

Komplexe Formulare mit Vue.js

Norbert D. Frank

For-mu-lar

standardisiertes Mittel zur Erfassung,
Ansicht und Aufbereitung von Daten

Komplexe Formulare mit Vue.js

kom-plex

vielschichtig; viele verschiedene Dinge
umfassend

Vue.js

clientseitiges JavaScript-
Webframework zum Erstellen von
Webanwendungen nach dem MVVM-
Muster

Formulare in der modernen Webentwicklung

Typische Assoziation mit „Formularen“:

┌

2018

└

1	<input type="checkbox"/> Einkommensteuererklärung	<input type="checkbox"/> Festsetzung der Arbeitnehmer-Sparzulage	Eingangsstempel
2	<input type="checkbox"/> Erklärung zur Festsetzung der Kirchensteuer auf Kapitalerträge	<input type="checkbox"/> Erklärung zur Feststellung des verbleibenden Verlustvortrags	
3	Steuernummer <input type="text"/>		
An das Finanzamt			
4	<input type="text"/>		
5	Bei Wohnsitzwechsel: bisheriges Finanzamt <input type="text"/>		
6	Allgemeine Angaben		Telefonische Rückfragen tagsüber unter Nr. <input type="text"/>
7	Steuerpflichtige Person (stpfl. Person), nur bei Zusammenveranlagung: Ehemann oder Person A *) (Ehegatte A / Lebenspartner[in] A nach dem LPartG) Identifikationsnummer (IdNr.) <input type="text"/>		
8	Name <input type="text"/>		<input type="text"/> *) Bitte Anleitung beachten.
9	Vorname <input type="text"/>		Geburtsdatum <input type="text"/>
	Religionsschlüssel:		

Formulare in der modernen Webentwicklung

Auch ein Formular:

Konto erstellen

Ihr Name

E-Mail

Passwort

[i](#) Passwörter müssen mindestens 6 Zeichen lang sein.

Passwort nochmals eingeben

Erstellen Sie Ihr Amazon-Konto

Formulare auf Smartphones:

Input

First Name

Last Name

Create Account

Und noch eins:

W

Codex Sample

0 + New

Howdy, wpac

Dashboard

Posts

All Posts

Add New

Categories

Tags

Media

Pages

Comments

Appearance

Plugins

Users

Tools

Settings

Collapse menu

Add New Post

Enter title here

Add Media

VisualText

B

I

ABC

☰

☰

☰

☰

☰

☰

☰

☰

☰

☰

Path: p

Word count: 0

Excerpt

Send Trackbacks

Custom Fields

Discussion

Publish

Save Draft

Preview

Status: **Draft** [Edit](#)

Visibility: **Public** [Edit](#)

Publish immediately [Edit](#)

Move to Trash

Publish

Format

Standard

Aside

Image

Video

Audio

Quote

Link

Gallery

Categories

All Categories

Most Used

Uncategorized

+ Add New Category

Tags

Oder das:

Create Issue

⚙️ Configure Fields ▾

Project *

Support (SUP) ▾

Issue Type *

? Support request ▾

?

Summary *

Affects Version/s

▾

Start typing to get a list of possible matches or press down to select.

Epic Link

▾

Choose an epic to assign this issue to.

Priority

✔ Major ▾

?

Assignee

Automatic ▾

Assign to me

Description *

Style ▾

B I U A ▾ A° ▾

🔗 ▾ 📎 ▾

☰ ☷

😊 ▾ + ▾

⬆

☐ Create another

Create

Cancel

Formulare in der modernen Webentwicklung

Formulardefinition in diesem Talk:

*Erfassung eines in sich zusammenhängenden
Datensatzes*

(auch über mehrere Seiten, verschachtelt, ggf. mit Binärdaten)

- Developer, Consultant, IT-Architekt @ Lucom GmbH
- Vue-Enthusiast (und Web im allgemeinen)
- @norbertdfrank

LUCOM



Zentrales Ziel

Validierung, Eingabehilfen

*Sinnvolle Anordnung, Vorbelegung,
die richtigen Controls, Zeitpunkt der
Validierung, Usability*

*Korrekte Daten schnell und intuitiv
erfassen und bearbeiten*

*Für die reine Bearbeitung bestehender
Datensätze kann in manchen Fällen eine
andere Darstellung sinnvoll sein als für die
Erfassung.*

Formulare mit Vue.js

Typische Herausforderungen bei komplexen Formularen:

- Komplexe Validierungen
- Integration in zentrales State-Management
- Dynamik (wenn.. dann.. für Sichtbarkeit, Editierbarkeit, Validierung)
- Eigene Controls
- Formatierung von Formularwerten (Zahlen, Datum, Währungen)
- Usability & User Experience

Kritische Erfolgsfaktoren

Im Kern nur zwei Themen:

1. Stellenwert von Usability & UX im Projekt

Darf Zeit und Fokus darauf verwendet werden, welcher Anspruch besteht, wie wird während der Entwicklung bewertet und erprobt

2. Technische Möglichkeiten

Bei Einsatz von Vue.js gibt es keine Ausrede bzgl. technischer Machbarkeit 😊

Formulare mit Vue.js

The Basics

v-model

Input-Controls werden mit v-model an ein Attribut im Modell gebunden
(two-way data binding)

```
Vue.component('simple-form', {  
  template: `<input type="text" v-model="greeting">`,  
  data() {  
    return {  
      greeting: ""  
    }  
  }  
})
```

v-model

v-model ist nur eine Kurzform für:

```
Vue.component("simple-form", {  
  template: `<input type="text"  
    v-bind:value="greeting"  
    v-on:input="greeting = $event.target.value"  
  >`,  
  data() {  
    return {  
      greeting: ""  
    };  
  }  
});
```

Formulare in Vue.js

- Eingabe-Controls werden per v-model an das Modell gebunden
- Es gibt keine eigene Forms-API in Vue.js (im Gegensatz z.B. zu Angular)
- In Vue-Templates kann das normale HTML-Forms-Element genutzt werden
- Eigene Controls lassen sich leicht umsetzen

Tipp: Modifiers (<https://vuejs.org/v2/guide/forms.html#Modifiers>)

.lazy

.number

.trim

Formulare in Vue.js

*„Also einfach alle Input-Controls per
v-model an das Modell binden und
das Formular ist fertig?“*

.. nun ja, etwas mehr gehört üblicherweise schon dazu.

Validierung

...Kontrolle ist besser

Validierung

Warum überhaupt validieren?

1. Nutzern helfen
2. Fehler frühzeitig vermeiden
3. Datenqualität erhöhen
4. Sicherheit (Achtung: bezieht sich nur auf serverseitige Validierung!)

Validierung

Wo validieren:

1. Im Browser

*Anwender bei der Eingabe
unterstützen*

2. Im Server

Valide Daten sicherstellen

Wann kann validiert werden:

- *Beim Tippen*
- *Beim Verlassen eines Controls*
- *Beim Seitenwechsel*
- *Beim Abschicken*

Achtung: Validierung im Browser ersetzt niemals serverseitige Validierung!

Client-Side-Validation mit Vue.js

Möglichkeiten:

1. HTML Constraint Validation

https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/HTML5/Constraint_validation

2. Template-basiert: VeeValidate

<https://baianat.github.io/vee-validate/>

3. Modell-basiert: Vuelidate

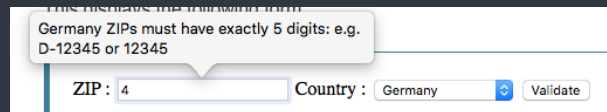
<https://monterail.github.io/vuelidate/>

Zusatzoption: selber per JavaScript validieren.

HTML Constraint Validation

- Mit HTML-Mitteln den Browser validieren lassen
- Sehr einfach und schnell
- Einfache API, CSS-Selektoren
- Aber:
 - Darstellung sehr verschieden in den Browsern
 - Wenig Flexibilität

Safari:

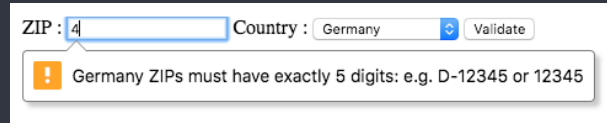


This message has been generated from:

Germany ZIPs must have exactly 5 digits: e.g. D-12345 or 12345

ZIP : 4 Country : Germany Validate

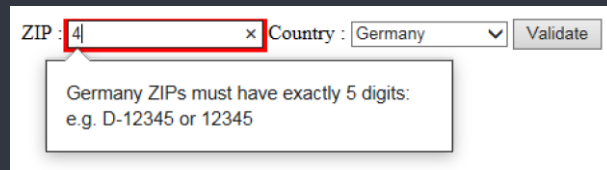
Chrome:



ZIP : 4 Country : Germany Validate

! Germany ZIPs must have exactly 5 digits: e.g. D-12345 or 12345

Internet Explorer:



ZIP : 4 x Country : Germany Validate

Germany ZIPs must have exactly 5 digits:
e.g. D-12345 or 12345

VeeValidate

Template-basiert: die Validierungsregeln werden direkt im Template definiert

```
<input v-validate="'required|max:128'" v-model="greeting" type="text"></input>
```

Directive

Eigenschaften:

- Definition von Control und Validierung an einer Stelle
- Inferred Rules: VeeValidate interpretiert HTML-Constraints
- Viele Validierungsregeln
- Error-Messages integriert, ebenso Localization
- Eigene Regeln möglich
- Zugriff auf andere Elemente per **ref** (<https://vuejs.org/v2/api/#ref>)

VeeValidate 2.1

VeeValidate hat ab Version 2.1 ein anderes empfohlenes Vorgehen:

```
<ValidationProvider name="greeting" rules="required|max:128">
  |   <input v-model="greeting">
</ValidationProvider>
```

Anstatt einer **Directive** kommt eine Validierungs-Komponente zum Einsatz. Diese gibt per **Scoped-Slots** auch Zugriff auf die Error-Meldungen:

```
<ValidationProvider name="greeting" rules="required|max:128">
  <div slot-scope="{ errors }">
    <input v-model="greeting" />
    <span v-show="errors.length > 0">{{ errors[0] }}</span>
  </div>
</ValidationProvider>
```

Zum Vergleich per Directive:

```
<input v-validate="required|max:128" v-model="greeting" type="text" />
<span v-show="errors.has('greeting')">{{ errors.first("greeting") }}</span>
```

Tipp: Artikel zu den Neuerungen:

<https://medium.com/@logaretm/vee-validate-validation-providers-b5b38647c05c>

VeeValidate

Eigene Validierungsregeln:

```
Validator.extend("minorder", {  
  // Custom validation message  
  getMessage: () => `Minimum order value is 20€`,  
  // Custom validation rule  
  validate: value =>  
    new Promise(resolve => {  
      resolve({  
        valid: value ≥ 20  
      });  
    })  
});
```

Eigene Validierungsregeln sind Funktionen, die den Wert erhalten und entweder ein **Promise** oder das Validierungsergebnis zurückgeben.

Vuelidate

Modell-basiert: die Validierungsregeln werden in der Komponente definiert

```
export default {  
  data() {  
    return {  
      greeting: ''  
    }  
  },  
  validations: {  
    greeting: {  
      required,  
      maxLength: maxLength(128)  
    }  
  }  
}
```

Vuelidate registriert ein globales Mixin

Vuelidate

Eigenschaften:

- Registriert ein globales Mixin
- Validierung des Modells unabhängig von der Darstellung
- Verschachtelte Strukturen und Arrays möglich
- Standard-Validatoren, leicht erweiterbar
- Leichtgewichtig

Vuelidate

Eigene Validierungsregeln:

```
validations: {  
  orderValue: {  
    minOrder(value) {  
      if (!value) return true;  
  
      // async ist möglich per Promise  
      return new Promise(resolve => {  
        setTimeout(() => {  
          resolve(value ≥ 20);  
        }, 350);  
      });  
    }  
  }  
}
```

VeeValidate vs. Vuelidate

VeeValidate:

- komfortabler für einfache Formulare
- Mehr integrierte Validatoren
- Kann auch für Formulare ohne Modell-Binding eingesetzt werden
- Kürzlich erschienene API-Erweiterung (*Validator.verify*) ermöglicht programmatische Validierung

Vuelidate:

- deutlich flexibler
- Schlanker
- Für komplexe Formulare wesentlich besser geeignet
- Aber: seit vielen Monaten keine Aktivität im Projekt

Wann validieren?

Zustand eines Elements:

- **touched / untouched**: wurde das Feld bereits fokussiert?
- **pristine / dirty**: wurde der Wert verändert?
- **valid / invalid**

Beim Tippen: direkte Rückmeldung, aber Fehlermeldung schon während der Eingabe

Beim Verlassen eines Eingabefelds: Fehlermeldung direkt nach der Eingabe, aber der Fokus ist dann beim nächsten Feld

Beim Seitenwechsel: sinnvoll in Ergänzung zur direkten Validierung am Eingabefeld

Beim Abschicken: finale Validierung im Client

Fehler anzeigen

Validierungsfehler sollen im richtigen Moment an der richtigen Stelle angezeigt werden, um den Nutzer optimal zu unterstützen.

Möglichkeiten:

- Am Feld
- Zusammenfassung an zentraler Stelle (z.B. oberhalb)

Die fehlerhaften Eingaben sollten optisch hervorgehoben werden bis die Eingabe korrigiert ist. Aus einer Fehlerliste oberhalb des Formulars sollte das betroffene Feld direkt anspringbar sein.

Fehler anzeigen

Fehlermeldung am Feld:

Konto erstellen

Ihr Name

E-Mail

! Geben Sie Ihre E-Mail-Adresse ein

Passwort

! Geben Sie Ihr Passwort ein.

Passwort nochmals eingeben

Fehlerübersicht oberhalb:

Beachten Sie Folgendes:

- Bitte geben Sie einen Vornamen ein.
- Bitte geben Sie einen Familiennamen ein.
- Bitte tragen Sie Ihr Geburtsdatum (tt.mm.jjjj) ein.
- Bitte geben Sie Straße und Hausnummer ein.
- Bitte geben Sie eine Postleitzahl ein.
- Bitte geben Sie einen Ortsnamen ein.
- Bitte tragen Sie eine gültige Telefonnummer ein.

Ja, ich möchte Mitglied werden.

☒ Sobald wie möglich

Ihre persönlichen Daten

Anrede

Herr

Titel

Vorname

Nachname

Mehrseitige Formulare validieren

Mehrseitige Formulare bringen einige zusätzliche Herausforderungen mit sich, insbesondere in Bezug auf die Validierung.

Es gibt zwei grundsätzliche Herangehensweisen:

1. Ein Wechsel zur nächsten Seite ist nur möglich, wenn alle Eingaben erfolgt sind.
2. Anwender können sich frei auf allen Seiten bewegen.

Variante 1 verhindert, dass eine übergreifende Validierung erforderlich wäre, schränkt aber die Flexibilität der Anwender ein.

Variante 2 erfordert ein übergreifendes Validieren.

Mehrseitige Formulare validieren

Umsetzung per Vuelidate:

- Die Validierungsregeln werden nicht in den einzelnen Komponenten gepflegt, sondern als **Mixin** bereitgestellt
- Jede Komponente importiert die benötigten Regeln per Mixin
- Für eine mehrseitige Validierung werden die Validierungs-Mixins aller betroffenen Seiten importiert. Achtung: gleichzeitig muss Zugriff auf das Modell des gesamten Formulars bestehen. Am elegantesten wird dies per **Vuex** gelöst (gleich mehr dazu).

==> DEMO

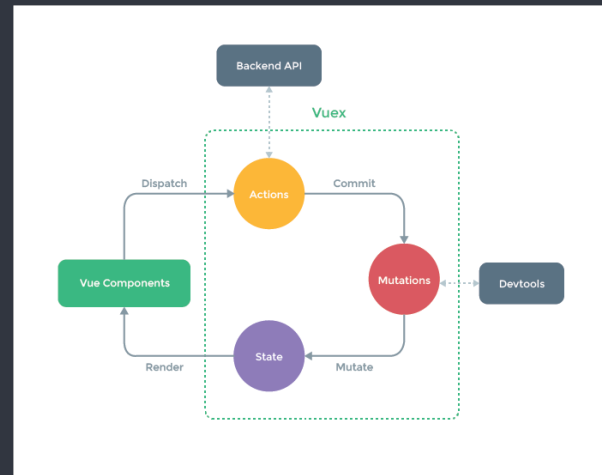
Vuex

State Management

Vuex

Flux für Vue.js

- Vuex ist die offizielle State Management Library for Vue.js
- Wird als Plugin installiert
- Entwickelt und gewartet vom Vue-Core-Team
- Vuex basiert auf dem Reactivity System von Vue.js
- Single State Tree und optionale Module



Vuex & v-model

Das Two-Way-Data-Binding kann nicht direkt an Properties aus dem Vuex-Store gebunden werden, da dies das Flux-Konzept verletzen würde!

Möglichkeiten:

1. Auf v-model verzichten
2. Computed Prop mit Getter & Setter
3. Lokaler State

Bonus: Einsatz von **vuex-pathify** (<https://davestewart.github.io/vuex-pathify/#/>)

Vuex

Ohne v-model:

Manuelles Value-Binding und Aktualisierung bei Änderungen per Methode:

```
<input :value="greeting" @input="updateGreeting" />
```

```
computed: {  
  ...mapState({  
    greeting: state => state.greeting  
  })  
},  
methods: {  
  updateGreeting(e) {  
    this.$store.commit("updateGreeting", e.target.value);  
  }  
}
```

Einschätzung: Umständlich. Es müssen sowohl Template als auch die Komponente angepasst werden.

Vuex

v-model mit Getter/Setter:

Normaler Einsatz v-model:

```
<input v-model="greeting" />
```

```
computed: {  
  greeting1: {  
    get() {  
      return this.$store.state.greeting;  
    },  
    set(value) {  
      this.$store.commit("updateGreeting", value);  
    }  
  }  
}
```

Einschätzung: das Template muss nicht angepasst werden, was von Vorteil ist. Bei vielen Properties entsteht allerdings viel Code.

Vuex

Lokaler State:

Erzeugen einer lokalen Kopie:

```
data: () => ({
  localGreeting: ""
}),
beforeMount() {
  // deep clone
  this.localGreeting = JSON.parse(JSON.stringify(this.greeting));
},
computed: {
  ...mapState({
    greeting: state => state.greeting
  })
}
```

Einschätzung: für einzelne Werte wenig sinnvoll, aber für den Zustand eines gesamten Formulars einsetzbar. Achtung: nicht nutzbar, wenn sich der zentrale Zustand asynchron verändern kann!

Vuex

Vuex-Pathify:

Vuex-Pathify vereinfacht über Hilfsmethoden und Konventionen den Umgang mit Vuex. Alternativ sind natürlich auch eigene Hilfsmethoden denkbar.

```
import { sync } from "vuex-pathify";

export default {
  computed: {
    greeting: sync("greeting")
    /* shortcut für
    greeting: {
      get () {
        return this.$store.state.greeting
      },
      set (value) {
        return this.$store.commit('SET_GREETING', value)
      },
    }
    */
  }
};
```

Einschätzung: lohnenswert einmal auszuprobieren oder als Inspiration für eigene Lösungen.
Artikel dazu: <https://alligator.io/vuejs/vuex-pathify/>

Validierung auslagern

Grundidee:

- Die Validierung in ein Vuex-Modul auslagern
- Innerhalb des Vuex-Moduls wird eine eigenständige Vue-Instanz für die Validierung genutzt
- Die Vue-Instanz nutzt Vuelidate-Regeln für die Validierung
- Die Validierungsergebnisse werden im Store abgelegt
- Dieses Vorgehen eignet sich insbesondere für dynamische Formular-Generierung aus einem Schema

Die Sample-App enthält ein lauffähiges Beispiel

Eingabe-Controls

Das Formular gestalten

Eingabe-Controls

Drei grundsätzliche Möglichkeiten:

1. HTML-Controls direkt nutzen
2. Eine UI-Library einsetzen
3. Eigene Controls entwickeln

Gründe für eigene Controls:

Das Komponentensystem von Vue.js ist wunderbar dafür geeignet, eigene Logik in Komponenten zu kapseln. Eigene Controls können auch auf UI-Libraries basieren und zusätzliche Features ergänzen. Der Einsatz von *Functional Components* sollte in Betracht gezogen werden (besonders leichtgewichtig da ohne *State*).

UI-Libraries einsetzen

Die bekanntesten UI-Libraries für Vue.js:

- **Vuetify** <https://vuetifyjs.com/>
- **Quasar** <https://quasar-framework.org/>
- **Element** <https://element.eleme.io/>
- **Bootstrap Vue** <https://bootstrap-vue.js.org/>
- **Buefy** <https://buefy.github.io/>

Reine CSS-Libraries lassen sich auch leicht selbst integrieren (Bulma, Pure.CSS, Kickstart...).

Dynamik

Dynamik rund um Eingabe-Controls

Sichtbarkeit:

Dynamische Sichtbarkeitsbedingungen lassen sich mit Vue.js mittels *v-if* / *v-show* leicht umsetzen.

Editierbarkeit:

Das `readonly`-Attribut für Eingabefelder kann für eine dynamische Steuerung der Editierbarkeit verwendet werden.

Validierung:

Dynamische Validierungsregeln lassen sich sowohl über `VeeValidate` als auch `Vuelidate` realisieren.

Dynamik

Berechnete Werte

Auch bei berechneten Werten (Summen,) können die Funktionen von Vue.js direkt genutzt werden (**Computed Properties**, **Watcher**, **Vuex-Getter**). Vue.js sorgt mit seinem Reactivity-System dafür, dass alle Werte stets aktuell sind.

Aber **Achtung**: da die Werte im Browser manipulierbar sind, sollte immer das Backend die letztendliche Hoheit haben!

Eigene UI-Controls

Ein Control mit v-model-Support lässt sich leicht erstellen:

```
<template>
  <input
    v-bind:value="value"
    v-on:input="$emit('input', $event.target.value)"
  />
</template>

<script>
export default {
  props: ["value"]
};
</script>
```

v-model-Support:

- Empfang des Werts per *value*-prop (Lesezugriff auf Modell)
- Änderungen kommunizieren per *input*-Event (Schreibzugriff auf Modell)

Beispiel 1: Read / Edit

Ein Control zur Unterscheidung von Read- und Edit-Mode

```
<template>
  <div>
    <input
      :type="isFocussed ? 'number' : 'text'"
      v-model="displayValue"
      ref="input"
      @blur="isFocussed = false;"
      @focus="isFocussed = true;"
    />
  </div>
</template>
```

Wenn nicht fokussiert, wird ein formatierter Währungsstring angezeigt. Bei Fokus wechselt der Eingabetyp auf „number“ mit dem numerischen Wert. An die Parent-Komponente wird der numerische Wert übermittelt.

```
<script>
export default {
  name: "currency-input",
  data() {
    return {
      isFocussed: false
    };
  },
  props: ["value"],
  computed: {
    displayValue: {
      get: function() {
        if (this.isFocussed) {
          // Formattierung entfernen
          return this.value.toString();
        } else {
          // Formattierten Wert zurückgeben
          return new Intl.NumberFormat("de-DE", {
            style: "currency",
            currency: "EUR"
          }).format(this.value);
        }
      },
      set: function(modifiedValue) {
        // emitte den neuen Wert
        this.$emit("input", modifiedValue);
      }
    }
  }
};
</script>
```


Beispiel 2: In-Place-Edit

Bearbeitungsmodus per Doppelklick:

```
<template>
  <div>
    <input
      v-show="isFocussed"
      :type="isFocussed ? 'number' : 'text'"
      v-model="displayValue"
      ref="input"
      @blur="isFocussed = false"
      @focus="isFocussed = true"
    />
    <div v-show="!isFocussed" @dblclick="setEdit()">{{ displayValue }}</div>
  </div>
</template>
```

Die Komponente zeigt ein einfaches Div-Element mit dem Anzeigewert. Ein Doppelklick darauf, wechselt in den Bearbeitungsmodus und zeigt ein Eingabefeld anstelle des Div-Elements.

```
<script>
export default {
  name: "currency-input",
  data() {
    return {
      isFocussed: false
    };
  },
  props: ["value"],
  computed: {
    displayValue: {
      get: function() {
        if (this.isFocussed) {
          // Formattierung entfernen
          return this.value.toString();
        } else {
          // Formattierten Wert zurückgeben
          return new Intl.NumberFormat("de-DE", {
            style: "currency",
            currency: "EUR"
          }).format(this.value);
        }
      },
      set: function(modifiedValue) {
        // emitte den neuen Wert
        this.$emit("input", modifiedValue);
      }
    }
  },
  methods: {
    setEdit() {
      this.isFocussed = true;
      this.$nextTick(() => this.$refs.input.focus());
    }
  }
};
</script>
```

Beispiel 3: Masked Input

Wert während der Eingabe formatieren:

```
<template>
  <div>
    <input type="text" v-model="displayValue" @keypress="isNumber($event)" />
  </div>
</template>
```

Die Komponente lässt nur die Eingabe von Ziffern zu und formatiert diese in 4er-Gruppen. Eine Eingabe ruft die **set**-Methode auf und der unformatierte Wert wird an die Parent-Component gegeben. Das Template ruft die **get**-Methode auf und erhält den formatierten Anzeigewert.

```
<script>
export default {
  name: "grouped-number-input",
  props: ["value"],
  computed: {
    displayValue: {
      get: function() {
        if (!this.value) return;
        // zunächst alles außer Ziffern entfernen
        let value = this.value.toString().replace(/^[^d]/g, "");

        //Gruppieren
        value = value
          .toString()
          .match(/.{1,4}/g)
          .join(" - ");

        return value;
      },
      set: function(modifiedValue) {
        //alles außer Ziffern entfernen
        let value = modifiedValue.toString().replace(/^[^d]/g, "");

        // emitte den neuen Wert
        this.$emit("input", value);
      }
    }
  },
  methods: {
    isNumber: function(evt) {
      var charCode = evt.which ? evt.which : evt.keyCode;
      if (charCode > 31 && (charCode < 48 || charCode > 57))
        evt.preventDefault();
      return true;
    }
  }
};
</script>
```

Form Generators

Formulare dynamisch generieren

Formulare dynamisch generieren

Warum?

Es gibt Anwendungsfälle (z.B. generische Produkte, umfangreiche Anwendungen mit gleichartigen Formularen), für die es von Vorteil ist, wenn Formulare nicht einzeln ausprogrammiert, sondern aus einem Schema generiert werden. Auch hierfür ist Vue.js gut gerüstet.

Mehr Informationen:

- Vue-Form-Generator: <https://github.com/vue-generators/vue-form-generator>
- Do it with Elegance: How to Create Data-Driven User Interfaces in Vue:
<https://blog.rangle.io/how-to-create-data-driven-user-interfaces-in-vue/>
- Die Sample-App zu diesem Talk enthält ein lauffähiges einfaches Beispiel

Usability & User Experience

Tipps & Ressourcen

Usability & User Experience

Auf dem Weg zu großartigen Formularanwendungen

Eine intuitive und stolperfreie Bedienung ist die Basis eines guten Formulars. Mit einem UI-Framework wie Vue.js sind die technischen Voraussetzungen gegeben, die Anwender bestmöglich zu unterstützen.

UI-Libraries enthalten viele Best-Practices von guten Formular-Controls und können entweder direkt verwendet oder als Ideengeber für eigene Lösungen genutzt werden.

Ein paar Tipps:

- Den Anwendungsfall immer im Blick behalten
- Sinnvolle Vorbelegung von Feldern
- Die Inhalte semantisch gruppieren
- Inhalte aneinander ausrichten – nicht zu viele Fluchten

Usability & User Experience

Tipps (continued):

- Nicht an Papiervorlagen orientieren
- Genügend Abstand und Weißraum vorsehen
- Barrierefreiheit beachten
- Sinnvolle Tab-Reihenfolge
- Für verschiedene Auflösungen optimieren
- Auf die Besonderheiten verschiedener Eingabegeräte eingehen (z.B. kein Hover auf Touch-Devices)
- Eingabehilfen integrieren (Hilfetexte am Element, modale Dialoge)
- Die richtigen Controls auswählen (Range-Slider vs. Input, Checkbox vs. Toggle..)
- Ausprobieren und Feedback einholen!

Usability & User Experience

Tipps (continued):

Eine sehr gute Quelle für Informationen und Inspirationen sind die öffentlich zugänglichen **Design Systems** von Unternehmen.

*A **design system** is a collection of reusable components, guided by clear standards, that can be assembled together to build any number of applications.*

<https://www.invisionapp.com/inside-design/guide-to-design-systems/>

In der Sample-App sind diverse Links auf Artikel und Design Systems enthalten.

Tipps & Ressourcen

Folien und Sample-App zu diesem Talk:

<https://github.com/NoFrank/vue-complex-forms>

Die App enthält beispielhafte Implementierungen für viele angesprochenen Themen und ist eine gute Ausgangsbasis für eigene Versuche.

Vielen Dank – Das war's!

@norbertdfrank