VeeValidate

install & setup

install & setup

VeeValidate — самая популярная библиотека форм Vue.js. Она заботится об отслеживании значений, валидации, ошибках и многом другом.
Также мы должны установить Yup - конструктор схем для анализа и проверки значений во время выполнения.

```
yarn add vee-validate
# or
npm install vee-validate --save
```

```
yarn add @vee-validate/yup
# or
npm i @vee-validate/yup
```

Declaring forms

Declaring forms

Вы можете объявлять формы с помощью функции **useForm**. Вызов **useForm** предоставляет контекст формы любому дочернему компоненту. Это означает, что **useForm** можно вызвать в компоненте только один раз. Создание контекста формы выполняет несколько задач:

1) Действует как сборщик значений для всех полей, которые вы объявите как дочерние компоненты.
2) Проверяет поля и объединяет ошибки.
3) Собирает достоверные, затронутые и измененные состояния всех полей.

```
<script setup lang="ts">
import { useForm } from 'vee-validate';

const { values } = useForm();
</script>
<template>
  <pre>{{ values }}</pre>
</template>
```

HTML Inputs

useForm предоставляет функцию **defineField**. Эта функция принимает путь к полю и возвращает значение поля и объект атрибутов. Объект **emailAttrs** содержит атрибуты или прослушиватели событий, которые нужно привязать в элемент управления вводом, который включает настраиваемые триггеры проверки и многое другое.

```
<script setup>
import { useForm } from 'vee-validate';

const { values, defineField } = useForm();

const [email, emailAttrs] = defineField('email');
</script>

<template>
  <input v-model="email" v-bind="emailAttrs" type="text" />

  <pre>values: {{ values }}</pre>
</template>
```

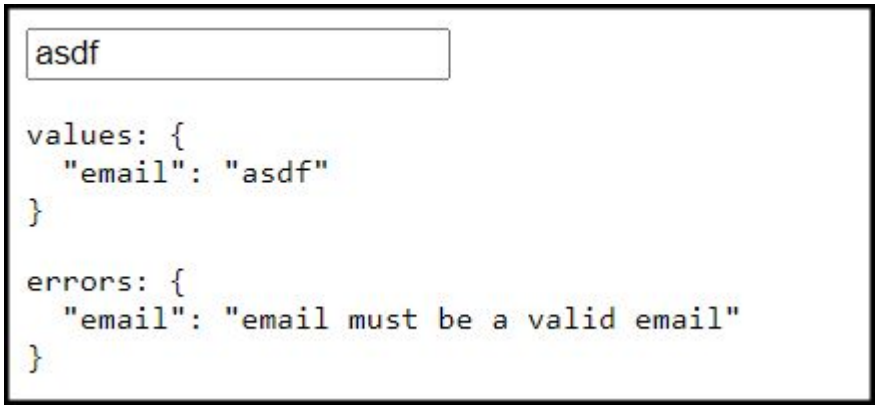
Обратите внимание, что по мере ввода данных **values** автоматически обновляются, при этом values readonly обьект

Validation schema

Мы можем добавить в форму схему валидации используя yup. Поле **email** должно быть строкой, более того корректным email и оно обязательно

```vue
<script setup>
import { useForm } from 'vee-validate';
import * as yup from 'yup';

const { values, errors, defineField } = useForm({
  validationSchema: yup.object({
    email: yup.string().email().required(),
  }),
});

const [email, emailAttrs] = defineField('email');
</script>

<template>
  <input v-model="email" v-bind="emailAttrs" />

  <pre>values: {{ values }}</pre>
  <pre>errors: {{ errors }}</pre>
</template>
```

Preview:



Configuration

По умолчанию VeeValidate валидирует “агрессивно”, то есть при изменении модели. Мы можем изменить это поведение, используя **validateOnModelUpdate**

```vue
<script setup lang="ts">
import { useForm } from 'vee-validate'
import * as yup from 'yup'

const { values, errors, defineField } = useForm<{ email: string }>({
  validationSchema: yup.object({
    email: yup.string().email().required()
  })
})

const [email, emailProps] = defineField('email', {
  validateOnModelUpdate: false
})
</script>

<template>
  <form>
    <input v-model="email" v-bind="emailProps" type="text" />
    <div>{{ errors.email }}</div>

    <button type="submit" @click.prevent="">Submit!</button>
  </form>

  <pre>values: {{ values }}</pre>
</template>
```

Или задать динамическое значение, валидировать агрессивно только если уже есть ошибки ввода:

```vue
const [email, emailAttrs] = defineField('email', state => {
  return {
    validateOnModelUpdate: state.errors.length > 0,
  };
});
```

Configuration

Также можно задать начальные значения формы или начальные ошибки:

```
const { values } = useForm({
  initialValues: {
    email: 'test@example.com',
    password: 'p@$$w0rd',
  },
});
```

```
const { values } = useForm({
  initialErrors: {
    email: 'email required',
    age: 'age required'
  }
});
```

Configuration

Вы можете установить значение любого поля, используя setFieldValue или setValues, возвращаемые useForm.

```
const { setFieldValue, setValues } = useForm();

setFieldValue('fieldName', 'value');

setValues({
  fieldName: 'value',
});
```

Устанавливает значение определенного поля в значениях формы.

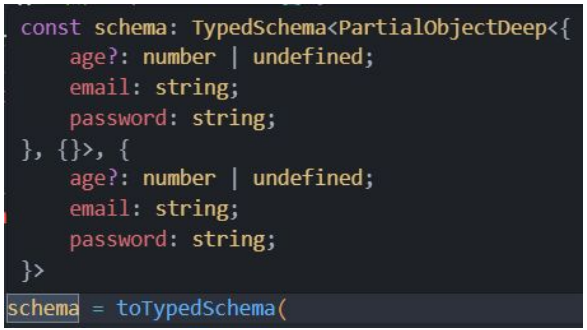
Объединяет данный объект с текущими значениями формы.

Yup

Yup

Пакет **@vee-validate/yup** предоставляет функцию **toTypedSchema**, которая позволяет vee-validate выводить типы формы.

```
const schema = toTypedSchema(
  yup.object({
    email: yup.string().required().email().default('foo@bar.baz'),
    password: yup.string().min(1).max(255).required(),
    age: yup.number().min(15).max(100)
  })
)
```



Больше примеров с yup на [github]

Handling Forms

Form Metadata

useForm возвращает объект метаданных, содержащий полезную информацию о форме.

```
const { meta } = useForm();
meta.value.dirty;
meta.value.pending;
meta.value.touched;
meta.value.valid;
meta.value.initialValues;
```

1) **valid**: статус валидации формы.
2) **touched**: true если хотя бы одно поле было сфокусированным
3) **dirty**: true если значение хотя бы одного поля было обновлено.
4) **pending**: true если хотя бы одно поле все еще ожидает проверки.
5) **initialValues**: начальные значения всех полей. Это объект, ключами которого являются имена полей.

Handling Submissions

VeeValidate позволяет обрабатывать submit форм.

```
<script setup>
import { useForm } from 'vee-validate';
import * as yup from 'yup';

const { errors, handleSubmit, defineField } = useForm({
  validationSchema: yup.object({
    email: yup.string().email().required()
  }),
});

const onSubmit = handleSubmit(values => {
  console.log(values);
});

const [email, emailAttrs] = defineField('email');
</script>

<template>
  <form @submit="onSubmit">
    <input type="email" v-model="email" v-bind="emailAttrs" />
    <div>{{ errors.email }}</div>

    <button>Submit</button>
  </form>
</template>
```

handleSubmit выполнит коллбэк только если все поля валидны, причем **onSubmit** можно вызвать либо вручную, либо через событие @submit.

Более того возвращаемая функция **onSubmit** автоматически вызывает **event.preventDefault**, поэтому самостоятельно этого делать не нужно

Handling Submissions

Вторым аргументом **handleSubmit** принимает коллбэк для случая, когда сабмит формы не может быть выполнен из за ошибок валидации. В таком случае мы можем проскроллить экран пользователя до ошибки

script:

```
const onSubmit = handleSubmit(
  values => console.log(values),
  ({ errors }) => {
    const firstError = Object.keys(errors)[0];
    const el = document.querySelector(`[name="${firstError}"]`);
    el?.scrollIntoView({
      behavior: 'smooth',
    });
    el.focus();
  },
);
```

template:

```
<input
  v-model="email"
  v-bind="emailAttrs"
  name="email"
  type="email"
/>
<span>{{ errors.email }}</span>
```

Custom Inputs

useField

До сих пор мы использовали **useForm** в связке с **defineField** Однако при таком подходе обычно требуется много шаблонного кода. На практике всегда используют **useField** - декларативный способ управления полями.

```
const { values, errors, defineField } = useForm()

const [email, emailAttrs] = defineField('email')
```

VS

```
const { values } = useForm()

const { value, errorMessage } = useField('email')
```

Custom input

useField работает так же, как и **defineField**, но теперь, мы не должны предоставлять компоненту MyInput лишние пропсы. **useForm** провайдит контекст формы, а **useField** просто его инъектирует

MyInput.vue

```vue
<script setup lang="ts">
import { useField } from 'vee-validate'

const props = defineProps<{
  name: string
}>()
const { value, errorMessage } = useField(() => props.name)
</script>

<template>
  <div>
    <input v-model="value" />
    {{ errorMessage }}
  </div>
</template>
```

ParentComponent.vue

```vue
<script setup lang="ts">
import { useForm } from 'vee-validate'
import { toTypedSchema } from '@vee-validate/yup'
import * as yup from 'yup'

const schema = toTypedSchema(yup.object({
    email: yup.string().email().required()
}))
const { handleSubmit } = useForm({
  validationSchema: schema
})
const onSubmit = handleSubmit(console.log)
</script>

<template>
  <form @submit="onSubmit">
    <MyInput name="email" />
    <button>Submit</button>
  </form>
</template>
```

Triggers

Handling Events

Функция useField предоставляет некоторые функции-обработчики, каждая из которых обрабатывает определенный аспект проверки:

1) **handleChange**: обновляет значение поля, можно настроить для запуска проверки или автоматического обновления значения.
2) **handleBlur**: обновляет флаг мета.touched, не запускает проверку.

```
const { handleChange, handleBlur } = useField('someField');
```

Handling Events

В этом примере мы валидируем по событию @input, что делает проверку агрессивной:

```
<script setup lang="ts">
import { useField } from 'vee-validate'

const props = defineProps<{
  name: string
  type: string
}>()
const {
  errorMessage, value, handleChange
} = useField(() => props.name)
</script>

<template>
  <input @input="handleChange" :value="value" type="text" />
  <span>{{ errorMessage }}</span>
</template>
```

Или мы можем сделать нашу проверку ленивой, изменив прослушиватель на @change (когда пользователь покидает поле ввода):

```
<div>
  <input
    @change="handleChange"
    :value="value"
    type="text"
  />
  <span>{{ errorMessage }}</span>
</div>
```

Подробнее в документации

Error Messages

Error Messages

Мы уже видели, как отображать ошибки с помощью **useForm**. С помощью **useField** мы можем использовать **errorMessage** - ссылку на ошибку или **errors** - массив ошибок

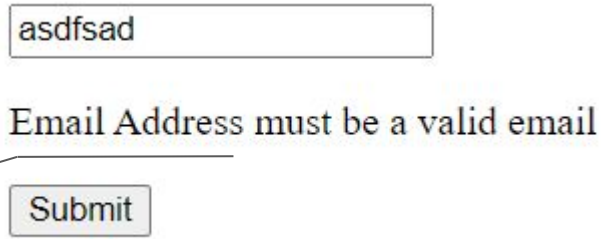
```
const { errorMessage, value } = useField('fieldName', yup.string().required());
// contains the error message if available
errorMessage.value;
```

```
const { errors, value } = useField('fieldName', yup.string().required());
// contains an array of error messages, otherwise empty array
errors.value;
```

Error Messages

Мы также можем кастомизировать лейбл нашего поля использую yup.

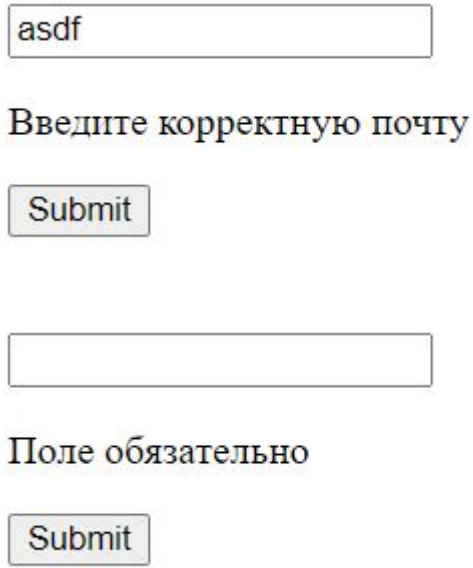
```vue
<script setup lang="ts">
import { useForm } from 'vee-validate'
import * as yup from 'yup'
import MyInput from './MyInput.vue'

const { handleSubmit } = useForm({
  validationSchema: yup.object({
    userEmail: yup.string().email().required().label('Email Address')
  })
})

const onSubmit = handleSubmit(console.log)
</script>

<template>
  <form @submit="onSubmit">
    <MyInput name="userEmail" />
    <button>Submit</button>
  </form>
</template>
```



Error Messages

Или написать собственное сообщение ошибки

```
const { handleSubmit } = useForm({
  validationSchema: yup.object({
    userEmail: yup
      .string()
      .email('Введите корректную почту')
      .required('Поле обязательно')
  })
})
```



Field-level Meta

Error Messages

С каждым полем связаны метаданные. Meta-свойство, возвращаемое из useField, содержит информацию о поле:

```
const { meta } = useField('fieldName');
meta.dirty;
meta.pending;
meta.touched;
meta.valid;
meta.initialValue;
```

1) **valid**: валидность поля
2) **touched**: было ли поле затронуто.
3) **dirty**: если значение поля было обновлено, вы не можете изменить его значение.
4) **pending**: если проверка поля все еще выполняется, это полезно для длительной асинхронной проверки.
5) **InitialValue**: начальное значение поля

Meta свойства поля также readonly. **touched** можно изменить с помощью одной из функций **setTouched** или **handleBlur**, все остальные метасвойства автоматически обновляются при валидации поля или при изменении его значения.
Подробнее в документации

Спасибо за внимание!