Formulare 2.0

[Stand: 13.05.2020 | Programmversion: 12.00.000]

Konzept der Formulare 2.0

Inhalt

[**1** **Wie werden Formulardefinitionen gespeichert?** 3](#_Toc46463144)

[1.1 Konfiguration Plugin 3](#_Toc46463145)

[1.1.1 Konfigurationen in elo-forms: 3](#_Toc46463146)

[1.1.2 Übertragung der Daten per Rest 5](#_Toc46463147)

[1.1.3 Umsetzung 5](#_Toc46463148)

[1.2 Formular Editor 6](#_Toc46463149)

[1.2.1 Vorgehen 6](#_Toc46463150)

[1.2.2 Umsetzung 7](#_Toc46463151)

[2 Feldtypen 8](#_Toc46463152)

[3 Stichwortlisten 9](#_Toc46463153)

[3.1 Statische Stichwortlisten 9](#_Toc46463154)

[3.2 Dynamische Stichwortlisten 9](#_Toc46463155)

[3.3 Benutzerlisten 9](#_Toc46463156)

[3.4 Lokalisiertte Listen/Status Listen 9](#_Toc46463157)

[3.5 Maskenauswahl 9](#_Toc46463158)

[4 Formular-Scripting 10](#_Toc46463159)

[4.1 Allgemeines 10](#_Toc46463160)

[4.1.1 TODO Mehrere Felder für einen Wert 10](#_Toc46463161)

[4.2 Validators 10](#_Toc46463162)

[4.3 Formeln 11](#_Toc46463163)

[4.3.1 Zyklische Formeln 12](#_Toc46463164)

[4.4 Komponenten Events 12](#_Toc46463165)

[4.5 API 13](#_Toc46463166)

Das Dokument ist veraltet, im Git Projekt entsteht eine neue Doku:

https://git.elo.dev/client/mobile/forms/-/tree/master/docs

1. **Wie werden Formulardefinitionen gespeichert?**
   1. Konfigurationsplugin

Überlegungen und Durchführung, wie Formulare in einzelne Objekte aufgeteilt und mittels des Konfigurationsplugins gespeichert werden können.

Das Plugin speichert Werte in folgender Struktur:

{

    "info": {

      "solution": "visitor", <-- Name der Lösung (wird für Portierung benötigt)

      "component": "flows", <-- Allgemeiner Komponentenname "forms"

      "key": "new.value", <-- Schlüssel zum Auffinden des Wertes

      "level": 100 <-- Level der Anpassung

    },

    "value": "foo", <-- Zu speichernder Wert/Objekt

    "protectedProperty": false <-- true: kann nur als Admin ausgelesen werden

}

* + 1. Konfigurationen in elo-forms:

{

    "info": {

      "solution": "<xxx>", <-- Kommt über den Editor

      "component": "forms",

      "key": "new.value", <-- Siehe unten

      "level": 100 <-- Kann im Editor gesetzt werden?

    },

    "value": "foo", <-- Zu speichernder Wert/Objekt

    "protectedProperty": false <-- true: kann nur als Admin ausgelesen werden

}

Wichtig ist der key, da hierüber jedes Objekt identifiziert und effizient aufgefunden werden muss. Der Key ist Punkt-separiert und kann in der Schnittstelle über eine begins-with Logik ausgelesen werden. Deshalb wird der Key in folgende Prefixgruppen unterteilt.

Formulargruppe

Eine Formulargruppe fasst eine Gruppe von Formularen zusammen.

formGroup.<formGroup>

Eine Formgruppe definiert eine Liste von abhängigen Templates.

*Für ELO Objekte ist eine Maske eine Formulargruppe. Diese kann verschiedene Formulare haben, die dann die Datensätze zu dieser Maske auf verschiedene Art visualisieren. Für Masken wird kein extra „formGroup.“ Eintrag geschrieben.*

* formGroup: Bezeichnet hier den Namen der Formulargruppe (alternativ zum *Namen der Maske).*

Unklar: Als Wert wird ein Objekt gespeichert. Eventuell enthält dieses eine Liste von Templategruppen, <templateGroup>.

KONFLIKT: Eine formGruppe hat einen Namenskonflikt mit Masken, da beide dann den Key forms.<formGruppe|Maske>. schreiben.

Templategruppe

Eine Templategruppe fasst eine Gruppe von Templates (Ansichten) zusammen.

templateGroup.<templateGroup>

*Für ELO Objekte ist ein Aspekt eine Templategruppe. Diese kann verschiedene Templates haben, die dann die Felder zu diesem Aspekt auf verschiedene Art visualisieren. Für einen Aspekt wird kein „templateGroup.“ Eintrag geschrieben.*

* templateGroup: Bezeichnet hier den Namen der Templategruppe (alternativ zu *Namen des Aspekts).*

Aktuell wird nur der Eintrag an sich benötigt, um zwischen einer Gruppe von einem Aspekt und einer frei angelegten Gruppe unterscheiden zu können.

Die Daten sollten sich auf die übertragenen Daten aus der Metadatenanfrage zu Aspektzuordnungen beziehen.

{

id: <templateGroup>,

labelTranslationKey: <Name oder Übersetzungsvariable>,

hasMany: ?

}

Diese Gruppen dienen zur Einschränkung der Auswahl im Formular-Editor, damit nicht beliebige Templates zur Auswahl an Formulare kommen, die nichts mit dem eigentlichen Datenobjekt zu tun haben.

KONFLIKT: Eine templateGruppe hat einen Namenskonflikt mit einem Aspekt, da beide den key templates.<templateGruppe|Aspekt>. schreiben.

Formulare/Maskenansichten

forms.

Bezeichnet ein Formular (Zusammenschluss von Templates), das so ausgeliefert werden kann (ein solcher Schlüssel hängt an Aufgaben, Masken zur Verschlagwortung, Flow-Konfigurationen etc.).

Jedes Formular gehört zu einer Formgruppe, die dann entsprechend eine Menge anwendbarer Templates definieren:

forms.<formGroup>.<formId>

* formGroup: Bezeichnet hier den Namen der Formgruppe, auf die sich dieses Formular bezieht. Formulare der gleichen Gruppe beziehen sich auf das gleiche Datenobjekt (nur mit anderem Schwerpunkt, Ansichten, Editiermöglichkeiten). Typischerweise ist so eine Gruppe eine Maske. Zu einer Maske kann es mehrere Formulare geben.
* formId: Bezeichnet die Instanz eines Formulars. Der Name kann frei vergeben werden.

Templates/Aspektansichten

templates.

Bezeichnet ein Template (Teil eines Formulars). Typischerweise stellt ein Template die Visualisierung einer Gruppe von Feldern im Formular (Tabs, einklappbare Gruppen etc.) dar.

*In ELO wäre ein Template die Visualisierung eines Aspekts.*

Jedes Template gehört zu einer Templategruppe, die dann die Menge der zu verwendenden Felder definiert.

templates.<templateGroup>.<templateId>

* templateGroup: Bezeichnet hier den Namen der Templategruppe oder des Aspekts, auf die sich dieses Template bezieht.
* templateId: Bezeichnet die Instanz eines Templates. Der Name kann frei vergeben werden.

Felder

fields.

Speichert die Konfiguration eines einzelnen Feldes. Felder werden einer Templategruppe zugeordnet.

*Bei ELO Einträgen wäre ein Feld einem Aspekt zugeordnet, zu dem es gehört.*

fields.<templateGroup>.<fieldId>

* templateGroup: Bezeichnet hier den Namen der Templategruppe, auf die sich dieses Feld bezieht.
* fieldId: Bezeichnet die Instanz eines Feldes. Der Name kann frei vergeben werden.

Anfragen über keys

Bei der Erstellung eines Formulars, muss dieses zuerst einer Formgruppe (*Maske*) zugewiesen werden. Damit lassen sich dann die zugehörigen Templategruppen ermitteln.

formGroup.<formGroup>

Zu den Templategruppen können alle dazugehörenden Templates abgefragt werden.

templates.<templateGroup>.\*

Im Formular Editor kann so ein Layout für die Formular Instanz zusammengebaut werden, in dem alle vorhandenen Templates genutzt werden können.

Im Template Editor wird für eine neues Template zuerst angegeben, zu welcher Templategruppe es gehört. Damit können dann die Felder abgerufen werden, die für diese Templategruppe bereits erstellt wurden.

fields.<templateGroup>.\*

Neue Felder werden direkt dieser Templategruppe zugeordnet.

* + 1. Übertragung der Daten per Rest

Hier stell sich vor allem die Frage, wie eine Formulardefinition übertragen wird.

* Wird immer ein gesamter Blob (Formular, alle Templates und Felder) übertragen und gespeichert oder speichert der Editor einzelne Teile davon, die dann zu einem Formular nach und nach zusammenwachsen?
* TODO: Hier sind zuerst die Konzepte des Formular Editors gefragt.
  + 1. Umsetzung

Zur Umsetzung wären folgende Schritte zu machen:

* EFRM-14: [vue] Editor Formulare für Felder werden an die Felddefinitionen gehängt und über die registriert.
* EFRM-15: [vue] FeldEditor: Die Komponente bekommt eine Feldkonfiguration (möglicherweise leer) und Felddefinition. Sie zeigt ein Formular, mit dem die Feldkonfiguration erstellt/geändert werden kann.
* EFRM-16: [backend] Möglichkeit bereitstellen, um Formular-/Template-/Felddefinitionen speichern zu können.
  1. Formular Editor

Wie auch immer ein Konzept für einen Editor ausfallen wird, er wird auf Datenebene bestimmte Dinge benötigen.

Generell wird im Formular Editor immer in Bezug auf eine bestimmte **Lösung** (**Solution**) gearbeitet. Diese wird per URL übergeben und an alle Datenanfragen gehängt.

Global wird dann der Level der Anpassung bestimmt. Auch dieser steuert das Lesen/Schreiben aller Formular Daten.

* + 1. Vorgehen

Betrachtet man die Erstellung eines Formulars, muss zuerst (neben den oben erwähnten Solution- und Level-Einstellungen) eine Formulargruppe gewählt werden, in der man das Formular erstellen möchte.

1. Liste aller Formulargruppen (formGroup) mit Möglichkeit eine neue Formulargruppe zu erstellen.

Wird eine Gruppe gewählt, kann dazu alle verfügbaren bestehenden Formulare aufgelistet werden.

1. Liste aller Formular-Instanzen (formId) mit der Möglichkeit, ein bestehendes Formular zu bearbeiten oder ein neues anzulegen.

Wird ein neues Formular angelegt oder ein bestehendes Formular bearbeitet, wird die Ebene gewechselt und der Layout Editor erscheint.

1. Layout Editor bietet die Möglichkeit, verschiedene Komponenten durch ein Layout zu positionieren.

Der Layout Editor ist dabei wahrscheinlich die zentrale Komponente des Formular Editors.

Er enthält eine Liste von Komponenten (Content), die er in sein Layout einfügen kann, und Liste von Komponenten (Layouts), wie er diese Komponenten einfügen kann (untereinander, nebeneinander etc).

So erstellt der Layout Editor ein Formular. Als Content-Komponenten dienen die bestehenden Templates (templateId), die aus den Templategruppen ausgewählt werden können. Die Layout-Komponenten sind immer ähnlich (Tabs, Section, Row etc.).

Die Liste des Contents ist erweiterbar, so lassen sich neue Templategruppen aufnehmen oder Templates erstellen (dies öffnet einen zweiten Editor).

Eine Ebene tiefer werden die gleichen Komponenten verwendet, um a) die Liste der Templategruppen zu verwalten, b) die Liste der Templates zu einer Templategruppe zu erstellen.

Ein Template wird dann ebenfalls über den Layout Editor erstellt, als Content dienen die einzelnen Felder (fieldId), die in der entsprechenden Templategruppe vorhanden sind.

Der Layout-Editor erstellt dabei immer nur eine Layout-Komponente mit Inhalt, der Inhalt ist jedoch verlinkt, so dass Änderungen direkt auf allen Layouts greifen.

* + 1. Umsetzung
* EFRM-16: [backend] Möglichkeit bereitstellen, um Formular/Template/Felddefinitionen speichern zu können.
* [Nicht angelegt] [vue] Service um Editor Daten zu lesen und zu schreiben.
* [Nicht angelegt] [vue] Komponenten für den Editor

1. Feldtypen

Hier ist ein aktueller Stand zu den Typen bezüglich der Indexserver Aspekte.



Feedback Feldtypen 24.08.2020

**URL**

* Prüfung scheint nicht vollständig.
* Wie wird die Prüfung gemacht?
  + Der RegExp ist: /^((https?)|(elodms)|(ftp)):\/\/(www\\.)?[-a-zA-Z0-9@:%\_+.~#?&/=]+/
* Sinnvoll wäre ein editierbarer RegEx.

Done

* Eingegebene URL sollte anklickbar sein -> In Browser öffnen
  + Das ist nicht so einfachg, da es sich um ein Eingabefeld handelt, wenn man es anklickt will man ja den Wert editieren, nicht dem Link folgen. Man könnte rechts einen Button machen, der dann dem Link folgt oder einen Button der das Feld erst erditierbar macht, aber irgendwie muss man beide Fälle unterscheiden.

**Tel**

* Gängige Telefonnummern werden nicht akzeptiert. Was wird da geprüft?
  + ^\[0-9]{1,3}\.[0-9]{4,14}(?:x.+)?$
* Sinnvoll wäre ein editierbarer RegEx.
  + done

**Mail**

* Prüfung scheint ok.
* Anpassbarkeit ist zu diskutieren.
  + done
* Eingegebene E-Mail sollte anklickbar sein -> Mailto-Link öffnen
  + Siehe Url

**Integer, Boolean** – sehen ok aus

* Sollen negative Zahlen bei Integer Feldern möglich sein?
  + Aktuell nicht

**Number** / Fließkomma:

* Negative Zahlen als Minimum geben Fehlermeldung, sollten aber möglich sein



* + done
* Einstellbarkeit von Dezimal- und Tausenderzeichen fehlt noch
  + Hier ist die Frage woher diese Einstellung kommt (fest aus Sprachdatei, JavaClient Konfig oder der Formular Konfig?)
* Was bedeutet „renderAmount“?

Ist der Hacken gesetzt werden Tausenderseparatoren beim Verlassen des Feldes gesetzt.

**Date**

* DatePicker funktioniert bei mir gar nicht
* Bei Eingabe per Hand bleibt Date Picker offen
* Bei Fehleingaben kommt keine Fehlermeldung, es wir einfach nicht übernommen (Datum außerhalb erlaubten Bereich, 30.02.2020, …)
  + das ist Buefy verschuldet, wenn eine Eingabe nicht als Datum geparst werden kann, entfernt die Komponente den kompletten Wert beim Verlassen. Wir können das Parsen beeinflussen, aber nicht die Logik was der DatePicker macht, wenn er das Datum nicht erkennt.
* Konfiguration von Eingabeformanten und Ausgabeformat fehlen noch, derzeit geht nur TT.MM. YYYY
  + Auch hier ist die Frage wo das konfiguriert wird (Sprachdatei, aus JavaClient oder im Formular).

**KWL**

* Wie definiere ich eine Stichwortlist hierfür?

**Folgende Typen fehlen** noch komplett:

* Benutzer
* Relation
* Große Zahl / Geldbetrag

1. Stichwortlisten
   1. Statische Stichwortlisten
   2. Dynamische Stichwortlisten
   3. Benutzerlisten
   4. Lokalisierte Listen/Status Listen
   5. Maskenauswahl
2. Formular-Scripting

Im Formular gibt es an verschiedenen Stellen die Möglichkeit, durch Scripte individuelle Anpassungen zur Laufzeit vorzunehmen.

Scripte sind dabei erst einmal reine JavaScript-Code-Snippets, die als einfache Strings in JSON-Objekte gespeichert werden.

* 1. Allgemeines

Innerhalb eines Formulars werden Komponenten über einen apiName identifiziert. Falls diese nicht gesetzt ist, stehen für diese Komponente keine API-Methoden zur Verfügung (diese werden dann nicht registriert).

Eingabefelder verwenden als apiName automatisch den technischen Namen „name“, der auch dazu verwendet wird den Feldwert zu speichern.

* + 1. Problem: Mehrere Felder für einen Wert

Sind auf einem Formular mehrere Felder/Komponenten vorhanden, die sich auf den gleichen Wert beziehen, müssen diese mit unterschiedlichen apiNames ausgestattet werden, um diese einzeln per API ansprechen zu können. Aktuell überschreibt die Registrierung die API-Methode der vorher registrierten Komponente mit gleichem Namen.

* + 1. Problem: Namensvergabe durch Aspekte

Der Fallback zur Identifikation einer Feldkomponente ist der technische Name (Key des Wertes). Dieser Key beinhaltet für Aspekte die Aspekt-Relation und den Line-Names.

Die Aspekt-Relation ist jedoch abhängig von der Maske, deshalb kann ein Script nur auf Ebene der Maske definiert werden, da nur dadurch die Felder eindeutig benannt sind.

Idee: Scopes

Per Default haben die Scripte als Scope die Maske, d.h. die Scripte werden bezüglich einer Maske gespeichert, alle Formulare, die sich auf die Maske (Formgroup) beziehen bekommen die Scripte ausgeliefert.

Angedacht ist ein weiterer Scope-Aspekt, so dass die Scripte auf Aspekt-Ebene gespeichert werden und nur auf Inhalte des gleichen Aspektes zugreifen können.

TODO: Hier ist noch offen, wie genau die Variablen Bezeichnungen dann auf das ausgelieferte Formular übertragen werden können.

* + 1. Problem: Sub-Komponenten

Eine Komponente besteht aus Label, Input etc. Möchte man auf eine konkrete Unterkomponente zugreifen (nur das Inputfeld verstecken), dann muss man aktuell per Namenskonvention: apiName + „Input“ darauf zugreifen.

* 1. Script Events (Felder)

Es gibt verschiedene Typen von Events, die während der Laufzeit eines Formulars ausgeführt werden können.

Ein Event ist ein JSON-Objekt, das aus den folgenden Eigenschaften besteht:

* event: Name des Events.
* fct: Auszuführender Code.

Je nach Event kommen Felder hinzu oder sind optional.

* + 1. elo.event.Validation

Dieses Event wird von Feldern geworfen, wenn sich ihr Wert ändert. Das Event beinhaltet zusätzlich folgende Felder:

* apiNames: Array von Feldern auf die sich das Event bezieht. Wenn ein Feld aus der Liste geändert wurde, wird für dieses Feld die Validierung ausgeführt.
* fct: [Optional] Validierungsfunktion, sie muss true zurückliefern, damit das Feld valide ist, anonsten wird es als invalide betrachtet. In diesem Fall wird der Rückgabewert als Fehlermeldung betrachtet und ausgegeben.
* errorMsg: [Optional] Eine Fehlermeldung, die ausgegeben werden soll, wenn die Funktion nicht true zurückliefert, der Rückgabewert wird dann ignoriert.
* triggers: [Optional] Array von Feldnamen, die im Anschluss an dieses Feld ebenfalls validiert werden müssen (nötig um Feldabhängigkeiten zu validieren).

An die Funktion werden folgende Parameter übergeben:

* value: Interne Wert (aus dem Eingabestring konvertiert) des Feldes
* valueStr: String Wert (aktuelle Eingabe) des Feldes
* current: String Der Name des aktuellen Feldes (wichtig falls das Script mehrere apiNames enthält).
* form: Wert des Formulars (aktuell invalide Felder haben hier null als Wert)
* api: Allgemeines API Objekt (Komponenten, Hilfsfunktionen etc.)

Beispiel:

{

"event": "elo.event.Validation",

"apiNames": ["text2"],

"fct": "return !value || !!form.text1.get()", (text2 ist leer oder aber Feld text1 muss einen Wert haben)

"errorMsg": "Das Feld text1 muss gefüllt sein."

}

* + 1. elo.event.Formula

Formeln sind Scripte die auf bei Änderung eines Feldes einen Wert für ein Zielfeld setzen.

Eine Formel besteht aus:

* fct: Ein Script-Snippet-String, das den Code der Funktion beinhaltet.
* apiNames: Ein Array von Feldnamen, deren Änderung die Berechnung der Formel auslöst.
* target: Name des Feldes, das durch die Formelberechnung gefüllt werden soll.

An die Funktion werden folgende Parameter übergeben:

* value: Interner Wert (aus dem Eingabestring konvertiert) des Feldes
* valueStr: String Wert (aktuelle Eingabe) des Feldes
* current: String Der Name des aktuellen Feldes (wichtig falls das Script mehrere apiNames enthält).
* form: Wert des Formulars (aktuell invalide Felder haben hier null als Wert)
* api: Allgemeines API Objekt (Komponenten, Hilfsfunktionen etc.)

Beispiel:

{

"event": "elo.event.Formula",

"apiNames": ["text1", "text2"],

"target": "text3",

"fct": "return form.text1.get() + form.text2.get()"

}

Hier wird der Wert von text1 und text2 konkateniert (String + Operator) und in text3 geschrieben. Im Falle von Nummernfelder wäre dieses Script die Summe von text1 und text2.

Zyklische Formeln

Zu beachten sind zyklische Formeln: Werden zwei Felder durch den Wert des jeweils anderen beeinflusst (x = y +1; y = x+1) dann entsteht eine zyklische Formel. Solche Formeln laufen in einer Endlosschleife und beeinflussen massiv die JavaScript Performance der Webseite. Sie können vom Formular nicht automatisch entdeckt werden.

* 1. Script Events (Komponenten)

Einige Komponenten haben Events, die wiederum per Scripting implementiert werden können. In diesem Fall können Scripte auch asynchrone Anfragen durchführen, da die Events für die Berechnung des Formulars bzw. den Lebenszyklus der Validierung keine Rolle spielen. Sollten im Verlauf eines Events Formularwerte geändert werden, werden diese Änderungen wie Benutzereingaben behandelt und die Validierung ausgeführt.

* + 1. elo.event.Enter
* onEnter: (Auf allen Text-Eingabefelder (Autocomplete, Datum, Text)) Das Event wird ausgelöst, wenn der Fokus innerhalb des Eingabefeldes ist und die Enter Taste gedrückt wird.
* onClick: (Auf Buttons) Das Event wird ausgelöst, wenn auf die Schaltfläche geklickt wird.
* onBlur: (Alle Eingabefelder) Das Event wird ausgelöst, wenn der Fokus aus dem Feld entfernt wird (der Anwender das Feld mit Tab, einem Mausklick oder durch Fokussieren eines anderen Feldes verlässt).

Ein Event besteht aus:

* fct: Ein Script-Snippet-String, dass den Code der Funktion beinhaltet.

An die Funktion werden folgende Parameter übergeben:

* value: (Falls die Komponente einen Formularwert verwaltet) Interne Wert (aus dem Eingabestring konvertiert) des Feldes
* valueStr: (Falls die Komponente einen Formularwert verwaltet) String Wert (aktuelle Eingabe) des Feldes
* form: (Falls die Komponente ein Formular kennt) Wert des Formulars (aktuell invalide Felder haben hier null als Wert)
* api: Allgemeines API Objekt (Komponenten, Hilfsfunktionen etc.)

Beispiel:

"onEnter": {

"fct": "

let num1 = api.form['field\_api2\_number'].get();

let num2 = api.form['field\_api2\_number\_f2'].get();

console.info(\"test\", num1, num2);

if (num1 !== num2) {

api.form['field\_api2\_number\_f2'].set(num1);

} else {

let newNum = num1 ? num1 + 1 : num1 === 0 ? 1 : 0;

api.form['field\_api2\_number'].set(newNum);

}"

}

* 1. Script Events (Formular)
  2. API

Das API-Objekt wird in alle Script-Funktionen übergeben, es setzt sich wie folgt zusammen.

* value: (Falls vorhanden) Interne Wert (aus dem Eingabestring konvertiert) des Feldes. Hier ist zu beachten das der Typ des Wertes stark von Feld zu Feld variieren kann. Ein Datum wird als Datumsobjekt verwaltet, eine Zahl als Number etc.

Nicht validierte Felder (z.B. bei Formeln) haben immer einen Wert null.

* valueStr: (Falls vorhanden) String Wert (aktuelle Eingabe) des Feldes. Dies entspricht entweder der aktuellen Benutzereingabe im Feld (auch bei invaliden Feldern) oder ein normalisierter String, der den internen Wert visualisiert (Datum entsprechend der Benutzersprache).
* form: Dieses Objekt verwaltet alle im Formular bekannten Feldwerte, diese können mit ihrem Namen („name“) aufgerufen werden (form[„myFieldName“]) und bieten die Methoden:
  + set(val): Setzt den Wert eines Feldes und löst die Validierung aus.
  + get(): Liest den Wert eines Feldes, dabei wird null zurückgegeben, falls das Feld invalide ist.

[EFRM-56 - ScriptEventMixinFactory: Das form API Objekt sollte nicht innerhalb des api Objektes sein, sondern auf gleicher Höhe zu value, valueStr und api (wie bei anderen Script Aufrufen auch)](https://eloticksy.elo.com/browse/EFRM-56)

* api: Dieses Objekt setzt sich aus verschiedenen Punkten zusammen:
  + components: Alle auf der Seite registrierte Komponenten. Auf eine solche Komponente kann über ihren Namen zugegriffen werden (api.components[„myComponent“]. Entsprechend der Komponente stehen dann verschiedene Methoden zur Verfügung:

Deaktivieren: Alle Komponenten, die eine Interaktion (Setzen eines Wertes) erlauben, können deaktiviert werden. Dadurch wird diese durch eine graue Maske visuell verändert, und alle Möglichkeiten zur Interaktion unterbunden.

* + - disable: Deaktiviert die Komponente
    - enable: Aktiviert die Komponente
    - isDisabled: Prüft, ob die Komponente deaktiviert ist.

Ausblenden: Lässt eine Komponente verschwinden bzw. anzeigen. Die Komponente wird dabei nicht aus der Webseite entfernt, sondern nur per CSS versteckt. Je nach Komponente bleibt der Platz dafür erhalten (visibility:hidden) oder sie gibt ihren Platz frei (display:none). Typischerweise geben einzelne Felder ihren Platz nicht frei, damit das Layout erhalten bleibt, Feldcontainer jedoch verschwinden komplett, damit nachfolgende Container diesen Platz einnehmen können.

* + - hide: Versteckt die Komponente
    - show: Zeigt die Komponente an
    - isHidden: Prüft, ob die Komponente versteckt ist

Die Tab-Komponente hat eigene spezialisierte Methoden, um einzelne Tabs zu kontrollieren. Sie bekommen immer den Index des Tabs als Parameter übergeben.

Beim Anzeigen bzw. Aktivieren eines Tabs gibt ein zusätzlicher Paramter (Boolean) an, ob dieser Tab auch selektiert werden soll.

Beim Verstecken bzw. Deaktivieren eines Tabs gibt ein zusätzlicher Parameter (Zahl) an, welcher Tab selektiert werden soll, falls der aktuelle Tab aktiviert war.

* + - showTab: Zeigt einen Tab an
    - hideTab: Versteckt einen Tab
    - isTabHidden: Prüft, ob der Tab versteckt ist
    - enableTab: Aktiviert einen Tab
    - disableTab: Deaktiviert einen Tab
    - isTabDisabled: Prüft, ob ein Tab deaktiviert ist
  + forms: Alle registrierten Formulare. Diese bieten zusätzliche Funktionen:
    - save: Führt eine Validierung, falls diese erfolgreich ist, wird das Formular gespeichert. Dies kann nur bei Formularen verwendet werden, die gegen das Backend gespeichert werden, bei lokal geladenen Formularen hat dieser Aufruf keinen Effekt.
  1. Feedback

1. Formeln: Werden sie erst ausgeführt, nachdem der Validator lief? (Sowohl vorher als auch nachher muss möglich sein)

*Ich sehe hier keinen Grund eine Formelberechnung durchzuführen, wenn die dazu benötigten Felder nicht valide sind. Was wäre der Use-Case für so etwas.*

1. Alle fct Definitionen müssen in ein Skript auslagerbar sein.

*Das ist sicherlich eine sinnvolle Erweiterung: Im Script werden Funktionen mit Namen implementiert, im Formular kann man statt einer fct dann den Funktionesnamen angeben, der hier ausgeführt werden soll.*

1. OnBlur: Für was wird OnEnter benötigt? OnBlur würde reichen? (Wieso sollte man zwischen Enter und Tab unterscheiden?)

* OnEnter würde ich eher so verstehen, dass etwas passiert, sobald man in das Input-Feld klickt.

*OnEnter ist das Enter-Tastenevent, das wurde von Michael Weiler in einer ersten Analyse der bisherigen BS Scripte benannt. Der User ist in einem Feld, macht dort eine Eingabe und möchte dann einen Prozess (Validierung, Berechnung, Übertragung etc. anstoßen).*

*Sicherlich sinnvoll zu diskutieren wäre, ob man hier andere Tasten als Events hinzufügt.*

*Das Betreten eines Feldes wäre ein onFocus Event: Wird das benötigt?*

1. Wie sollen Tabellen im Formular behandelt werden?

*Dafür gibt es bereits eine Komponente, diese ist etwas komplexer und bedarf sicherlich einer eigenen Untersuchung.*

1. Feldwerte setzen muss programmatisch auch ohne Validierung möglich sein. (.set(val, false) ?)

*Was ist dafür der use-case? Ein Formular-Feld sollte nur einen Wert haben, wenn dieser Wert auch gültig ist, alles andere führt zu inkonsistenten Daten und großer Fehleranfälligkeit, da jedes Mal geprüft werden muss, ob der Wert darin jetzt gültig ist oder nicht. Ein ungültiges Feld hat deshalb immer den Wert null. Eventuell ist jedoch der valueStr gefüllt, dieser hat dann die ungültige Eingabe des Users.*

1. Es wäre schön, wenn die toString und valueOf Methoden der Felder einen vernünftigen Wert zurückgeben würden.

Beispiel:

     Datumsfeld explizit: “Heute ist der “ + form.mydatefield.get()             à “Heute ist der 11.08.2020“

     Datumsfeld implizit: “Heute ist der “ + form.mydatefield                        à “Heute ist der 11.08.2020“

*Das ist stark eine Frage des use-cases, wenn form.mydatefield statt des Datum Objektes nur einen String zurück gibt, dann wird daraus quasi ein ungültiger Wert (der String hat z.B. keine Relation zu einer Zeitzone).*

*Hilfsfunktionen die so etwas erledigen, machen allerdings Sinn: api.helpers.Renderer.date(form.mydatefield) oä. -> Issue Api Renderer*

1. Man hat im Skript Zugriff auf value und valueStr. Gibt es auch eine Möglichkeit der get-Methode eines Feldes mitzuteilen, ob man den Wert als value oder als valueStr erhalten möchte? Das ist notwendig.

*Aktuell nicht neben einem get() für den value, wäre ein getStr() sicher sinnvoll. -> Issue Form API erweitern*

Weitere Funktionen die noch fehlen:

* Platzhalter für ein Feld definieren (der graue Text, welcher als Eingabevorschlag angezeigt wird)

*Platzhalter sind aktuell bereits möglich (fehlen jedoch im Konfigurationsobjekt). -> Issue Platzhalter konfigurierbar machen*

* Zugriff auf den Labeltext eines Feldes. Label und Inputfeld sollten verbunden sein, um sowohl Feld, als auch Label gleichzeitig ein-/ausblenden zu können.

*Das wird sowieso als Einheit betrachtet:*

*if (value) { api.components.text2.hide() } else { api.components.text2.show() }*

*Hier wird jeweils text2 mit Label versteckt bzw angezeigt.*

* Tooltips eines Feldes müssen im laufenden Betrieb ein-/ ausblendbar und änderbar sein.

*Sind das die Validierungsmeldungen darunter oder sind wirklich Tooltips (onmouseover) gemeint?*

*Issue: Tooltips in DynamicField einbauen und konfigurierbar machen (tooltip: „Bitte Betrag eingeben.“)*

* Validatorennachrichten und Validatoren selbst müssen im laufenden Betrieb ein-/ ausschaltbar und änderbar sein.

*Das würde ich auch hinterfragen, am Ende muss ein Formular bezüglich seinen Inhaltes valide sein, daran ist nichts dynamisch. Eventuell gibt es komplizierte Custom Validator Funktionen, die mehrere Felder in Relation prüfen, aber valide sein müssen am Ende alle.*

* Folgende Attribute von Feldern müssen außerdem definiert und im laufenden Betrieb geändert werden können: readonly Ja/Nein, optional oder Pflichtfeld

*Disabled/Enabled sollte gehen -> Disabled Mixins nochmal prüfen*

*Optional/Pflichtfeld ist ein Validator (siehe oben Validator dynamisch ein und ausschalten).*

* Zu unterscheiden ist zwischen den Aktionen eines Anwenders und was man vielleicht programmatisch machen möchte: Nur weil ein Feld readonly für den Anwender ist, möchte ich trotzdem noch Werte über das Skript setzen können.

*Das geht.*

* Es muss auf jeden Fall einen Einstiegspunkt geben, der aufgerufen wird, wenn das Formular zum ersten Mal vollständig aufgebaut ist.

*Das gibt es noch nicht, das wird eventuell in der App gemacht, nicht in der Formular -Komponente.*

* Es muss dazwischen unterschieden werden können, ob ein Formular während eines Workflows angezeigt wird oder außerhalb. (z.B. api.formShowsActiveWorkflow, api.formShowsCompletedWorkflow, api.formShowsStandardSord)

*Das wäre auch eher Sache der App, dazu müsste es so etwas wie Metadaten geben, die in die API aufgenommen wird.*

*Gleiches gilt für aktueller Benutzer, Eintrags-ID etc.*

* Ein Einstiegspunkt analog zu nextClicked/saveClicked. Da in den Solutions soweit ich weiß nur nextClicked verwendet wird, wäre einfach ein Einstiegspunkt gut, der angesprungen wird, bevor das Formular geschlossen wird. Im Idealfall sollte man hier auch die Möglichkeit haben „false“ zurückzugeben, wenn der Anwender das Formular so nicht abschicken darf.

*Das ist auch im Rahmen der App anzusiedeln, dazwischen liegt ja noch das Ausbereiten der validen Daten, um diese dann senden zu können.*

* Tabellen sollten eine eigene Abstraktion erhalten. Anstatt addLineClicked und removeLineClicked dann tables.mytable.onRemoveLine, tables.mytable.afterRemoveLine,  tables.mytable.onAddLine und tables.myTable.afterAddLine

*Es wird nicht zwischen Tables und anderen Komponenten unterschieden:*

*api.components.myTable.removeLine, Es werden per Script nie Events registriert, diese werden ja an der Komponente konfiguriert.*

* Folgende Properties/Methoden wären praktisch für eine Tabelle: (Das Arbeiten mit den Indizes der Mapfelder ist nicht komfortabel und fehleranfällig!)
  + columns: gibt ein Array der Spaltenfelder zurück, welche in der Tabelle vorhanden sind.
  + size: Anzahl der Zeilen
  + getRow: gibt ein Objekt zurück, welches alle Felder der jeweiligen Zeilennummer beinhaltet.
  + insert: fügt eine neue Zeile in die Tabelle ein. Z.B. tables.mytable.insert({ myField1: „wert1“, anotherField: „wert2“ })
  + forEach: iteriert über jede Zeile und führt ein callback aus. Jeder call erhält als Parameter die aktuelle Zeile (getRow)

*Generell ist die Table ein eigenes Thema, da müssen wir noch ran.*

* das Arbeiten mit Stichwortlisten muss einfacher sein.
  + D.h. direkter Zugriff auf den Key und Wert (einzeln oder kombiniert) des Wertes aus einer lokalisierten Stichwortliste.

*Der Wert einer Stichwortliste ist (je nach Stichwortliste) das gesamte Objekt, also key und wert. Bekommt man also das Objekt, kann man sich alles weitere dort auslesen.*

* + Suche in der Stichwortliste nach möglichen Treffern programmatisch

*Was genau ist damit gemeint???*

* + Wenn ein Wert bereits in ein Inputfeld gesetzt wurde, darf er nicht vergessen, dass er aus einer Stichwortliste kommt. (Problem: Gespeicherter Wert wird in mehrsprachigen Systeme mit lokalisierten Stichwortlisten nicht in der aktiven Sprache angezeigt, sondern immer so wie abgespeichert.)

*Dafür soll es ja ein komplett neues Konzept geben „lokalisierte Stichwortlisten“*

Es sollte auf jeden Fall auch möglich sein Formgroups aus mehreren Solutions innerhalb eines einzelnen Formulars anzuzeigen.

*Formgroups würden sich eher auf eine Maske beziehen, wo diese Masken zum Einsatz kommen wäre dann noch zu klären.*

Beispiel: Wir entwickeln einen „E-Mail Editor“ als Solution. Der Editor sollte z.B. im Recruiting oder im Meeting Management gleichermaßen eingebettet werden können, ohne dass man die Forms in die jeweilige Solution kopieren muss.

*Wäre der Editor nicht dann sowieso in elo-commons oä drin?*

|  |  |  |  |
| --- | --- | --- | --- |
| Revisionsgeschichte dieses Dokuments | | | |
| Version | Datum | Bearbeiter | Änderungen |
|  |  |  |  |
|  |  |  |  |