VUE

Intro

Wie schon mehrfach bemerkt, ist es sehr umständlich ein dynamisches Frontend von Grund auf per Hand in HTML und JavaScript umzusetzen. Abhilfe schaffen z.B. Frameworks wie Vue (ausgesprochen wie das englische „View", manch einer nennt es auch eingedeutscht „Wü"). Diese kümmern sich um die Dynamisierung einer Ansicht anhand von Daten.

Ein ersten Projekt war die Reisekostenabrechnung im neuem CRM. Wer das noch nicht kennt, darf gern in den Feldern rumschreiben und die bunten Knöpfchen drücken (keine Angst, es wird nichts gespeichert oder gelöscht). Es wurde dabei nur der Frontend-Code angefasst (das Formular sendet immer noch an dasselbe Action) und in knapp einer Stunde war das Thema erledigt.

Dabei ist sogar eine neue Funktion (Live-Anzeige der Summe der Posten) rausgesprungen. Zu Demozwecken habe ich unten drunter noch die aktuellen Daten ausgegeben, die sich bei Änderungen live aktualisieren!

Es werden hier nicht alle Begriffe erklärt, sondern nur das, was für das Beispiel nötig ist. Wer mag kann gern Google bemühen oder den weiterführenden Links folgen. Manches ist vielleicht nicht 100% Best Practice, das würde aber sonst den Rahmen sprengen.

Installation

Vue ist fix installiert. Man muss lediglich eine .js Datei via `<script>` einbinden. Während der Entwicklung empfiehlt sich die *Development Version*, da man hier hilfreiche Warnungen und genaue Fehler erhält, sollte mal etwas schief gehen. Die Dateigröße hält sich in Grenzen:

- Development Version (also unminified): **283 KB** (ca. wie jQuery UI minified)
- Production Version (minified): **85 KB** (wie jQuery minified) [**31 KB**, wenn sie vom Server noch mit GZIP komprimiert übertragen wird]

Wer einfach mal etwas mit Vue testen will, kann z.B. hier einfach live rumprobieren:

http://jsfiddle.net/filipelinhares/2eAqE/ (nach Änderungen oben auf Run drücken)

Grundlegendes

Oberste Priorität bei der Entwicklung mit Vue (und auch anderen MVC/MVVM/etc.-Frameworks) hat die **Trennung von Daten (Model) und Anzeige (View)**.

Das bedeutet, dass alle Daten, die für den Aufbau der Anzeige nötig sind, im JavaScript als Objekte vorhanden sind, bzw. daraus berechnet werden können. In erweiterten Szenarien können Daten auch asynchron nachgeladen werden.

Ist das erledigt, definiert man die Anzeige mittels einem Template. Das erfolgt größtenteils mit normalen HTML-Tags. Hier und da können spezielle Platzhalter verwendet werden, ähnlich unseren Batix-Tags. Diese unterscheiden sich in Interpolationen und Direktiven wie z.B. Schleifen (genaue Infos dazu weiter unten).

Das Besondere an Vue ist nun, dass es die Daten nimmt und anhand derer den View zusammenbaut. Es wird also kein HTML-Code beim Server angefragt. So weit, so schnell selbst gebaut. Jetzt kommt aber der Clou! Werden nun die Daten geändert (das kann u.a. das Ändern eines Strings oder das Hinzufügen/Entfernen von Array-Elementen sein) bekommt Vue das **direkt** mit und passt die Anzeige entsprechend an. Es rendert also den View neu. Hierbei ist Vue auch auf Performance bedacht und rendert nur die nötigen Teile neu, die sich geändert haben. Wow! Man kann das im CRM-Beispiel ganz gut sehen, wenn man irgendwo tippt und es aktualisiert sich die Debug-Ausgabe oder die Summe.

Das ist aber noch nicht alles, denn Vue kann auch vom User gemachte Eingaben in Textfelder oder via Checkboxen etc. **wieder zurück in die Daten schreiben**. Vue benötigt dafür kein jQuery oder sonstige Libraries.

Model (Daten)

Im Beispiel habe ich die User-Daten der Einfachheit halber größtenteils mit Batix-Tags gefüllt. Das Objekt sieht so aus:

```
var fzkData = {
 date: moment().format("DD.MM.YYYY"),
  lastMonth: moment().subtract(1, 'month').format("MM.YYYY"),
  settings: {
    kennzeichen: "<bx:recordfield.FZK_Kennzeichen encode="javascript" />",
    iban: "<bx:recordfield.FZK_IBAN encode="javascript" />",
    bic: "<bx:recordfield.FZK_BIC encode="javascript" />",
    kontoinhaber: "<bx:recordfield.FZK_Kontoinhaber encode="javascript" />"
  },
```

```
  rows: []
};
```

Dabei ist `lastMonth` eine Hilfsvariable, `settings` und `date` entsprechen den Feldern im linken Bereich und `rows` sind die Tabellenzeilen im rechten Bereich.

Die Daten werden beim Erzeugen einer Vue-Instanz übergeben und sind dann innerhalb der Instanz sowie im Template verfügbar.

View (Anzeige)

Das komplette Template habe ich ein div mit der ID `fzkApp` gepackt. Ich zeige hier ein paar Ausschnitte davon. Die ganzen Bootstrap Klassen und das Beiwerk lasse ich dabei zwecks Lesbarkeit weg.

Die linke Seite sieht im Quelltext so aus

```
<h5>Daten</h5>
<div>
  <label for="Kennzeichen">Kennzeichen</label>
  <input type="text" name="Kennzeichen" v-model="settings.kennzeichen">
</div>
<div>
  <label for="IBAN">IBAN</label>
  <input type="text" name="IBAN" v-model="settings.iban">
</div>
<div>
<label for="BIC">BIC</label>
  <input type="text" name="BIC" v-model="settings.bic">
</div>
<div>
  <label for="Kontoinhaber">Kontoinhaber</label>
  <input type="text" name="Kontoinhaber" v-model="settings.kontoinhaber">
</div>
<div>
  <label for="Datum">Datum</label>
  <input type="text" name="Datum" v-model="date">
</div>
```

Der einzige Unterschied zu normalem HTML ist die v-model Direktive. Damit man diese von normalen Attributen unterscheiden kann, fangen die Direktiven übrigens mit `v-` (für Vue) an.

Dank `v-model` wird von Vue für dieses Element ein sogenanntes **Two-Way-Binding** zu den Daten erzeugt. Das bedeutet die folgenden Prozesse laufen automatisch ab:

- ändern sich die Daten -> ändert sich das angezeigte Input
- ändert sich das Input (durch User-Interaktion) -> ändern sich die Daten

Man sieht hier schön, dass der Wert von `v-model` dem „Pfad" in den Daten entspricht. Am Ende ist es quasi JavaScript-Code, der die gewünschte Variable holt.

Die rechte Seite sieht so aus:

```
<h5>Posten</h5>

<table>
  <thead>
    <tr>
      <th>Belegdatum</th>
      <th>Ausgabe (z.B. Menge + Kraftstoffart)</th>
      <th>Betrag [€]</th>
      <th></th>
    </tr>
  </thead>
  <tbody>
    <tr v-for="(row, index) in rows">
      <td><input type="text" :name="'ph_' + index + '_datum'" v-model="row.datum"></td>
      <td><input type="text" :name="'ph_' + index + '_ausgabe'" v-model="row.ausgabe"></td>
      <td><input type="text" :name="'ph_' + index + '_betrag'" v-model="row.betrag"></td>
      <td><button @click="removeRow(index)">x</button></td>
    </tr>
  </tbody>
</table>

<span>Summe: {{ sum }} €</span>

<button @click="addRow()">+</button>
<button type="submit">erstellen</button>
```

Hier gibt es einige neue Sachen! Zum einen wäre da v-for, was man sich wie eine for-Schleife z.B. in JavaScript vorstellen kann. Diese Direktive sorgt dafür, dass ein `<tr>` für jedes Element im `rows`-

Array erzeugt wird. Hier hilft es sich die Daten gefüllt vorzustellen, nehmen wir also mal an:

```
"rows": [
  { // = row, index 0
    "datum": "12.10.2018",
    "ausgabe": "23,5 l Diesel",
    "betrag": "33,44"
  },
  { // = row, index 1
    "datum": "20.10.2018",
    "ausgabe": "Waschen",
    "betrag": "4,99"
  }
]
```

Für diesen Fall werden also bei Benutzung von `<tr v-for="(row, index) in rows">` zwei `<tr>`s erzeugt werden. In der Direktive gibt man der Laufvariable einen Name (hier `row`), optional kann man auch noch den aktuellen Schleifendurchlauf bekommen (hier `index`). Innerhalb der Schleife kann auf Daten des aktuellen Array-Elements mittels `row` (der Laufvariable) zugegriffen werden, was man hier mit dem bekannten `v-model` sieht (`v-model="row.datum"`). In diesem Fall kommt `v-model` auch nicht mit den Array-Elementen durcheinander – Änderungen landen beim richtigen Objekt, ohne dass man extra nochmal einen Index o.ä. angeben muss.

Ein weiteres neues Konstrukt ist der Doppelpunkt. Das ist nichts weiter als eine Abkürzung für v-bind. Statt `:name="..."` könnte man also auch `v-bind:name="..."` schreiben. Im Gegensatz zu `v-model` erzeugt `v-bind` nur ein **One-Way-Binding**, die Anzeige wird also nur aktualisiert, falls sich die Daten ändern, nicht andersrum.

Zu beachten ist, dass der Wert von `v-bind` kein reiner Text, sondern JavaScript Code ist! Das ist übrigens bei fast allen Direktiven der Fall. Möchte man also einen dynamischen String zusammenbasteln, müssen z.B. Quotes und Plusse verwendet werden.

Kurz dazu, warum ich den `name` der Inputs so dynamisch zusammenbaue: Im Beispiel wird eine `<form>` an den Server geschickt und dabei müssen die Elemente natürlich sinnvoll gruppiert sein (ähnlich new-1/Feldname beim Save-Action). Man kann den `name` auch weglassen und später die Daten direkt irgendwo hin schicken, diese spiegeln ja den Stand im View wider. Ich habe hier wie gesagt die Backend-Logik nicht angefasst und nur das Frontend analog zum Alten gebaut.

Eine weitere Abkürzung bei `@click="addRow()"` ist das @, nämlich für v-on. Anstatt `@click="..."` könnte man also auch `v-on:click="..."` schreiben. Hiermit lassen sich Events binden und auch hier ist der Wert der Direktive im Prinzip wieder JavaScript-Code (man kann die runden Klammern auch weglassen). Alle Variablen, die im Scope sind, können natürlich verwendet werden – hier z.B. `index`, um zu übergeben, welche Zeile (also welches Array-Element) gelöscht werden soll. Wo die

Methoden `removeRow` und `addRow` herkommen, sehen wir weiter unten.

Als letztes, zentrales Syntax-Element ist die Interpolation, ausgedrückt durch die geschweiften Klammern, zu nennen. Innerhalb dieser, man ahnt es vielleicht schon, wird wieder JavaScript-Code geschrieben. Im einfachsten Fall ist das nur eine Variable, die dann eben ausgegeben wird. Im Beispiel steht da `sum`, was keine Variable, sondern eine „Computed Property". Details dazu gibt es auch weiter unten.

Die Ausgabe der Live-Daten funktioniert ebenfalls über Interpolation und eine Computed Property:

```
Live-Data:<br>
<small><pre>{{ dataAsJson }}</pre></small>
```

Es ist vielleicht aufgefallen, dass hier nicht darauf geachtet werden muss, irgendwo `htmlencode` o.ä. einzubauen. Vue kümmert sich selbst darum, dass die Daten passend escaped werden und so nicht zur Sicherheitslücke werden können (zumindest was XSS angeht).

Vue-Instanz erzeugen

Das Template wird folgendermaßen mit den Daten verbunden und „reaktiv" gemacht:

```
new Vue({
  el: "#fzkApp",
  data: fzkData
})
```

Das reicht schon und man erhält seine fertige Komponente. Natürlich gibt es noch mehr Optionen, ein paar davon brauchen wir auch noch, um z.B. die Methoden und Computed Properties zu definieren.

Methoden

Im Template oben wurden ja die Funktionen `addRow` und `removeRow` aufgerufen. Diese müssen natürlich noch definiert werden. Das geschieht über die Option `methods`. Hier sollten Funktionen rein, die in der Komponente (entweder im Template oder in anderen Methoden) gebraucht werden und nur spezifisch für diese Komponente sind. Ich habe die zwei Methoden folgendermaßen definiert:

```
new Vue({
  // ...
```

```
  methods: {
    addRow: function() {
      this.rows.push({
        datum: "__." + this.lastMonth,
        ausgabe: "",
        betrag: ""
      });
    },
    removeRow: function(index) {
      this.rows.splice(index, 1);
    }
  }
})
```

(splice ist vielleicht nicht so bekannt: es entfernt hier einfach die angegebene Anzahl an Elementen vom übergebenen Index an)

Das `this` in solchen Methoden ist automatisch die Vue-Instanz. Darüber erhält man direkten Zugriff auf z.B. die Daten, Methoden und Computed Properties. Wie man sieht bearbeite ich hier auch nur die Daten, den Rest (das Rendern des entsprechenden HTMLs) übernimmt Vue!

Damit nach dem Laden der Seite direkt eine Zeile mit leeren Inputs zum Eingeben erscheint, wird `addRow` dann noch im `created` Lifecycle Hook aufgerufen, welcher feuert, sobald die Vue-Instanz angelegt wurde:

```
new Vue({
  // ...
  created: function() {
    this.addRow();
  }
})
```

Computed Properties

Computed Properties sind den Methoden ähnlich. Sie werden über die Option `computed` definiert. Der Unterschied zu Methoden ist, dass sie ein Ergebnis zurückliefern. Dieses kann dann im Template ausgegeben werden. Auch hier ist Vue so schlau den View zu aktualisieren, wenn sich das Ergebnis ändert. Das kann ja der Fall sein, da man in den Computed Properties irgendetwas anhand der Daten der Instanz berechnet.

Im Template oben wurden zwei Stück benutzt: `dataAsJson` gibt einfach die aktuellen Daten als JSON formatiert aus und `sum` berechnet die Summe der Tabellenzeilen. Das ist in wenigen Zeilen niedergeschrieben:

```
new Vue({
  // ...
  computed: {
    dataAsJson: function() {
      return JSON.stringify(this.$data, null, 2);
    },
    sum: function() {
      var s = 0;
      this.rows.forEach(function(row) {
        var amount = row.betrag ? parseFloat(row.betrag.replace(/,/, ".")) : 0;
        s += amount;
      });
      return s.toFixed(2).replace(/\./, ",");
    }
  }
});
```

Man sieht hier, dass auf `this.$data` zugegriffen wird. Das ist eine von diversen Spezial-Properties der erzeugten Vue-Instanz (die ja als `this` verfügbar ist). Ich hätte genauso gut auch `fzkData` nehmen können, da wir diese Daten im Beispiel als globales Objekt haben. Das ist aber nicht schön. Im Normalfall sollte man die Daten inline beim Instanz Erzeugen angeben, damit diese nicht aus Versehen noch von einer anderen Instanz mitbenutzt werden. Natürlich kann man auch Instanzen/Komponenten verschachteln und Daten rumreichen, das würde aber hier wieder den Rahmen sprengen. Es gibt mit Vuex auch eine State Management Library für komplexere Anwendungen, das aber nur am Rande.

Weiterführende Infos und Features

Vue bietet noch viele Features, die hier nur ganz kurz oder gar nicht erwähnt wurden. Ein paar picke ich hier extra nochmal raus. Als Dokumentation gibt ansonsten z.B. den Guide (beschreibt die Features und erklärt, wie man etwas macht) und die API Referenz (welche alle möglichen Optionen, Properties, etc. auflistet).

v-if / v-show

Mit `v-if` oder `v-show` kann man dynamisch Blöcke verstecken oder anzeigen, ähnlich unserem `bx:if`. So können z.B. Abschnitte in Formularen zugeschalten werden, wenn eine bestimmte Checkbox angehakt wird. Zu den Unterschieden von if und show siehe die Doku.

Beispiel (`ok` wäre dann z.B. ein Boolean in den Daten):

```
<h1 v-if="ok">Yes</h1>
<h1 v-show="ok">Hello!</h1>
```

v-on mit Modifiern

Beim Event-Binden kann man noch bestimmte Modifier setzen. Das reduziert Boilerplate-Code, wenn man z.B. nur an Rechtsklicks oder Enter-Tasten interessiert ist und/oder man das Event preventen will. Ein paar Beispiele aus der Doku:

```
<!-- prevent default -->
<button @click.prevent="doThis"></button>
<!-- prevent default without expression -->
<form @submit.prevent></form>
<!-- key modifier using keyAlias -->
<input @keyup.enter="onEnter">
<!-- the click event will be triggered at most once -->
<button @click.once="doThis"></button>
```

Filter

Ausgaben im Template oder mit v-bind können mittels Filtern angepasst werden. Denkbar ist z.B. eine schicke Preisformatierung mit zwei Nachkommastellen oder das Großschreiben des ersten Buchstabens einer Meldung:

```
{{ message | capitalize }}
```

Components

Man kann nachnutzbare Komponenten definieren. Ein einfacher Anwendungsfall ist z.B. ein Listenelement. Hat man z.B. eine Kontaktliste, so kann man für den einzelnen Kontakt-Eintrag eine extra Komponente definieren. Das hilft beim Organisieren von größeren Projekten und ist vergleichbar mit Textbausteinen. Jede Komponente bekommt dabei ihre eigene Vue-Instanz, das sieht man z.B. hier in einem Beispiel.

Apropos Components: Es gibt sehr viele vorgefertigte Komponenten für Vue! Das reicht von Datepickern und Kalendern bis hin zu Bootstrap-Komponenten.

Weitere Links:

https://vuejs.org/ - offizielle Website

https://entwickler.de/online/javascript/vue-js-javascript-framwork-einfuehrung-579850224.html - Einführung auf Deutsch