'vee-validate';

export default {
  name: 'PersonneAdd',
  components: {
    Form,
    Field,
    ErrorMessage
  },
  data() {
    return {
    },
  },
  methods: {
    ajouterPersonne(values) {
      console.log(values)
    },
  }
}
</script>
```

## Importons Yup

```
<script>
import { Form, Field, ErrorMessage } from 'vee-validate';
import * as yup from 'yup';

export default {
  name: 'PersonneAdd',
  components: {
    Form,
    Field,
    ErrorMessage
  },
  data() {
    return {
    },
  },
  methods: {
    ajouterPersonne(values) {
      console.log(values)
    },
  }
}
</script>
```

# Vue.js

## Utilisons Yup pour redéfinir les validateurs précédents

```
<script>
import { Form, Field, ErrorMessage } from 'vee-validate';
import * as yup from 'yup';

export default {
  name: 'PersonneAdd',
  components: {
    Form,
    Field,
    ErrorMessage
  },
  data() {
    return {
      validateAge : yup.number().required().min(18).max(120),
      validateName: yup.string().required().matches(/^[A-Z]{1}.*$/)
    }
  },
  methods: {
    ajouterPersonne(values) {
      console.log(values)
    },
  }
}
</script>
```

# Vue.js

## Le template

```
<template>
  <h1>Ajouter une nouvelle personne</h1>
  <Form @submit="ajouterPersonne">
    <div>
      Nom :
      <Field type=text name="nom" :rules="validateName" />
      <ErrorMessage name="nom" />
    </div>
    <div>
      Prénom :
      <Field type=text name="prenom" :rules="validateName" />
      <ErrorMessage name="prenom" />
    </div>
    <div>
      Age :
      <Field type=number name="age" :rules="validateAge" />
      <ErrorMessage name="age" />
    </div>
    <div>
      <button>
        Ajouter
      </button>
    </div>
  </Form>
</template>
```



# Vue.js

## Autres validateurs pour `string`

- `length`
- `url`
- `uuid`
- `lowercase`
- `uppercase`
- `trim`
- ...

## Autres validateurs pour `number`

- `lessThan`
- `moreThan`
- `positive`
- `negative`
- `integer`
- ...

## Remarque

- Par défaut, les messages d'erreurs s'affichent en anglais
- Cependant, il est possible d'utiliser `yup-locales` pour les afficher en français

© Achref L.

# Vue.js

## Remarque

- Par défaut, les messages d'erreurs s'affichent en anglais
- Cependant, il est possible d'utiliser `yup-locales` pour les afficher en français

## Pour installer

```
npm install yup-locales
```

# Vue.js

**Importons** `yup-locales` **dans** `main.js`

```
import { fr } from 'yup-locales';
```

© Achref EL MOUELHI ©

# Vue.js

**Importons** `yup-locales` **dans** `main.js`

```
import { fr } from 'yup-locales';
```

**Importons également** `setLocale`

```
import { setLocale } from 'yup';
```

# Vue.js

**Importons** `yup-locales` **dans** `main.js`

```
import { fr } from 'yup-locales';
```

**Importons également** `setLocale`

```
import { setLocale } from 'yup';
```

**Modifions ensuite la** `Locale` **et vérifions que les messages d'erreur s'affichent désormais en français**

```
setLocale(fr);
```

## Question

Comment afficher des messages d'erreur personnalisés ?



# Vue.js

Chaque validateur peut prendre comme paramètre le message à afficher en cas d'erreur

```
data() {  
  return {  
    validateAge:  
      yup  
        .number()  
        .required("L'age est obligatoire")  
        .min(18, "L'age min est 18")  
        .max(120, "L'age max est 120"),  
    validateName:  
      yup  
        .string()  
        .required("Ce champ est obligatoire")  
        .matches(/^[A-Z]{1}.*$/, "Première lettre en majuscule")  
  },  
}
```

# Vue.js

Pour récupérer le seuil indiqué dans certaines fonctions comme `min`, `max`...

```
data() {  
  return {  
    validateAge:  
      yup  
        .number()  
        .required("L'age est obligatoire")  
        .min(18, (params) => `Vous avez saisi ${params.value} et l'age min est  
          ${params.min}`)  
        .max(120, "L'age max est 120"),  
    validateName:  
      yup  
        .string()  
        .required("Ce champ est obligatoire")  
        .matches(/^[A-Z]{1}.*$/, "Première lettre en majuscule")  
  },  
}
```

# Vue.js

Pour afficher un message d'erreur relatif au type

```
data() {  
  return {  
    validateAge:  
      yup  
        .number()  
        .typeError("Ce champ ne doit contenir que des caractères numériques")  
        .required("L'age est obligatoire")  
        .min(18, (params) => `Vous avez saisi ${params.value} et l'age min est  
          ${params.min}`)  
        .max(120, "L'age max est 120"),  
    validateName:  
      yup  
        .string()  
        .required("Ce champ est obligatoire")  
        .matches(/^[A-Z]{1}.*/ , "Première lettre en majuscule")  
  }  
},
```

## Remarque

Nous pourrions aussi définir un schéma de validation et l'associer au formulaire (**pas aux champs**).

# Vue.js

Commençons par définir un schéma de validation : chaque élément doit porter le nom d'un champ su formulaire

```
data() {  
  return {  
    simpleSchema: yup.object({  
      age:  
        yup  
          .number()  
          .required("L'age est obligatoire")  
          .min(18, "L'age min est 18")  
          .max(120, "L'age max est 120"),  
      nom:  
        yup  
          .string()  
          .required("Ce champ est obligatoire")  
          .matches(/^[A-Z]{1}.*/ , "Première lettre en majuscule"),  
      prenom:  
        yup  
          .string()  
          .required("Ce champ est obligatoire")  
          .matches(/^[A-Z]{1}.*/ , "Première lettre en majuscule"),  
    })  
  },  
}
```

# Vue.js

Dans `template`, associons le schéma de validation au formulaire via l'attribut `:validation-schema`

```
<template>
  <h1>Ajouter une nouvelle personne</h1>
  <Form @submit="ajouterPersonne" :validation-schema="simpleSchema">
    <div>
      Nom :
      <Field type=text name="nom" />
      <ErrorMessage name="nom" />
    </div>
    <div>
      Prénom :
      <Field type=text name="prenom" />
      <ErrorMessage name="prenom" />
    </div>
    <div>
      Age :
      <Field type=number name="age" />
      <ErrorMessage name="age" />
    </div>
    <div>
      <button>
        Ajouter
      </button>
    </div>
  </Form>
</template>
```

## Gestion de formulaires ?

- Vider les champs après soumission
- Initialiser les champs d'un formulaire
- Désactiver le bouton de soumission si formulaire invalide
- Spécifier quand on affiche le message d'erreur

## Utilisation de `ref`

- Déclarer une `ref` au niveau du formulaire
- Récupérer des méthodes et des propriétés du formulaire



# Vue.js

Dans `template`, ajoutons une référence de template sur le formulaire

```
<template>
  <h1>Ajouter une nouvelle personne</h1>
  <Form @submit="ajouterPersonne" :validation-schema="simpleSchema" ref="personneForm"
  >
    <div>
      Nom :
      <Field type="text" name="nom" />
      <ErrorMessage name="nom" />
    </div>
    <div>
      Prénom :
      <Field type="text" name="prenom" />
      <ErrorMessage name="prenom" />
    </div>
    <div>
      Age :
      <Field type="number" name="age" />
      <ErrorMessage name="age" />
    </div>
    <div>
      <button>
        Ajouter
      </button>
    </div>
  </Form>
</template>
```

# Vue.js

## Pour vider le formulaire après soumission

```
ajouterPersonne(values) {  
  console.log(values)  
  this.$refs.personneForm.resetForm()  
}
```

© Achref EL MOUELHI

# Vue.js

## Pour vider le formulaire après soumission

```
ajouterPersonne(values) {  
  console.log(values)  
  this.$refs.personneForm.resetForm()  
}
```

## Pour initialiser quelques (ou tous les) champs du formulaire

```
mounted() {  
  this.$refs.personneForm.setValues({ nom: 'Doe', prenom: 'John' })  
}
```

# Vue.js

## Pour vider le formulaire après soumission

```
ajouterPersonne(values) {  
  console.log(values)  
  this.$refs.personneForm.resetForm()  
}
```

## Pour initialiser quelques (ou tous les) champs du formulaire

```
mounted() {  
  this.$refs.personneForm.setValues({ nom: 'Doe', prenom: 'John' })  
}
```

## Pour initialiser un seul champ du formulaire

```
mounted() {  
  this.$refs.personneForm.setFieldValue('nom', 'Doe')  
}
```

## Utilisation de `v-slot`

- Déclarer `v-slot` au niveau du formulaire permet de gérer les erreurs du formulaires
- Déclarer `v-slot` au niveau d'un du formulaire permet de gérer les erreurs de ce champ

# Vue.js

Dans `template`, ajoutons `v-slot` sur le formulaire

```
<template>
  <h1>Ajouter une nouvelle personne</h1>
  <Form
    @submit="ajouterPersonne"
    :validation-schema="simpleSchema"
    ref="personneForm"
    v-slot="{ errors }">

    <!-- le contenu précédent -->

    <div>
      <button>
        Ajouter
      </button>
    </div>
  </Form>
</template>
```

`errors` peut être utilisé de la manière suivante pour lister les messages d'erreur

```
<template>
  <h1>Ajouter une nouvelle personne</h1>

  <Form @submit="ajouterPersonne" v-slot="{ errors }" :validation-schema="
    simpleSchema" ref="personneForm">
    <div>
      <template v-if="Object.keys(errors).length > 0">
        <p>Merci de corriger les erreurs suivantes</p>
        <ul>
          <li v-for="(value, key) in errors">
            {{ key }} : {{ value }}
          </li>
        </ul>
      </template>
    </div>
    <div>
      Nom : <Field type=text name="nom" />
    </div>
    <div>
      Prénom : <Field type=text name="prenom" />
    </div>
    <div>
      Age : <Field type=number name="age" />
    </div>
    <div>
      <button>
        Ajouter
      </button>
    </div>
  </Form>
</template>
```

# Vue.js

## Pour désactiver le bouton de soumission si formulaire invalide

```
<template>
  <h1>Ajouter une nouvelle personne</h1>
  <Form
    @submit="ajouterPersonne"
    :validation-schema="simpleSchema"
    ref="personneForm"
    v-slot="{ errors, meta }">

    <!-- le contenu précédent -->

    <div>
      <button :disabled="!meta.valid">
        Ajouter
      </button>
    </div>
  </Form>
</template>
```



# Vue.js

## Pour récupérer les valeurs du formulaire

```
<template>
  <h1>Ajouter une nouvelle personne</h1>
  <Form
    @submit="ajouterPersonne"
    :validation-schema="simpleSchema"
    ref="personneForm"
    v-slot="{ errors, meta, values }">

    {{ values }}

    <!-- le contenu précédent -->

    <div>
      <button :disabled="!meta.valid">
        Ajouter
      </button>
    </div>
  </Form>
</template>
```

# Vue.js

## Pour afficher une bordure rouge pour un champ invalide

```
<div>  
  <Field :class="{ 'red-left-border': errors.nom }" />  
  <ErrorMessage name=nom />  
</div>
```

© Achref EL MOUL

# Vue.js

## Pour afficher une bordure rouge pour un champ invalide

```
<div>  
  <Field :class="{ 'red-left-border': errors.nom }" />  
  <ErrorMessage name=nom />  
</div>
```

## Et la classe .red-left-border

```
<style scoped>  
.red-left-border {  
  border-left: 5px solid red;  
}  
</style>
```

## Configuration de la validation

Spécifier quel évènement déclenchera la validation d'un

- champ
- formulaire

## Rappel : différences entre quelques évènements

- `blur` : se déclenche lorsque le champ perd le focus même si la valeur reste inchangée
- `input` : se déclenche à chaque changement de valeur
- `change` : se déclenche lorsque le champ perd le focus et la valeur a changé

© Achref EL MOUADJIB

## Rappel : différences entre quelques évènements

- `blur` : se déclenche lorsque le champ perd le focus même si la valeur reste inchangée
- `input` : se déclenche à chaque changement de valeur
- `change` : se déclenche lorsque le champ perd le focus et la valeur a changé

## Remarques

- `input` : se déclenche suite à un changement du contenu textuel `input:text`, `input:password`, `textarea...`
- `change` : se déclenche lorsqu'on perd le focus et suite à un changement de valeur dans un `input`, `textarea`, `select...`

## Explication

- Par défaut dans **vee-validate**, l'affichage des messages d'erreur est lié à l'évènement `blur`
- **vee-validate** nous permet de changer cet évènement déclencheur

# Vue.js

Pour activer l'affichage des messages d'erreur sur le champ Nom à la saisie

```
<template>
  <h1>Ajouter une nouvelle personne</h1>
  <Form
    @submit="ajouterPersonne"
    :validation-schema="simpleSchema"
    ref="personneForm"
    v-slot="{ errors, meta }">
    <div>
      Nom :
      <Field type=text name="nom" :validateOnInput="true" />
      <ErrorMessage name="nom" />
    </div>
    <div>
      Prénom :
      <Field type=text name="prenom" />
      <ErrorMessage name="prenom" />
    </div>
    <div>
      Age :
      <Field type=number name="age" />
      <ErrorMessage name="age" />
    </div>
    <div>
      <button :disabled="!meta.valid">
        Ajouter
      </button>
    </div>
  </Form>
</template>
```



Pour le faire d'une manière globale (code à ajouter dans `main.js`)

```
import { configure } from 'vee-validate'

configure({
  validateOnBlur: true,
  validateOnChange: true,
  validateOnInput: true,
});
```

## Question

Et si on voulait construire le formulaire d'une manière dynamique ?

© Achref EL MOUELHANI

## Question

Et si on voulait construire le formulaire d'une manière dynamique ?

## Démarches

- Indiquer les informations nécessaires dans un tableau (dans la partie `script`)
- Itérer sur ce tableau et construire dynamiquement le tableau dans la partie `template`

Déclarons les champs du formulaire dans un objet `formSchema` dans `data`

```
formSchema: {
  fields: [
    {
      label: 'Nom',
      name: 'nom',
      as: 'input',
      rules: yup
        .string()
        .required('Ce champ est obligatoire `')
        .matches(/^[A-Z]{1}.*/ , "Première lettre en majuscule"),
    },
    {
      label: 'Prénom',
      name: 'prenom',
      as: 'input',
      rules: yup
        .string()
        .required('Ce champ est obligatoire `')
        .matches(/^[A-Z]{1}.*/ , "Première lettre en majuscule"),
    },
    {
      label: 'Age',
      name: 'age',
      as: 'input',
      rules: yup
        .number()
        .required("L'age est obligatoire")
        .min(18, "L'age min est 18")
        .max(120, "L'age max est 120")
    },
  ],
}
```

Dans `template`, construisons le formulaire d'une manière dynamique

```
<template>
  <h1>Ajouter une nouvelle personne</h1>

  <Form @submit="ajouterPersonne" v-slot="{ errors, meta }" ref="personneForm">
    <div v-for="field in formSchema.fields" :key="field.name">
      <label :for="field.name">{{ field.label }}</label>
      <Field :as="field.as" :id="field.name" :name="field.name" :rules="field.rules"/>
      <ErrorMessage :name="field.name" />
    </div>
    <div>
      <button :disabled="!meta.valid">
        Ajouter
      </button>
    </div>
  </Form>
</template>
```

## Récapitulatif

- En cliquant sur le bouton `Ajouter`, un objet s'affiche dans la console du navigateur
- Format de l'objet :  

```
{ nom: 'Wick', prenom: 'John', age: '45' }
```

© Achref EL

## Récapitulatif

- En cliquant sur le bouton `Ajouter`, un objet s'affiche dans la console du navigateur
- Format de l'objet :  

```
{ nom: 'Wick', prenom: 'John', age: '45' }
```

## Dans l'objet récupéré

- La clé est un string
- La valeur est de type simple : `string`, `number`...

## Question

Et si une personne avait une adresse et pouvait pratiquer jusqu'au 3 sports ?



## Question

Comment tout récupérer dans un seul objet de la forme :

```
{  
  nom: 'Wick',  
  prenom: 'John',  
  age: '45',  
  adresse: { rue: 'Paradis', ville: 'Marseille', codePostal: '13006' },  
  sports: ['foot', 'tennis', 'basket']  
}
```

© Actif

## Question

Comment tout récupérer dans un seul objet de la forme :

```
{  
  nom: 'Wick',  
  prenom: 'John',  
  age: '45',  
  adresse: { rue: 'Paradis', ville: 'Marseille', codePostal: '13006' },  
  sports: ['foot', 'tennis', 'basket']  
}
```

## Réponse

Il faut tout spécifier dans l'attribut `name` du formulaire.

# Vue.js

Nouveau contenu de `template` : la valeur de l'attribut `name` de la partie imbriquée est préfixée par `adresse`.

```
<template>
  <h1>Ajouter une nouvelle personne</h1>
  <Form @submit="ajouterPersonne" :validation-schema="simpleSchema">
    <!-- contenu précédent du formulaire -->
    <div>
      <h2>Adresse</h2>
      <div>
        Rue :
        <Field type=text name="adresse.rue" />
      </div>
      <div>
        Ville :
        <Field type=text name="adresse.ville" />
      </div>
    </div>
    <div>
      <button>
        Ajouter
      </button>
    </div>
  </Form>
</template>
```

# Vue.js

## Et pour les validateurs

```
simpleSchema: yup.object({  
  nom: validate('Nom'),  
  prenom: validate('Prénom'),  
  age: yup  
    .number()  
    .required()  
    .min(15, (args) => `${args.value} doit être supérieur à ${args.min}`)  
    .max(150),  
  adresse: yup.object({  
    rue: validate('Rue'),  
    ville: validate('Ville'),  
  })  
})
```

© Achref EL

# Vue.js

## Et pour les validateurs

```
simpleSchema: yup.object({
  nom: validate('Nom'),
  prenom: validate('Prénom'),
  age: yup
    .number()
    .required()
    .min(15, (args) => `${args.value} doit être supérieur à ${args.min}`)
    .max(150),
  adresse: yup.object({
    rue: validate('Rue'),
    ville: validate('Ville'),
  })
})
```

## Pour afficher une bordure rouge pour un champ imbriqué invalide

```
<div>
  <label for="ville">Ville</label>
  <Field id="ville"
    name="adresse.ville"
    :class="{ 'red-left-border': errors['adresse.ville'] }"
  />
  <ErrorMessage name="adresse.ville" />
</div>
```

# Vue.js

Nouveau contenu de `template` : la valeur de l'attribut `name` de la partie imbriquée contient l'opérateur `[]`

```
<template>
  <h1>Ajouter une nouvelle personne</h1>
  <Form @submit="ajouterPersonne" :validation-schema="simpleSchema">
    <!-- contenu précédent du formulaire avec la partie pour Adresse -->
    <div>
      <h2>Sports</h2>
      <div>
        Sport 1 :
        <Field type=text name="sports[0]" />
      </div>
      <div>
        Sport 2 :
        <Field type=text name="sports[1]" />
      </div>
      <div>
        Sport 3 :
        <Field type=text name="sports[2]" />
      </div>
    </div>
    <div>
      <button>
        Ajouter
      </button>
    </div>
  </Form>
</template>
```

## Exercice

Faites les modifications nécessaires pour permettre à l'utilisateur d'ajouter 0, 1 ou plusieurs sports.

## Correction

```
<template>
  <h1>Ajouter une nouvelle personne</h1>
  <Form @submit="ajouterPersonne" :validation-schema="simpleSchema">
    <!-- contenu précédent du formulaire avec la partie pour Adresse -->
    <div>
      <h2>Sports</h2>
      <div v-for="(elt, index) in i" :key="index">
        Sport {{ index + 1 }} :
        <Field type=text :name="`sports[${index}]`" />
      </div>
      <button type="button" @click="() => i++">
        Ajouter un sport
      </button>
    </div>
    <div>
      <button>
        Ajouter
      </button>
    </div>
  </Form>
</template>
```



## Correction

```

<template>
  <h1>Ajouter une nouvelle personne</h1>
  <Form @submit="ajouterPersonne" :validation-schema="simpleSchema">
    <!-- contenu précédent du formulaire avec la partie pour Adresse -->
    <div>
      <h2>Sports</h2>
      <div v-for="(elt, index) in i" :key="index">
        Sport {{ index + 1 }} :
        <Field type=text :name="`sports[${index}]`" />
      </div>
      <button type="button" @click="() => i++">
        Ajouter un sport
      </button>
    </div>
    <div>
      <button>
        Ajouter
      </button>
    </div>
  </Form>
</template>

```

## Remarque

N'oublions pas de déclarer `i` dans `data` avec comme valeur initiale 0.